



Vue.js 2

Wprowadzenie dla profesjonalistów

Adam Freeman



Tytuł oryginału: Pro Vue.js 2

Tłumaczenie: Krzysztof Rychlicki-Kicior

ISBN: 978-83-283-5499-9

Original edition copyright © 2018 by Adam Freeman.

All rights reserved.

Polish edition copyright © 2019 by HELION SA.

All rights reserved.

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

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione.

Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

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

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/vue2wp.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/vue2wp_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

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

*Mojej kochanej Żonie — Jacqui Griffyth
(a także Orzeszkowi).*



Spis treści

O autorze	15
O korektorze merytorycznym	17
Część I Zaczynamy pracę z Vue.js	19
Rozdział 1. Twoja pierwsza aplikacja w Vue.js	21
Przygotowanie środowiska programistycznego	21
Instalowanie Node.js	21
Instalowanie pakietu @vue/cli	22
Instalowanie narzędzia Git	23
Instalowanie edytora	23
Instalowanie przeglądarki	24
Tworzenie projektu	24
Struktura podkatalogów w projekcie	24
Uruchamianie narzędzi deweloperskich	25
Zamiana treści zastępczych	26
Dodawanie frameworka do obsługi stylów CSS	28
Stylowanie elementów HTML	29
Dodawanie treści dynamicznych	29
Wyświetlanie listy zadań	31
Dodawanie przycisku wyboru (checkbox)	33
Filtrowanie zakończonych zadań	34
Tworzenie nowych zadań	36
Trwałe przechowywanie danych	38
Ostatnie szlify	40
Podsumowanie	42

Rozdział 2. Zrozumieć Vue.js	43
Czy warto korzystać z Vue.js?	44
Zasada działania aplikacji wielostronowych	44
Zasada działania SPA	44
Złożoność aplikacji	46
Co muszę wiedzieć?	46
Jak skonfigurować swoje środowisko programistyczne?	46
Jaki jest układ treści w tej książce?	46
Część I. Zaczynamy pracę z Vue.js	47
Część II. Vue.js pod lupą	47
Część III. Zaawansowane funkcje Vue.js	47
Czy znajdę tu dużo przykładów?	47
Gdzie znajdę przykładowe kody?	49
Podsumowanie	49
Rozdział 3. Podstawy HTML i CSS	51
Przygotowania do rozdziału	51
Jak działają elementy języka HTML?	53
Element a jego treść	54
Jak działają atrybuty?	55
Analiza przykładowego dokumentu HTML	56
Jak działa Bootstrap?	58
Stosowanie podstawowych klas Bootstrapa	58
Stosowanie Bootstrapa do tworzenia siatki	60
Stosowanie Bootstrapa do stylowania tabel	60
Stosowanie Bootstrapa do stylowania formularzy	62
Podsumowanie	63
Rozdział 4. Elementarz JavaScriptu	65
Przygotowania do rozdziału	66
Stosowanie instrukcji	68
Tworzenie i używanie funkcji	68
Definicja funkcji z parametrami	70
Tworzenie funkcji zwracających wyniki	71
Przekazywanie funkcji przez argument	71
Zmienne i typy	72
Typy prymitywne	74
Operatory języka JavaScript	76
Instrukcje warunkowe	77
Operator równości a operator identyczności	77
Jawną konwersję typów	78
Obsługa tablic	79
Literaly tablicowe	80
Odczyt i modyfikacja zawartości tablicy	80
Przeglądanie zawartości tablicy	81
Operator rozwinięcia	81
Wbudowane metody do obsługi tablic	82

Obsługa obiektów	82
Literaly obiektowe	84
Stosowanie funkcji jako metod	85
Kopiowanie właściwości pomiędzy obiekttami	85
Moduły w języku JavaScript	86
Tworzenie i używanie modułów	86
Tworzenie wielu mechanizmów w jednym module	88
Łączenie wielu plików w jeden moduł	89
Zasady działania obietnic	90
Problemy z asynchronousznym wykonywaniem operacji	91
Przykład z użyciem obietnic	91
Uproszczenie kodu asynchronousznego	92
Podsumowanie	93
Rozdział 5. Sklep sportowy: prawdziwa aplikacja	95
Tworzenie projektu Sklep sportowy	95
Dodawanie dodatkowych pakietów	96
Przygotowanie REST-owej usługi sieciowej	98
Uruchamianie narzędzi projektowych	100
Tworzenie magazynu danych	101
Tworzenie magazynu produktów	103
Tworzenie listy produktów	104
Dodawanie listy produktów do aplikacji	106
Przetwarzanie cen	106
Obsługa stronicowania listy produktów	108
Obsługa wyboru kategorii	114
Zastosowanie REST-owej usługi sieciowej	117
Podsumowanie	119
Rozdział 6. Sklep sportowy: rozliczenie i zamówienia	121
Przygotowania do rozdziału	121
Tworzenie zastępczej treści dla koszyka	121
Konfiguracja trasowania adresów URL	122
Wyświetlanie trasowanego komponentu	123
Implementacja funkcji koszyka	124
Dodatkowy moduł w magazynie danych	125
Obsługa mechanizmu wyboru produktów	126
Wyświetlanie zawartości koszyka	128
Tworzenie globalnego filtru	131
Testowanie podstawowych funkcji koszyka	132
Utrwalanie koszyka	132
Dodawanie widżetu podsumowania koszyka	135
Obsługa rozliczenia i dodawania zamówień	137
Tworzenie i rejestracja komponentów rozliczenia	138
Dodawanie formularza validacji	141
Dodawanie pozostałych pól i validacji	144
Podsumowanie	147

Rozdział 7.	Sklep sportowy: skalowanie i administracja	149
	Przygotowania do rozdziału	149
	Obsługa dużej ilości danych	150
	Usprawnienie stronicowania	151
	Ograniczanie ilości danych pobieranych przez aplikację	152
	Obsługa wyszukiwania	157
	Praca nad funkcjami administracyjnymi	161
	Implementacja uwierzytelniania	161
	Dodawanie struktury komponentu administracyjnego	167
	Implementacja zarządzania zamówieniami	169
	Podsumowanie	172
Rozdział 8.	Sklep sportowy: administrowanie i wdrożenie	173
	Przygotowania do rozdziału	173
	Dodawanie funkcji administracyjnych	173
	Przedstawianie listy produktów	175
	Dodawanie treści zastępczej edytora i tras URL	177
	Implementacja edytora produktów	178
	Wdrażanie sklepu sportowego	181
	Przygotowanie aplikacji do wdrożenia	181
	Budowanie aplikacji do wdrożenia	185
	Testowanie aplikacji gotowej do wdrożenia	186
	Wdrożenie aplikacji	188
	Podsumowanie	190
Część II	Vue.js pod lupa	191
Rozdział 9.	Jak działa Vue.js?	193
	Przygotowania do rozdziału	193
	Dodawanie framework'a Bootstrap CSS	194
	Uruchamianie przykładowej aplikacji	194
	Tworzenie aplikacji za pomocą API modelu DOM	195
	Jak działa aplikacja w modelu DOM?	196
	Tworzenie obiektu Vue	198
	Stosowanie obiektu Vue	199
	Dodawanie funkcji obsługi zdarzenia	200
	Modyfikacja komunikatu	201
	Zasada działania obiektu Vue	202
	Komponenty w praktyce	203
	Rejestracja i wdrażanie komponentu	204
	Oddzielanie szablonu od kodu JavaScript	205
	Podsumowanie	207
Rozdział 10.	Projekty i narzędzia Vue.js	209
	Tworzenie projektu aplikacji Vue.js	209
	Konfiguracja lintera	212
	Zakończenie konfiguracji projektu	212
	Omówienie struktury projektu	213
	Omówienie katalogu z kodem źródłowym	214
	Omówienie katalogu pakietów	216

Omówienie narzędzi deweloperskich	218
Omówienie procesów komplikacji i transformacji	219
Omówienie serwera deweloperskiego HTTP	221
Omówienie mechanizmu zamiany modułów na gorąco	222
Omówienie wyświetlania błędów	224
Stosowanie lintera	226
Dostosowywanie reguł lintera	229
Debugowanie aplikacji	231
Analiza stanu aplikacji	231
Omówienie debuggera w przeglądarce	231
Konfiguracja narzędzi deweloperskich	233
Budowanie aplikacji do wdrożenia	233
Instalacja i zastosowanie serwera HTTP	236
Podsumowanie	237
Rozdział 11. Omówienie wiązań danych	239
Przygotowania do tego rozdziału	240
Omówienie składników komponentu	242
Omówienie elementu template	242
Omówienie elementu script	243
Omówienie elementu style	243
Zmiany komponentu w przykładowej aplikacji	243
Wyświetlanie wartości danych	244
Stosowanie złożonych wyrażeń w wiązaniach danych	247
Przeliczanie wartości we właściwościach obliczanych	249
Obliczanie wartości danych za pomocą metody	252
Formatowanie wartości danych za pomocą filtrów	255
Podsumowanie	260
Rozdział 12. Stosowanie podstawowych dyrektyw	261
Przygotowania do tego rozdziału	262
Ustawianie zawartości tekstowej elementu	263
Wyświetlanie czystego kodu HTML	265
Wyświetlanie wybranych elementów	267
Wyświetlanie wybranych elementów sąsiednich	268
Wybór fragmentów zawartości	270
Wybór wyświetlanych elementów za pomocą stylów CSS	272
Ustawianie atrybutów i właściwości elementu	274
Stosowanie obiektu do konfiguracji klas	276
Ustawianie pojedynczych stylów	277
Ustawianie innych atrybutów	279
Ustawianie wielu atrybutów	280
Ustawianie właściwości HTMLElement	281
Podsumowanie	283
Rozdział 13. Obsługa dyrektywy Repeater	285
Przygotowania do tego rozdziału	285
Przeglądanie tablicy	287
Stosowanie aliasu	289
Określanie klucza	291

Pobieranie indeksu elementu	293
Wykrywanie zmian w tablicy	296
Wyliczanie właściwości obiektu	298
Właściwości obiektu a kwestia kolejności	300
Powtarzanie elementów HTML bez źródła danych	302
Stosowanie właściwości obliczanych z dyrektywą v-for	303
Stronicowanie danych	303
Filtrowanie i sortowanie danych	305
Podsumowanie	307
Rozdział 14. Obsługa zdarzeń	309
Przygotowania do tego rozdziału	309
Obsługa zdarzeń	311
Omówienie zdarzeń i obiektów zdarzeń	312
Stosowanie metody do obsługi zdarzeń	313
Połączenie zdarzeń, metod i elementów powtarzanych	315
Nasłuchiwanie wielu zdarzeń z tego samego elementu	317
Stosowanie modyfikatorów obsługi zdarzeń	320
Zarządzanie propagacją zdarzeń	320
Zapobieganie duplikacji zdarzeń	326
Omówienie modyfikatorów zdarzeń myszy	327
Omówienie modyfikatorów zdarzeń klawiatury	328
Podsumowanie	330
Rozdział 15. Obsługa elementów formularzy	331
Przygotowania do tego rozdziału	331
Tworzenie dwukierunkowych wiązań modeli	333
Dodawanie wiązania dwukierunkowego	334
Dodawanie kolejnego elementu wejściowego	335
Upraszczanie wiązań dwukierunkowych	337
Wiiązania z elementami formularzy	338
Wiiązania do pól tekstowych	338
Wiiązania do przycisków opcji i wyboru	339
Wiiązania do elementów typu select	341
Stosowanie modyfikatorów dyrektywy v-model	343
Formatowanie wartości jako liczb	343
Opóźnianie aktualizacji	344
Usuwanie białych znaków	345
Wiiązania do różnych typów danych	346
Wybór tablicy elementów	346
Stosowanie własnych wartości w elementach formularza	348
Walidacja danych w formularzu	351
Definiowanie reguł walidacji	353
Stosowanie funkcji walidacji	354
Bieżące reagowanie na zmiany	357
Podsumowanie	358

Rozdział 16. Stosowanie komponentów	359
Przygotowania do tego rozdziału	359
Omówienie komponentów jako podstawowych składników aplikacji	361
Omówienie nazw komponentów i elementów dzieci	363
Wykorzystywanie możliwości komponentów w komponentach-dzieciach	365
Omówienie izolacji komponentów	366
Stosowanie propów w komponentach	368
Tworzenie własnych zdarzeń	373
Stosowanie slotów komponentów	376
Podsumowanie	381
Część III Zaawansowane funkcje Vue.js	383
Rozdział 17. Omówienie cyklu życia komponentu Vue.js	385
Przygotowania do tego rozdziału	386
Omówienie cyklu życia komponentu	388
Omówienie fazy tworzenia	389
Omówienie fazy montażu	390
Omówienie fazy aktualizacji	392
Omówienie fazy zniszczenia	398
Obsługa błędów komponentów	400
Podsumowanie	403
Rozdział 18. Luźno powiązane komponenty	405
Przygotowania do tego rozdziału	406
Tworzenie komponentu do wyświetlania produktu	408
Tworzenie komponentu edytora produktu	409
Wyświetlanie komponentów-dzieci	410
Omówienie wstrzykiwania zależności	411
Tworzenie usługi	411
Konsumowanie usługi za pomocą wstrzykiwania zależności	412
Przesłanianie usług pochodzących od przodków	413
Tworzenie reaktywnych usług	415
Zaawansowane wstrzykiwanie zależności	417
Stosowanie szyny zdarzeń	420
Wysyłanie zdarzeń za pomocą szyny zdarzeń	420
Odbieranie zdarzeń z szyny zdarzeń	421
Tworzenie lokalnych szyn zdarzeń	424
Podsumowanie	426
Rozdział 19. Stosowanie REST-owych usług sieciowych	427
Przygotowania do tego rozdziału	427
Przygotowanie serwera HTTP	428
Przygotowanie przykładowej aplikacji	429
Uruchamianie przykładowej aplikacji i serwera HTTP	432
Omówienie REST-owych usług sieciowych	433
Konsumowanie REST-owej usługi sieciowej	435
Obsługa danych odpowiedzi	435
Wykonywanie żądania HTTP	436

Otrzymywanie odpowiedzi	437
Przetwarzanie danych	438
Tworzenie usługi HTTP	440
Konsumowanie usługi HTTP	440
Dodawanie pozostałych operacji HTTP	441
Tworzenie usługi obsługi błędów	444
Podsumowanie	447
Rozdział 20. Stosowanie magazynu danych	449
Przygotowania do tego rozdziału	449
Tworzenie i używanie magazynu danych	452
Omówienie podziału na stan i mutacje	454
Udostępnianie magazynu danych Vuex	456
Stosowanie magazynu danych	456
Analiza zmian w magazynie danych	460
Definiowanie właściwości obliczanych w magazynie danych	461
Stosowanie gettera w komponencie	463
Przekazywanie argumentów do getterów	464
Wykonywanie operacji asynchronicznych	464
Otrzymywanie powiadomień o zmianach	468
Mapowanie funkcji magazynu danych w komponentach	471
Stosowanie modułów magazynu danych	474
Rejestrowanie i stosowanie modułu magazynu danych	475
Stosowanie przestrzeni nazw modułów	478
Podsumowanie	480
Rozdział 21. Komponenty dynamiczne	481
Przygotowania do tego rozdziału	482
Przygotowywanie komponentów do dynamicznego cyklu życia	483
Pobieranie danych aplikacji	483
Zarządzanie zdarzeniami obserwatora	484
Dynamiczne wyświetlanie komponentów	485
Przedstawianie różnych komponentów w elemencie HTML	486
Wybór komponentów za pomocą wiązania danych	486
Automatyczna nawigacja w aplikacji	490
Stosowanie komponentów asynchronicznych	494
Wylaczanie podpowiedzi wstępnego pobierania	497
Konfiguracja leniwego ładowania	498
Podsumowanie	501
Rozdział 22. Trasowanie URL	503
Przygotowania do tego rozdziału	503
Rozpoczynamy pracę z trasowaniem URL	505
Dostęp do konfiguracji trasowania	507
Stosowanie systemu trasowania do wyświetlania komponentów	507
Nawigowanie do innych adresów URL	510
Omówienie i konfiguracja dopasowania tras URL	513
Omówienie dopasowania i formatowania adresów URL	514
Stosowanie API historii HTML5 do trasowania	515

Stosowanie aliasu trasy	518
Pobieranie danych trasowania w komponentach	519
Dynamiczne dopasowywanie tras	522
Stosowanie wyrażeń regularnych do dopasowywania adresów URL	525
Tworzenie tras nazwanych	528
Obsługa zmian w nawigacji	531
Podsumowanie	534
Rozdział 23. Elementy związane z trasowaniem URL	535
Przygotowania do tego rozdziału	536
Obsługa elementów router-link	537
Wybór rodzaju elementu	538
Wybór zdarzenia nawigacji	541
Stylowanie elementów łącza routera	542
Tworzenie tras zagnieżdżonych	546
Planowanie układu aplikacji	547
Dodawanie komponentów do projektu	547
Definiowanie tras	548
Tworzenie elementów nawigacji	550
Testowanie klas zagnieżdżonych	551
Obsługa nazwanych elementów router-view	553
Podsumowanie	557
Rozdział 24. Zaawansowane trasowanie URL	559
Przygotowania do tego rozdziału	559
Stosowanie odrębnych plików dla powiązanych tras	560
Ochrona tras	562
Definiowanie globalnych strażników nawigacji	562
Definiowanie strażników dla konkretnych tras	566
Definiowanie strażników tras dla komponentów	570
Ładowanie komponentów na żądanie	577
Wyświetlanie komponentu z komunikatem ładowania	578
Tworzenie komponentów bez obsługi trasowania	582
Podsumowanie	585
Rozdział 25. Przejścia	587
Przygotowania do tego rozdziału	587
Tworzenie komponentów	589
Konfiguracja trasowania URL	592
Tworzenie elementów nawigacji	592
Rozpoczynamy pracę z przejściami	594
Omówienie klas przejść i przejść CSS	596
Omówienie sekwencji przejścia	597
Stosowanie biblioteki do obsługi animacji	598
Przełączanie pomiędzy wieloma elementami	599
Stosowanie przejścia do elementów z trasowaniem URL	601
Stosowanie przejścia podczas pojawiania się elementu	603
Stosowanie przejść dla zmian w kolekcji	604

Stosowanie zdarzeń przejść	606
Stosowanie zdarzeń początkowych i końcowych	608
Przyciąganie uwagi do innych zmian	609
Podsumowanie	612
Rozdział 26. Rozszerzanie możliwości Vue.js	613
Przygotowania do tego rozdziału	614
Tworzenie własnych dyrektyw	616
Omówienie zasady działania dyrektyw	618
Stosowanie wyrażeń własnych dyrektyw	620
Stosowanie argumentów własnej dyrektywy	621
Stosowanie modyfikatorów własnej dyrektywy	622
Komunikacja między funkcjami haków	624
Dyrektyny jednofunkcyjne	625
Tworzenie domieszek komponentów	626
Tworzenie wtyczki Vue.js	629
Tworzenie wtyczki	632
Stosowanie wtyczki	633
Podsumowanie	635
Skorowidz	637



O autorze



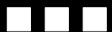
Adam Freeman to doświadczony specjalista IT, który pełnił funkcje kierownicze w wielu firmach, ostatnio jako dyrektor ds. technologicznych i dyrektor ds. operacyjnych w międzynarodowym banku. Po przejściu na emeryturę Adam poświęca czas na pisanie i biegi długodystansowe.



O korektorze merytorycznym

Fabio Claudio Ferracchiati jest starszym konsultantem i starszym analitykiem/programistą, doświadczonym w zakresie technologii firmy Microsoft. Obecnie pracuje dla firmy BluArancio (www.bluarancio.com). Posiada tytuł Microsoft Certified Solution Developer w zakresie .NET, a także Microsoft Certified Application Developer w zakresie .NET i Microsoft Certified Professional. Fabio jest wziętym autorem i korektorem merytorycznym. Przez ostatnie dziesięć lat napisał wiele artykułów dla włoskich i międzynarodowych czasopism, a także był współautorem ponad dziesięciu książek dotyczących różnych aspektów informatyki.

CZĘŚĆ I



Zaczynamy pracę z Vue.js



ROZDZIAŁ 1.

Twoja pierwsza aplikacja w Vue.js

Najprostszym sposobem na rozpoczęcie przygody z Vue.js jest zabranie się do pracy. W tym rozdziale pokażę Ci, jak przebiega proces tworzenia prostej aplikacji na przykładzie listy zadań do zrobienia. W rozdziałach 5. – 8. zajmiemy się opracowaniem bardziej złożonej i bliżej rzeczywistości aplikacji, ale na razie ten prosty przykład pozwoli Ci dobrze zrozumieć zasadę działania aplikacji Vue.js i ich podstawowe funkcje. Nie martw się, jeśli nie zrozumiesz któregoś z omawianych zagadnień — najważniejsze, abyś przyswoił sobie zasadę działania aplikacji Vue.js, a omówieniem szczegółów zajmiemy się w kolejnych rozdziałach.

-
- **Uwaga** Jeśli chcesz zapoznać się z bardziej tradycyjnym opisem możliwości Vue.js, przejdź do drugiej części tej książki, gdzie szczegółowo omawiam poszczególne mechanizmy dostępne w Vue.
-

Przygotowanie środowiska programistycznego

Przed rozpoczęciem pisania kodu musimy zainstalować niezbędne narzędzia. W kolejnych podrozdziałach omówię wszystkie aplikacje niezbędne do stworzenia naszej pierwszej aplikacji w Vue.js.

Instalowanie Node.js

Narzędzia używane w trakcie tworzenia aplikacji Vue.js wymagają do działania technologii Node.js, nazywanej też po prostu Node. Technologia ta istnieje na rynku od 2009 roku. Node.js to proste i wydajne środowisko uruchomieniowe przeznaczone do uruchamiania aplikacji serwerowych tworzonych w języku JavaScript. Node.js powstał na bazie silnika języka JavaScript stosowanego w przeglądarce Chrome, jednak nie przeszkadza mu to w wykonywaniu kodu poza środowiskiem tej przeglądarki.

Node.js zdobył popularność jako narzędzie do tworzenia aplikacji serwerowych, ale w tej książce stosujemy go z uwagi na jego znaczenie w tworzeniu nowej generacji wieloplatformowych narzędzi deweloperskich. Rozsądne decyzje projektowe podjęte przez zespół Node.js w połączeniu z wieloplatformowym środowiskiem uruchomieniowym dostarczonym przez Chrome daly nowe możliwości w zakresie tworzenia narzędzi deweloperskich. Mówiąc krótko, Node.js stał się kluczowy w tworzeniu aplikacji webowych.

Niezwyczajne ważne jest, abyś korzystał z tej samej wersji Node.js co ja. Mimo że Node to technologia dojrzała i stabilna, od czasu do czasu zdarzają się duże zmiany, które mogą uniemożliwić uruchomienie przykładów z tej książki.

W niniejszej książce stosuję wersję 8.11.2, która w momencie pisania była najnowszą wersją o przedłużonym wsparciu (ang. *Long-Term Support*). Być może, gdy będziesz czytać te słowa, będzie dostępna już nowsza wersja, jednak zdecydowanie zalecam stosowanie właśnie wersji 8.11.2. Kompletne wydanie, zawierające instalatory dla systemów Windows i macOS, a także skompilowane paczki dla innych systemów znajdziesz na stronie <https://nodejs.org/dist/v8.11.2>.

Po zakończonej instalacji upewnij się, że pliki wykonywalne Node.js zostały dodane do systemowej ścieżki. Po wszystkim wykonaj polecenie jak w listingu 1.1.

Listing 1.1. Sprawdzanie wersji Node.js

```
node -v
```

Jeśli wszystko poszło zgodnie z planem, powinien pojawić się numer wersji:

```
v8.11.2
```

Instalator Node.js zainstaluje także aplikację Node Package Manager (NPM — menedżer pakietów Node), która służy do zarządzania zależnościami (pakietami) w projekcie. Wykonaj polecenie z listingu 1.2, aby upewnić się, że NPM jest dostępny.

Listing 1.2. Weryfikacja działania programu NPM

```
npm -v
```

Ponownie powinien pokazać się numer wersji, ale inny niż w przypadku samego polecenia node:

```
5.6.0
```

Instalowanie pakietu @vue/cli

Pakiet @vue/cli pozwala na tworzenie projektów Vue.js i zarządzanie nimi. Teoretycznie nie trzeba z niego korzystać, ale jest to pakiet na tyle przydatny w codziennej pracy z Vue, że warto go zainstalować.

- **Uwaga** W trakcie pisania tych słów pakiet @vue/cli był dostępny w wersji beta. Być może przed wydaniem finalnej wersji zostaną w nim wprowadzone pewne zmiany, jednak główne funkcje powinny pozostać niezmienione. W przypadku jakichkolwiek istotnych zmian sprawdź erratę do tej książki na stronie <https://github.com/Apress/pro-vue-js-2>.

Aby zainstalować pakiet @vue/cli, otwórz wiersz poleceń i wykonaj polecenie z listingu 1.3. Jeśli korzystasz z systemu Linux lub macOS, to do wykonania tego polecenia możesz potrzebować sudo.

Listing 1.3. Instalacja pakietu Vue Tools

```
npm install --global @vue/cli
```

Instalowanie narzędzia Git

System kontroli wersji Git jest niezbędny do skorzystania z niektórych pakietów wymaganych do tworzenia aplikacji w Vue.js. Dla systemów Windows i macOS zostały opracowane instalatory, dostępne na stronie <https://git-scm.com/downloads> (w przypadku systemu macOS może być konieczna zmiana ustawień bezpieczeństwa, ponieważ instalator nie został podpisany przez jego twórców).

Git jest zawarty domyślnie w większości dystrybucji systemu Linux. Jeśli chcesz pobrać najnowszą wersję tego narzędzia, zapoznaj się z instrukcjami zawartymi na stronie <https://git-scm.com/download/linux>. Na przykład w systemie Ubuntu (z którego sam korzystam) wystarczy wykonać polecenie z listingu 1.4.

Listing 1.4. Instalacja narzędzia Git

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

Po zakończeniu instalacji otwórz wiersz poleceń i wykonaj polecenie z listingu 1.5.

Listing 1.5. Weryfikacja narzędzia Git

```
git --version
```

Wykonanie tego polecenia spowoduje wyświetlenie wersji narzędzia Git. W momencie pisania tej książki w systemach Windows i Linux najnowszą wersją była wersja 2.17, a dla systemu macOS — 2.16.3.

Instalowanie edytora

Aplikacje Vue.js można tworzyć w dowolnym edytorze tekstowym, przy czym niektóre z nich oferują specjalne wsparcie dla Vue.js, w tym podświetlanie składni i wyrażeń. Jeśli nie masz swojego ulubionego narzędzia, zapoznaj się z tabelą 1.1, w której opisuję popularne edytory dla Vue. Wybór edytora nie ma wpływu na wykonanie dalszych przykładów — skup się przede wszystkim na własnej wygodzie pracy.

Tabela 1.1. Popularne edytory z dodatkowym wsparciem dla Vue.js

Nazwa	Opis
Sublime Text	Komercyjny, wieloplatformowy edytor, który oferuje wsparcie dla większości istniejących języków programowania, frameworków i platform za pomocą systemu paczek. Więcej szczegółów znajdziesz na stronie www.sublimetext.com .
Atom	Darmowy, otwarty, wieloplatformowy edytor, w którym szczególny nacisk położono na możliwość dostosowywania i rozszerzania. Więcej szczegółów znajdziesz na stronie atom.io .
Brackets	Darmowy, otwarty edytor stworzony przez Adobe. Więcej szczegółów znajdziesz na stronie brackets.io .
Visual Studio Code	Darmowy, otwarty, wieloplatformowy edytor opracowany przez Microsoft, w którym to edytorze istotne znaczenie ma rozszerzalność. Więcej szczegółów znajdziesz na stronie code.visualstudio.com .
Visual Studio	To główne narzędzie deweloperskie firmy Microsoft. Istnieje wiele różnych wersji (darmowych i komercyjnych), a poza samym edytorem programista otrzymuje szereg dodatkowych narzędzi, które są pomocne w pracy z technologiami Microsoftu.

Instalowanie przeglądarki

Na zakończenie musimy wybrać przeglądarkę internetową, z której będziemy korzystać w trakcie tworzenia aplikacji. Wszystkie bieżące wersje popularnych przeglądarek oferują dobre wsparcie dla programistów i nadają się do pracy z Vue.js, ale warto pamiętać, że dla przeglądarek Chrome i Firefox stworzono rozszerzenie o nazwie `vue-devtools`, które pozwala na analizę stanu aplikacji Vue.js, co ma duże znaczenie w pracy nad złożonymi projektami. Więcej szczegółów na temat instalacji tego narzędzia znajdziesz na stronie <https://github.com/vuejs/vue-devtools>. W tej książce stosuję przeglądarkę Google Chrome i do tej samej przeglądarki chcę zachęcić również i Ciebie.

Tworzenie projektu

Tworzenie projektu i zarządzanie nim odbywa się za pomocą wiersza poleceń. Otwórz go, przejdź do wybranej lokalizacji, a następnie wykonaj polecenie przedstawione w listingu 1.6, aby utworzyć projekt.

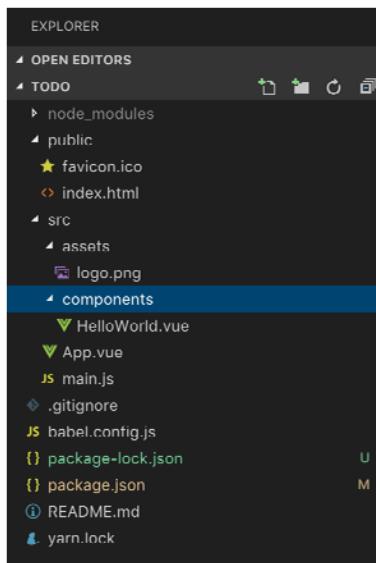
Listing 1.6. Tworzenie projektu

```
vue create todo --default
```

Polecenie `vue` zostało dodane w trakcie instalacji pakietu `@vue/cli` (listing 1.3). Powyższe polecenie tworzy nowy projekt o nazwie `todo`, który zostanie utworzony w katalogu o tej samej nazwie. Poza przygotowaniem struktury podkatalogów w projekcie narzędzie pobierze i zainstaluje wszystkie zależności, co może chwilę potrwać z uwagi na dużą liczbę pakietów koniecznych do uruchomienia najprostszej nawet aplikacji.

Struktura podkatalogów w projekcie

Otwórz katalog `todo` za pomocą wybranego edytora, a następnie przejdź do struktury projektu przedstawionej na rysunku 1.1. W moim przypadku jest to edytor Visual Studio, dlatego jeśli korzystasz z innego edytora, wygląd projektu może nieznacznie różnić się od mojego.



Rysunek 1.1. Struktura projektu

Struktura projektu może na początku wydawać się nieco skomplikowana, ale jestem przekonany, że pod koniec lektury tej książki będziesz zaznajomiony z wszystkimi plikami i katalogami, a także ich przeznaczeniem. W tabeli 1.2 krótko opisuję pliki najważniejsze z punktu widzenia tego rozdziału. Szczegółowo strukturą projektu Vue.js zajmę się w rozdziale 10.

Tabela 1.2. Najważniejsze pliki w projekcie

Nazwa	Opis
<i>public/index.html</i>	Ten plik HTML jest wczytywany przez przeglądarkę. W obrębie tego dokumentu występuje element, w ramach którego zostanie wyświetlona aplikacja, a także element <i>script</i> , który wczyta pliki aplikacji.
<i>src/main.js</i>	W tym pliku JavaScript ma miejsce konfiguracja aplikacji Vue.js. W pliku tym rejestruje się także wszelkie pakiety, które są niezbędne do działania aplikacji — liczne przykłady znajdziesz w kolejnych rozdziałach.
<i>src/App.vue</i>	Ten plik zawiera komponent Vue.js, w ramach którego znajdziesz: <ul style="list-style-type: none"> • kod komponentu w języku HTML, renderowany w przeglądarce użytkownika, • kod JavaScript niezbędny do prawidłowego wyrenderowania kodu HTML, • kod CSS, który jest odpowiedzialny za ostylowanie elementu HTML. Komponenty stanowią podstawowe struktury, z których tworzy się aplikację Vue.js, dlatego w niniejszej książce będziemy korzystać z nich niezwykle często.
<i>src/assets/logo.png</i>	Katalog <i>assets</i> zawiera treści statyczne, np. obrazki. W nowym projekcie znajdziesz plik <i>logo.png</i> , który zawiera logo Vue.js.

Uruchamianie narzędzi deweloperskich

Gdy tworzysz projekt za pomocą narzędzia vue, otrzymujesz dostęp do kompletnego zestawu narzędzi deweloperskich, dzięki czemu projekt można skompilować, spakować i obsłużyć za pomocą przeglądarki. Przejdz do wiersza poleceń, po czym wykonaj polecenia z listingu 1.7, aby przejść z kolei do katalogu todo i uruchomić narzędzia deweloperskie.

Listing 1.7. Uruchamianie narzędzi deweloperskich

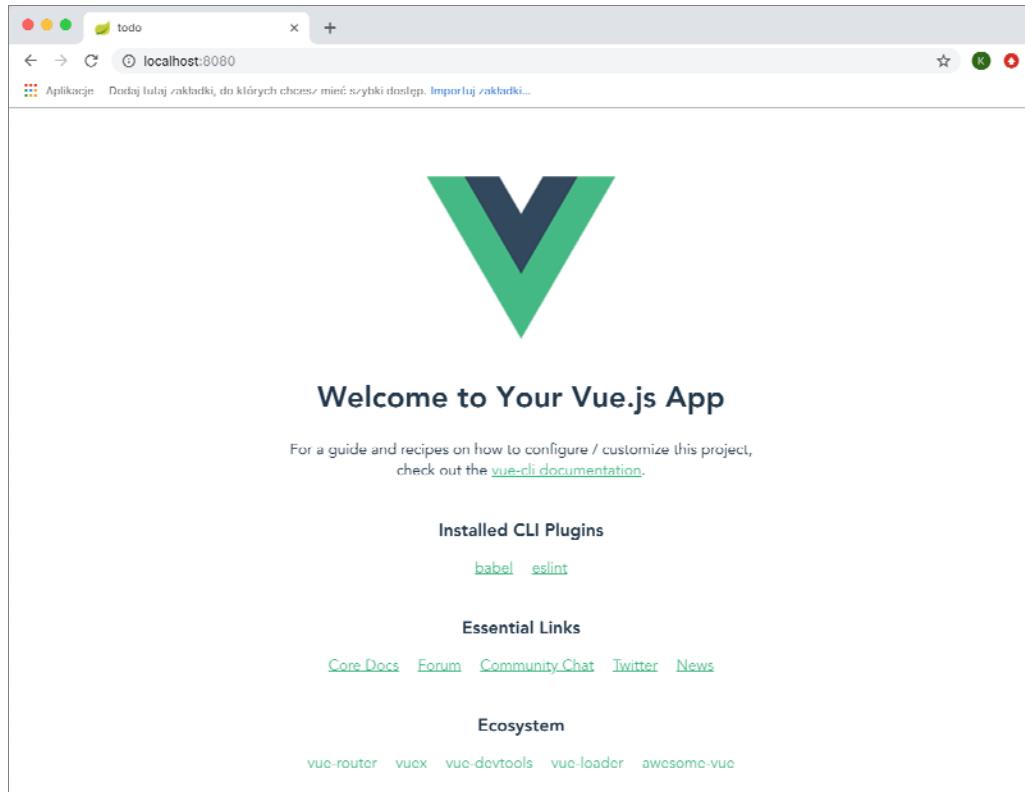
```
cd todo
npm run serve
```

Na samym początku, w trakcie uruchamiania narzędzi deweloperskich, konieczne jest przeprowadzenie procesu inicjalizacji, który może zajść dłuższą chwilę. Nie przejmuj się tym, ponieważ proces ten jest konieczny tylko na początku każdej sesji deweloperskiej.

Po uruchomieniu aplikacji powinieneś zobaczyć komunikat podobny do poniższego. Dzięki niemu dowiesz się, że aplikacja działa, a także o tym, z którego portu należy skorzystać w celu nawiązania połączenia.

```
App running at:
 - Local:  http://localhost:8080/
 - Network: http://192.168.0.77:8080/
 Note that the development build is not optimized.
 To create a production build, run npm run build.
```

Domyślny port to 8080, jednak jeśli jest on zajęty, serwer zajmie inny port. Otwórz przeglądarkę, a następnie wpisz adres <http://localhost:8080> (lub inny adres URL, zgodnie z treścią komunikatu, który zobaczysz). W ten sposób powinieneś uzyskać efekt jak na rysunku 1.2 (treści zastępcze mogą się nieco różnić, zważywszy na fakt, że narzędzia deweloperskie są stale rozwijane; dlatego nie przejmuj się, jeśli zawartość Twojego ekranu jest nieznacznie inna).



Rysunek 1.2. Uruchamianie przykładowej aplikacji

Zamiana treści zastępczych

Podstawą każdej aplikacji Vue.js są komponenty, definiowane w plikach z rozszerzeniem `.vue`. Oto treść komponentu App, którą znajdziesz w pliku `src/App.vue`:

```
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js App" />
  </div>
</template>
<script>
  import HelloWorld from './components/HelloWorld.vue'
  export default {
    name: 'app',
    components: {
      HelloWorld
    }
  }
</script>
```

```

        }
    }
</script>
<style>
    #app {
        font-family: 'Avenir', Helvetica, Arial, sans-serif;
        -webkit-font-smoothing: antialiased;
        -moz-osx-font-smoothing: grayscale;
        text-align: center;
        color: #2c3e50;
        margin-top: 60px;
    }
</style>

```

Komponent składa się z trzech części:

- elementu template zawierającego kod HTML, który zostanie zaprezentowany użytkownikowi;
- elementu script, w którym znajduje się kod JavaScript odpowiedzialny za obsługę szablonu;
- elementu style zawierającego niezbędne style CSS.

Vue.js łączy elementy template, script i style, aby stworzyć treść zastępczą, widoczną na rysunku 1.2.

Listing 1.8. zawiera zmodyfikowany element template i uproszczony element script. Usunąłem także element style. Wprowadzone zmiany pozwolą mi łatwo rozpoczęć tworzenie przykładowej aplikacji, bez konieczności uwzględniania w niej istniejącego, przykładowego kodu.

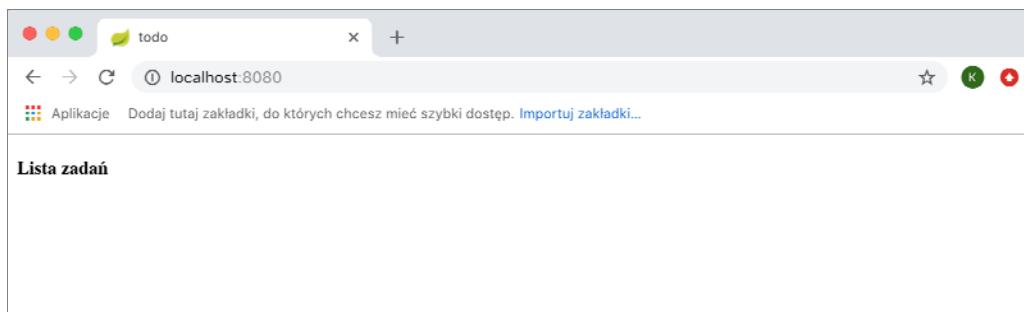
Listing 1.8. Usuwanie treści zastępczej w pliku src/App.vue

```

<template>
    <div id="app">
        <h4>
            Lista zadań
        </h4>
    </div>
</template>
<script>
    export default {
        name: 'app'
    }
</script>

```

Po zapisaniu zmian aplikacja zostanie automatycznie skompilowana ponownie, a przeglądarka odświeży treści na stronie, dzięki czemu otrzymamy efekt jak na rysunku 1.3.



Rysunek 1.3. Usunięcie treści zastępczej

Dodawanie framework'a do obsługi stylów CSS

Zmodyfikowana aplikacja nie wygląda zbyt dobrze. Mimo że komponenty mogą zawierać style CSS, zdecydowanie wolę skorzystać z framework'a do obsługi stylów CSS, dzięki któremu będę w stanie stylować elementy HTML w spójny sposób. W tej książce stosuję framework Bootstrap. Zatrzymaj narzędzia deweloperskie za pomocą kombinacji klawiszy *Ctrl+C*, a następnie wykonaj polecenie z listingu 1.9 w katalogu *todo*. W ten sposób dodasz framework Bootstrap do swojego projektu.

Listing 1.9. Dodawanie Bootstrapa do projektu

```
npm install bootstrap@4.0.0
```

W trakcie instalacji Bootstrapa mogą pojawić się ostrzeżenia o niespełnieniu pewnych zależności — nie musisz się nimi przejmować. Aby dodać plik CSS Bootstrapa do projektu, musimy zmodyfikować plik *src/main.js* jak w listingu 1.10.

Listing 1.10. Dodawanie Bootstrapa do pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

import "bootstrap/dist/css/bootstrap.min.css";
new Vue({
  render: h => h(App)
}).$mount('#app')
```

Instrukcja `import` odpowiada za dołączenie arkusza stylów CSS należącego do framework'a Bootstrap (dokładny opis instrukcji `import` znajduje się w rozdziale 4.). Nie martw się, jeśli nie miałeś do czynienia z Bootstrapem, ponieważ w rozdziale 3. znajdziesz bardziej szczegółowy opis jego możliwości.

Jak wybrać framework CSS do projektów Vue.js?

Wiele popularnych frameworków CSS — w tym Bootstrap — poza zwykłymi stylami CSS oferuje wsparcie dla tworzenia interaktywnych komponentów na poziomie języka JavaScript. Te funkcje języka JavaScript często wymagają dodatkowych bibliotek (np. jQuery), aby uzyskać dostęp do wybranych elementów dokumentu HTML. Może to prowadzić do konfliktów z używanym w projekcie Vue.js. W związku z tym niezwykle ważne jest, aby stosować wyłącznie style CSS dostarczone przez framework oraz wybrać framework napisany z myślą o Vue.js.

Pierwsze podejście — stosowanie wyłącznie stylów CSS — znajdziesz w niniejszej książce. Dzięki temu mogę korzystać z dobrze znanego mi frameworka, a ponadto nie muszę obawiać się o powiązanie funkcji dostarczonych przez Vue.js z mechanizmami stylowania. Taka sytuacja ma miejsce w przypadku stosowania frameworków dostosowanych do pracy z Vue.js, ponieważ działają one na bazie opisywanych przeze mnie funkcji, co koniec końców znacząco skomplikowałoby przykłady.

Jeśli chcesz skorzystać z frameworka dostosowanego do Vue.js, polecam Vuetify (<http://vuetifyjs.com>). Można także skorzystać z projektu, który dostosowuje Bootstrapa do projektów Vue.js — Bootstrap-Vue (<https://bootstrap-vue.js.org>).

Stylowanie elementów HTML

Skoro Bootstrap stał się integralną częścią naszej aplikacji, możemy ostylować zawartość elementu template naszego komponentu, aby nieco poprawić jego wygląd (listing 1.11).

Listing 1.11. Stylowanie zawartości pliku *src/App.vue*

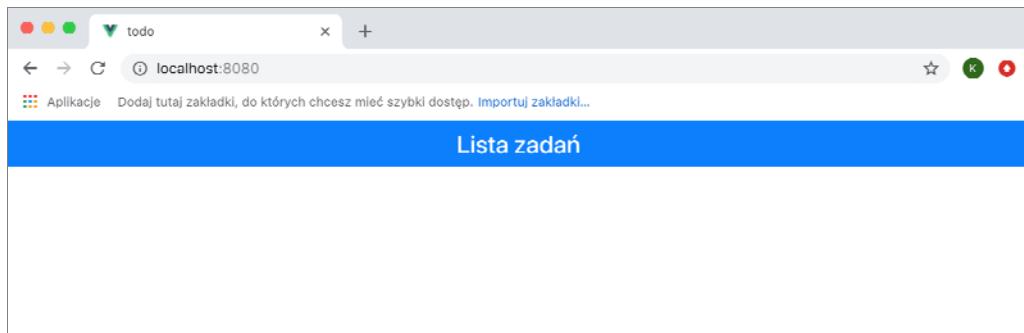
```
<template>
  <div id="app">
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań
    </h4>
  </div>
</template>
<script>
  export default {
    name: 'app'
  }
</script>
```

Wykonaj polecenie zawarte w listingu 1.12 w podkatalogu *todo*, aby zrestartować aplikację.

Listing 1.12. Restartowanie aplikacji

```
npm run serve
```

Przejdź na stronę <http://localhost:8080>, a uzyskasz efekt zbliżony do pokazanego na rysunku 1.4. Jeśli nic się nie zmieniło, ręcznie odśwież okno przeglądarki.



Rysunek 1.4. Stylowanie zawartości strony HTML

Dodawanie treści dynamicznych

Kolejnym krokiem w rozwoju aplikacji będzie wyświetlenie dynamicznej treści użytkownikowi. Listing 1.13 zawiera kilka zmian — zmodyfikowałem komponent App tak, aby wyświetlał dynamiczne dane.

Listing 1.13. Wyświetlanie danych w pliku *src/App.vue*

```
<template>
  <div id="app">
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika {{name}}
    </h4>
  </div>
</template>
```

```

        </h4>
    </div>
</template>
<script>
    export default {
        name: 'app',
        data() {
            return {
                name: "Adam"
            }
        }
    }
</script>

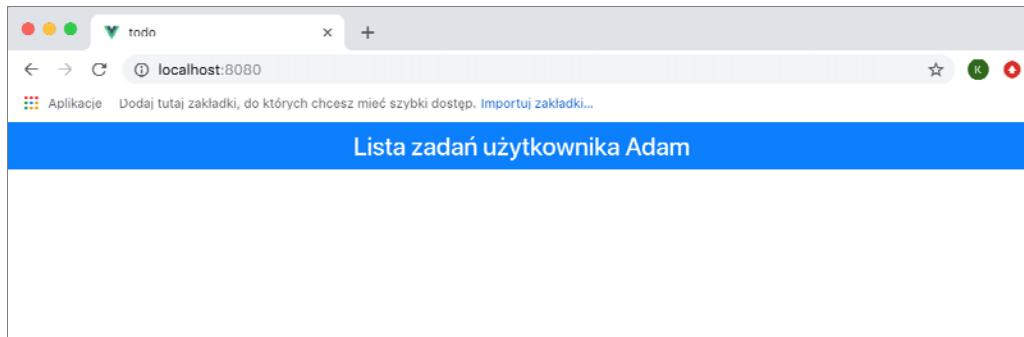
```

W elemencie template pojawia się nowa konstrukcja — wiązanie danych (ang. *data binding*). Dzięki temu w momencie generowania treści HTML Vue.js wstawi wartość w sposób dynamiczny. Ten rodzaj wiązania nosi nazwę **interpelacji tekstu** (ang. *text interpolation binding*), ponieważ wartość zostanie wstawiona jako tekst w elemencie HTML. W języku angielskim stosuje się także pojęcie *mustache binding* (dosł. „wiązanie wąsatów”) z uwagi na to, że podwójne nawiasy klamrowe, oznaczające wiązanie, wyglądają jak jeden z rodzajów wąsów.

Wiązanie danych informuje Vue.js o konieczności wstawienia wartości właściwości name do elementu h4 w momencie wygenerowania kodu HTML. Wartość name jest udostępniana dzięki zmianie, którą wprowadziłem w elemencie script w listingu 1.13. Właściwość data dostarcza dane do szablonu, dlatego w momencie przetwarzania wiązania danych Vue.js sprawdza tę właściwość, aby znaleźć wartość name.

-
- **Wskazówka** Nie przejmuj się nietypową składnią funkcji data z listingu 1.13. Wkrótce, gdy nabierzesz nieco doświadczenia w pracy z Vue.js, przyzwyczaisz się do niej. Więcej szczegółów na jej temat znajdziesz w rozdziale 11.
-

Po zapisaniu pliku *App.vue* narzędzia deweloperskie zaktualizują zawartość aplikacji, a okno przeglądarki zostanie odświeżone, dzięki czemu powinieneś zobaczyć efekt jak na rysunku 1.5.



Rysunek 1.5. Wyświetlanie wartości danych

Zwróć uwagę, że nie musisz odświeżać ręcznie okna przeglądarki za każdym razem, gdy wprowadzasz zmiany w plikach. Narzędzia Vue.js obserwują pliki projektu i w momencie zaistnienia zmian przeglądarka zostaje odświeżona. Więcej informacji na temat narzędzi deweloperskich znajdziesz w rozdziale 10.

Wyświetlanie listy zadań

Vue.js obsługuje różne rodzaje wiązań danych, dzięki którym generowanie dynamicznych treści jest możliwe na różne sposoby — przedstawiona przed chwilą interpolacja tekstu stanowi tylko jeden przykład. Kolejnym krokiem w rozwoju naszej aplikacji będzie utworzenie kolekcji obiektów reprezentujących nasze zadania do zrobienia, a następnie wyświetlenie jej jak w listingu 1.14.

Listing 1.14. Wyświetlanie listy zadań (plik src/App.vue)

```
<template>
  <div id="app">
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika {{name}}
    </h4>
    <div class="container-fluid p-4">
      <div class="row">
        <div class="col font-weight-bold">Zadanie</div>
        <div class="col-2 font-weight-bold">Zakończono?</div>
      </div>
      <div class="row" v-for="t in tasks" v-bind:key="t.action">
        <div class="col">{{t.action}}</div>
        <div class="col-2">{{t.done}}</div>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: 'app',
    data() {
      return {
        name: "Adam",
        tasks: [{ action: "Kup kwiaty", done: false },
                 { action: "Znajdź buty", done: false },
                 { action: "Odbierz bilety", done: true },
                 { action: "Zadzwoń do Janka", done: false }]
      }
    }
  }
</script>
```

Do wyświetlania listy zadań wykorzystam funkcję, która generuje podany fragment kodu HTML dla każdego obiektu z kolekcji. Przeanalizujmy uważnie poniższy fragment kodu; znalezienie go w gąszczu powyższego szablonu nie jest takie łatwe:

```
...
<div class="row" v-for="t in tasks" v-bind:key="t.action">
  <div class="col">{{t.action}}</div>
  <div class="col-2">{{t.done}}</div>
</div>
...
```

Zapisany pogrubioną czcionką fragment `v-for="t in tasks"` stanowi przykład **dyrektywy**. Wszystkie dyrektywy w Vue.js zaczynają się od ciągu `v-`. Dzięki nim elementy języka HTML otrzymują dodatkowe funkcje udostępniane przez Vue.js. Dyrektywa `v-for` duplikuje element, do którego zostanie zastosowana (podobnie jak wszystkie jego dzieci), dla każdego obiektu należącego do kolekcji. W ramach powtarzanego bloku uzyskujemy dostęp do specjalnej zmiennej, za pomocą której możemy tworzyć wiązania danych.

W ramach wyrażenia `t in tasks` możemy wyodrębnić dwie zmienne — `tasks`, która określa źródło danych, oraz `t`, pod którą pojedynczy element kolekcji będzie dostępny w ramach danego elementu HTML.

Dzięki zmienniej pomocniczej `t`, użytej w ramach dyrektywy `v-for`, możemy tworzyć wiązania danych do aktualnie przeglądanego obiektu, jak niżej:

```
...
<div class="row" v-for="t in tasks" v-bind:key="t.action">
  <div class="col">{{t.action}}</div>
  <div class="col-2">{{t.done}}</div>
</div>
...
```

W powyższym kodzie występują dwa wiązania w formie interpolacji tekstów. Ustawią one wartości właściwości `action` i `done` w wewnętrznych elementach `div`. Dyrektywa `v-bind` pomaga dyrektywie `v-for` śledzić kolejność elementów powiązanych z każdym obiektem.

Aby szablon mógł cokolwiek wyświetlić, dodałem właściwość `tasks` do obiektu zwracanego przez funkcję `data()`:

```
...
data() {
  return {
    name: "Adam",
    tasks: [{ action: "Kup kwiaty", done: false },
             { action: "Znajdź buty", done: false },
             { action: "Odbierz bilety", done: true },
             { action: "Zadzwoń do Janka", done: false }]
  }
}
...
```

Rezultatem powyższego kodu będzie zbiór bloków `div` wygenerowany przez Vue.js na podstawie tablicy `tasks`. Dzięki wiązaniom danych wyświetlimy wartości właściwości `action` i `done`.

Po zapisaniu zmian w pliku `App.vue` aplikacja ponownie zostanie odświeżona automatycznie, a naszym oczom powinien ukazać się efekt jak na rysunku 1.6. Dodatkowe zastosowane w tym przykładzie elementy `div` i powiązane klasy są używane w celu uzyskania układu siatki (ang. *grid layout*) i nie stanowią integralnej części Vue.js.

Zadanie	Zakończono?
Kup kwiaty	false
Znajdź buty	false
Odbierz bilety	true
Zadzwoń do Janka	false

Rysunek 1.6. Wyświetlanie listy zadań

Dodawanie przycisku wyboru (checkbox)

Nasza przykładowa aplikacja zaczyna wyglądać całkiem obiecująco, jednakże wskazywanie stanu zakończenia zadania za pomocą słów true i false z pewnością nie jest zbyt przyjazne dla użytkowników. Vue.js zawiera dyrektywy, za pomocą których można zmienić sposób wyświetlania tego rodzaju informacji. Jak widać na listingu 1.15, dodałem element input, który wyświetla wartość właściwości done każdego obiektu należącego do kolekcji tasks.

Listing 1.15. Dodawanie elementu formularza w pliku src/App.vue

```
<template>
  <div id="app">
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika {{name}}
    </h4>
    <div class="container-fluid p-4">
      <div class="row">
        <div class="col font-weight-bold">Zadanie</div>
        <div class="col-2 font-weight-bold">Zakończono?</div>
      </div>
      <div class="row" v-for="t in tasks" v-bind:key="t.action">
        <div class="col">{{t.action}}</div>
        <div class="col-2">
          <input type="checkbox" v-model="t.done" class="form-check-input" />
          {{t.done}}
        </div>
      </div>
    </div>
  </template>
<script>
  export default {
    name: 'app',
    data() {
      return {
        name: "Adam",
        tasks: [{ action: "Kup kwiaty", done: false },
                 { action: "Znajdź buty", done: false },
                 { action: "Odbierz bilety", done: true },
                 { action: "Zadzwoń do Janka", done: false }]
      }
    }
  }
</script>
```

Dyrektyna v-model wiąże wartość wyrażenia (w tym przypadku jest to po prostu właściwość z przyciskiem wyboru). Vue.js dostosowuje swoje działanie do rodzaju elementu, do którego zastosowano dyrektywę v-model. Jeśli pole input jest polem zaznaczanym, dyrektywa v-model ustawi jego stan na podstawie wartości logicznej wyrażenia. Po zapisaniu zmian w pliku App.vue powinieneś zobaczyć zmiany jak na rysunku 1.7.

Ten rodzaj dyrektywy pozwala na utworzenie wiązania dwukierunkowego, dzięki czemu zmiana stanu kontrolki (odznaczenie bądź zaznaczenie przycisku wyboru) spowoduje modyfikację wartości danych. Co za tym idzie, jeśli zmienisz stan zaznaczenia pola, zmieni się również tekst widoczny obok tego pola (rysunek 1.8).

Zadanie	Zakończono?
Kup kwiaty	<input type="checkbox"/> false
Znajdź buty	<input type="checkbox"/> false
Odbierz bilety	<input checked="" type="checkbox"/> true
Zadzwoń do Janka	<input type="checkbox"/> false

Rysunek 1.7. Zastosowanie dyrektywy w celu wyświetlenia przycisku wyboru

Zadanie	Zakończono?	Zakończono?
Kup kwiaty	<input type="checkbox"/> false	<input type="checkbox"/> false
Znajdź buty	<input type="checkbox"/> false	<input type="checkbox"/> false
Odbierz bilety	<input checked="" type="checkbox"/> true	<input type="checkbox"/> false
Zadzwoń do Janka	<input type="checkbox"/> false	<input type="checkbox"/> false

Rysunek 1.8. Efekt działania wiązania dwukierunkowego

W ten sposób przedstawiliśmy jeden z najważniejszych aspektów aplikacji w Vue.js, czyli możliwość natychmiastowego reagowania na zmiany zachodzące w danych. Gdy dochodzi do zmiany pewnej danej, wszystkie wiązania z nią powiązane zostają odświeżone. W tym przypadku zmiana stanu zaznaczenia kontrolki w interfejsie powoduje zmianę właściwości `done` obiektu z listy zadań. Ta zmiana z kolei prowadzi do odświeżenia tekstu powiązanego również z właściwością `done`. Jest to świetny dowód na to, jak mocny jest związek między danymi a ich wiązaniem.

Filtrowanie zakończonych zadań

Responsywny model danych oznacza, że w łatwy sposób możemy zarządzać sposobem prezentacji danych aplikacji. W przykładowej aplikacji najważniejsze znaczenie mają bez wątpienia zadania, które nie zostały ukończone. W listingu 1.16 została dodana funkcja, która pozwala odfiltrować ukończone zadania.

Listing 1.16. Filtrowanie zadań w pliku `src/App.vue`

```
<template>
<div id="app">
  <h4 class="bg-primary text-white text-center p-2">
    Lista zadań użytkownika {{name}}
  </h4>
```

```

<div class="container-fluid p-4">
  <div class="row">
    <div class="col font-weight-bold">Zadanie</div>
    <div class="col-2 font-weight-bold">Zakończono?</div>
  </div>
  <div class="row v-for="t in filteredTasks" v-bind:key="t.action">
    <div class="col">{{t.action}}</div>
    <div class="col-2 text-center">
      <input type="checkbox" v-model="t.done" class="form-check-input" />
    </div>
  </div>
  <div class="row bg-secondary py-2 mt-2 text-white">
    <div class="col text-center">
      <input type="checkbox" v-model="hideCompleted" class="form-check-input" />
      <label class="form-check-label font-weight-bold">
        Ukryj zakończone zadania
      </label>
    </div>
  </div>
</div>
</template>
<script>
  export default {
    name: 'app',
    data() {
      return {
        name: "Adam",
        tasks: [{ action: "Kup kwiaty", done: false },
                 { action: "Znajdź buty", done: false },
                 { action: "Odbierz bilety", done: true },
                 { action: "Zadzwoń do Janka", done: false }],
        hideCompleted: true
      }
    },
    computed: {
      filteredTasks() {
        return this.hideCompleted ?
          this.tasks.filter(t => !t.done) : this.tasks
      }
    }
  }
</script>

```

W elemencie script dodałem właściwość computed, w której zadeklarowałem funkcję filteredTasks. Właściwość computed służy do definiowania właściwości, które wykonują operację na danych aplikacji. W ten sposób Vue.js wydajniej wykrywa zmiany w danych aplikacji, co przydaje się zwłaszcza w bardziej skomplikowanych aplikacjach. Właściwość filteredTasks korzysta z nowo dodanej właściwości hideCompleted, aby dynamicznie określić, czy użytkownik chce oglądać wszystkie zadania, czy tylko te niezakończone.

Aby zarządzać właściwością hideCompleted, dodałem do elementu template przycisk wyboru, a także skorzystałem z dyrektywy v-model:

```

...
<input type="checkbox" v-model="hideCompleted" class="form-check-input" />
...

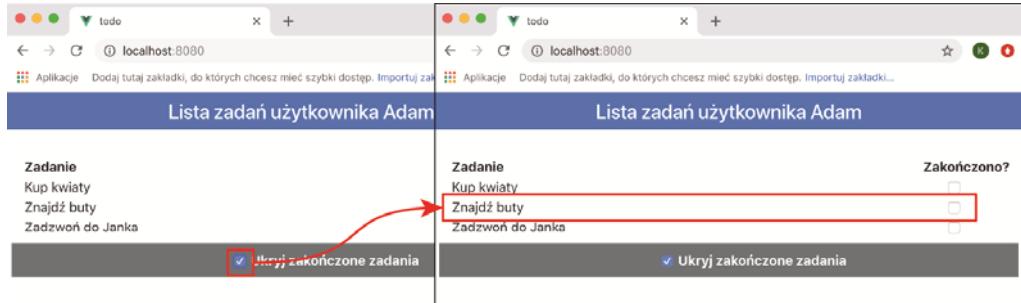
```

Aby upewnić się, że użytkownik widzi dane zgodnie ze swoim wyborem, zmieniłem treść dyrektywy v-for, aby odwoływała się do właściwości filteredTasks:

```
...
<div class="row" v-for="t in filteredTasks" v-bind:key="t.action">
...

```

Po przełączaniu przycisku wyboru wiązanie v-model aktualizuje stan właściwości hideCompleted, która w rezultacie zmienia wynik generowany przez właściwość filteredTasks i przedstawia użytkownikowi listę zadań, którą sobie zażyczył. Zapisz zmiany, poczekaj na odświeżenie przeglądarki i porównaj wynik z tym przedstawionym na rysunku 1.9.



Rysunek 1.9. Filtrowanie zadań

Tworzenie nowych zadań

Aplikacja listy zadań, która nie pozwala użytkownikowi na dodawanie nowych zadań, jest bezużyteczna. W listingu 1.17 dodaję kolejny składnik do elementu template. Dzięki niemu użytkownik może dodać nową pozycję do listy.

Listing 1.17. Tworzenie nowych zadań w pliku src/App.vue

```
<template>
<div id="app">
  <h4 class="bg-primary text-white text-center p-2">
    Lista zadań użytkownika {{name}}
  </h4>
  <div class="container-fluid p-4">
    <div class="row">
      <div class="col font-weight-bold">Zadanie</div>
      <div class="col-2 font-weight-bold">Czy wykonano?</div>
    </div>
    <div class="row" v-for="t in filteredTasks" v-bind:key="t.action">
      <div class="col">{{t.action}}</div>
      <div class="col-2 text-center">
        <input type="checkbox" v-model="t.done" class="form-check-input" />
      </div>
    </div>
    <div class="row py-2">
      <div class="col">
        <input v-model="newItemText" class="form-control" />
      </div>
      <div class="col-2">
        <button class="btn btn-primary" v-on:click="addNewTodo">Dodaj</button>
      </div>
    </div>
  </div>
</div>
```

```

</div>
<div class="row bg-secondary py-2 mt-2 text-white">
    <div class="col text-center">
        <input type="checkbox" v-model="hideCompleted" class="form-check-input" />
        <label class="form-check-label font-weight-bold">
            Ukryj zakończone zadania
        </label>
    </div>
</div>
</div>
</template>
<script>
    export default {
        name: 'app',
        data() {
            return {
                name: "Adam",
                tasks: [{ action: "Kup kwiaty", done: false },
                        { action: "Znajdź buty", done: false },
                        { action: "Odbierz bilety", done: true },
                        { action: "Zadzwoń do Janka", done: false }],
                hideCompleted: true,
                newItemText: "",
            }
        },
        computed: {
            filteredTasks() {
                return this.hideCompleted ?
                    this.tasks.filter(t => !t.done) : this.tasks
            }
        },
        methods: {
            addNewTodo() {
                this.tasks.push({
                    action: this.newItemText,
                    done: false
                });
                this.newItemText = "";
            }
        }
    }
</script>

```

Element input korzysta z dyrektywy v-model w celu utworzenia wiązania ze zmienną newItemText. W momencie edycji treści elementu Vue.js przypisze nową treść do tej zmiennej. Aby uruchomić proces tworzenia nowego elementu danych, dodałem dyrektywę v-on do elementu button, np.:

```

...
<button class="btn btn-primary" v-on:click="addNewTodo">
...

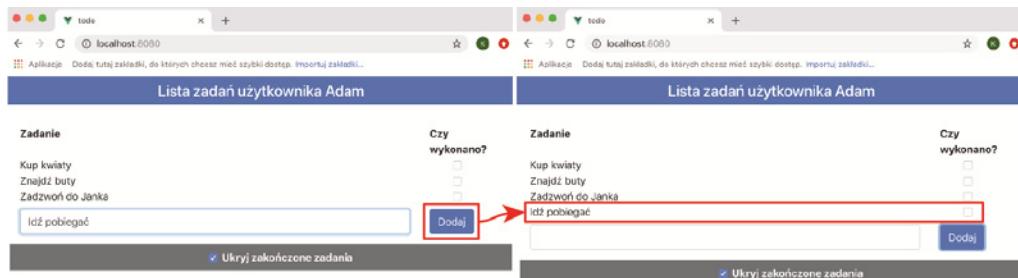
```

Dyrektyna v-on jest używana do obsługi zdarzeń na ogół mających podłożę w działaniach użytkownika. W tym przypadku dyrektywa v-on spowoduje wywołanie metody addNewTodo w momencie wyzwolenia zdarzenia click elementu button, czyli po prostu w chwili kliknięcia przycisku.

Zmiana w kodzie HTML pociąga za sobą konieczność modyfikacji kodu JavaScript w elemencie script. Dodałem właściwość methods, bo to właśnie w jej obrębie Vue.js poszukuje metod używanych w dyrektywach. W ramach właściwości methods dodałem metodę addNewTodo, która dodaje nowy obiekt do tablicy tasks,

przypisując wartość właściwości newItemText do atrybutu action. Po dodaniu obiektu do tablicy czyszczę wartość zmiennej newItemText. Wiązanie v-model połączone ze zmienną newItemText działa w obie strony, a więc wyczyszczenie zmiennej newItemText spowoduje usunięcie zawartości pola tekstowego input.

Po zapisaniu zmian i odświeżeniu przeglądarki zobaczyesz efekt jak na rysunku 1.10. Wprowadź opis zadania w elemencie input, a następnie kliknij przycisk *Dodaj*. Nowy element znajdzie się na liście.



Rysunek 1.10. Tworzenie nowego zadania

Trwałe przechowywanie danych

Przykładowa aplikacja nie zawiera mechanizmu do trwałego przechowywania danych. Oznacza to, że przy każdym odświeżeniu strony wszelkie zmiany wprowadzone na liście przez użytkownika zostaną stracone. Skorzystam z pamięci lokalnej (ang. *local storage*), która jest obsługiwana we wszystkich nowoczesnych przeglądarkach. Dzięki temu nie będę musiał konfigurować serwera, a ponadto udowodnię Ci, że w Vue.js można korzystać bezpośrednio z funkcji udostępnianych przez przeglądarki. Współpraca z serwerem rzecz jasna nas nie ominie, niemniej na razie chcę utrzymać prostą strukturę projektu. W listingu 1.18 dodałem do elementu script nowe instrukcje, dzięki którym będziemy w stanie zapisywać i pobierać dane (tym razem pominąłem element template, ponieważ nie uległ on żadnym zmianom). Więcej na temat tej zasady opowieściem w rozdziale 2.).

Listing 1.18. Stosowanie pamięci lokalnej w pliku src/App.vue

```
...
<script>
    export default {
        name: 'app',
        data() {
            return {
                name: "Adam",
                tasks: [],
                hideCompleted: true,
                newItemText: ""
            }
        },
        computed: {
            filteredTasks() {
                return this.hideCompleted ?
                    this.tasks.filter(t => !t.done) : this.tasks
            }
        },
        methods: {
            addNewTodo() {
                this.tasks.push({
                    action: this.newItemText,
                    done: false
                })
            }
        }
    }
</script>
```

```

        });
        localStorage.setItem("todos", JSON.stringify(this.tasks));
        this.newItemText = "";
    }
},
created() {
    let data = localStorage.getItem("todos");
    if (data != null) {
        this.tasks = JSON.parse(data);
    }
}

```

</script>
...

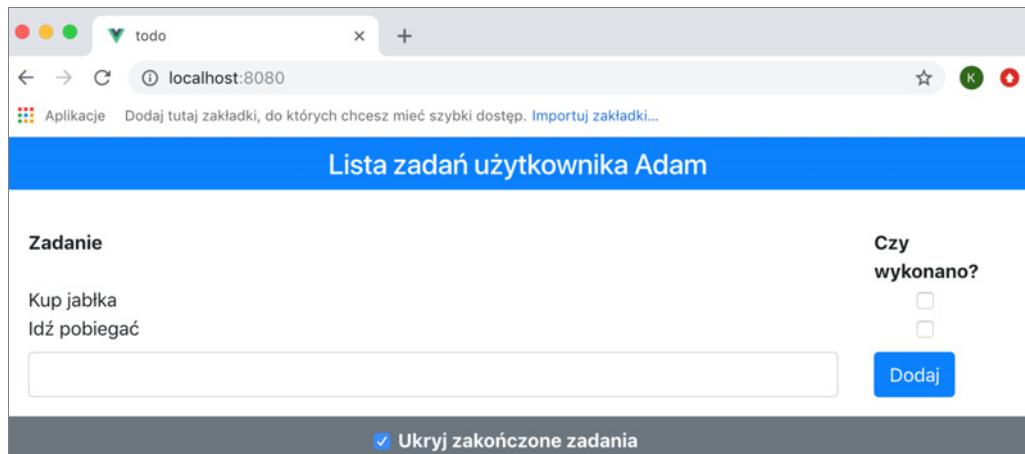
Pamięć lokalna jest dostępna za pomocą globalnego obiektu `localStorage`, który zawiera metody `getItem` i `setItem`. Obiekt ten jest dostarczony przez przeglądarkę.

- **Wskazówka** Obiekt `localStorage` jest dostępny również poza aplikacjami Vue.js. Jest to standardowy obiekt języka JavaScript, z którego można korzystać we wszystkich aplikacjach webowych niezależnie od stosowanego framework'a. Więcej informacji znajdziesz na stronie <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>.

Dodana do komponentu metoda `storeData` używa metody `setItem` do przechowywania listy zadań do zrobienia. Jest ona wywoływaną w momencie utworzenia nowego zadania do zrobienia lub przełączenia przycisku wyboru. Pamięć lokalna pozwala na przechowywanie wyłącznie tekstów, tak więc muszę zserializować obiekty do formatu JSON przed ich zapisaniem.

Metoda `created`, którą dodałem w listingu 1.18, jest wywoływana w momencie tworzenia komponentu. Dzięki niej mogę wczytać dane z pamięci lokalnej przed pokazaniem treści strony w przeglądarce. Ostatnia zmiana, która została uwzględniona w listingu 1.18, polega na usunięciu elementów zastępczych listy zadań, które nie są już nam w tym momencie potrzebne.

Po zapisaniu zmian przeglądarka odświeży zawartość, a aplikacja zacznie przechowywać elementy w trwałej pamięci lokalnej. W związku z tym będą one widoczne również po odświeżeniu strony, a nawet zamknięciu okna przeglądarki (rysunek 1.11).



Rysunek 1.11. Przechowywanie danych

Ostatnie szlify

Teraz dodam dwie ostatnie funkcje, które doprowadzą nas do finalnej postaci aplikacji. Najpierw pozwolimy użytkownikowi usunąć zakończone zadania z listy — jest to istotne, skoro przechowujemy je w sposób trwały. Dodamy także komunikat, który będzie wyświetlany, gdy na liście nie będzie żadnych zadań. Obie funkcje trafią do komponentu App (listing 1.19).

Listing 1.19. Wprowadzanie ostatnich zmian w pliku src/App.vue

```
<template>
  <div id="app">
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika {{name}}
    </h4>
    <div class="container-fluid p-4">
      <div class="row" v-if="filteredTasks.length == 0">
        <div class="col text-center">
          <b>Nie masz nic do zrobienia. Wspaniale!</b>
        </div>
      </div>
      <template v-else>
        <div class="row">
          <div class="col font-weight-bold">Zadanie</div>
          <div class="col-2 font-weight-bold">Czy wykonano?</div>
        </div>
        <div class="row" v-for="t in filteredTasks" v-bind:key="t.action">
          <div class="col">{{t.action}}</div>
          <div class="col-2 text-center">
            <input type="checkbox" v-model="t.done"
                  class="form-check-input" />
          </div>
        </div>
      </template>
    <div class="row py-2">
      <div class="col">
        <input v-model="newItemText" class="form-control" />
      </div>
      <div class="col-2">
        <button class="btn btn-primary"
               v-on:click="addNewTodo">Dodaj</button>
      </div>
    </div>
    <div class="row bg-secondary py-2 mt-2 text-white">
      <div class="col text-center">
        <input type="checkbox" v-model="hideCompleted"
              class="form-check-input" />
        <label class="form-check-label font-weight-bold">
          Ukryj zakończone zadania
        </label>
      </div>
      <div class="col text-center">
        <button class="btn btn-sm btn-warning"
               v-on:click="deleteCompleted">
          Usuń zakończone
        </button>
      </div>
    </div>
  </div>

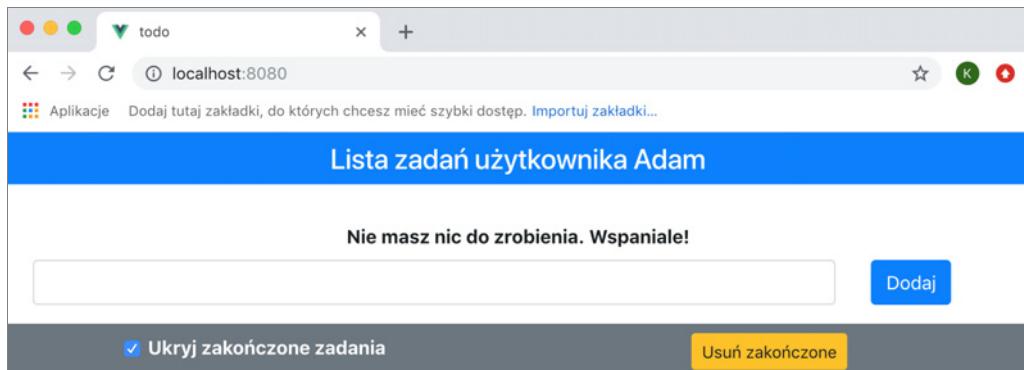
```

```

        </div>
    </div>
</div>
</template>
<script>
  export default {
    name: 'app',
    data() {
      return {
        name: "Adam",
        tasks: [],
        hideCompleted: true,
        newItemText: ""
      }
    },
    computed: {
      filteredTasks() {
        return this.hideCompleted ?
          this.tasks.filter(t => !t.done) : this.tasks
      }
    },
    methods: {
      addNewTodo() {
        this.tasks.push({
          action: this.newItemText,
          done: false
        });
        this.storeData();
        this.newItemText = "";
      },
      storeData() {
        localStorage.setItem("todos", JSON.stringify(this.tasks));
      },
      deleteCompleted() {
        this.tasks = this.tasks.filter(t => !t.done);
        this.storeData();
      }
    },
    created() {
      let data = localStorage.getItem("todos");
      if (data != null) {
        this.tasks = JSON.parse(data);
      }
    }
  }
</script>

```

Dyrektyny `v-if` i `v-else` pozwalają na warunkowe wyświetlanie elementów. Dzięki nim jesteśmy w stanie wyświetlić tekst tylko wtedy, gdy w tablicy nie ma żadnych elementów — w przeciwnym razie wyświetlamy zwykłą listę zadań. Dodajemy także element typu `button` i stosujemy dyrektywę `v-on`, aby obsłużyć zdarzenie `click`. Po kliknięciu przycisku odfiltruję zakończone zadania z listy, a następnie zapisuję te elementy, które pozostały. Po zapisaniu pliku `App.vue` aplikacja zostanie odświeżona. Przetestuj działanie aplikacji, zaznaczając pole *Ukryj zakończone zadania* i klikając przycisk *Usuń zakończone* (rysunek 1.12).



Rysunek 1.12. Ostatnie szlify

Podsumowanie

W tym rozdziale utworzyliśmy prostą, przykładową aplikację, dzięki której poznaliśmy podstawy programowania i budowania aplikacji w Vue.js. Choć przykład był prosty, poznaliśmy kilka ważnych dla Vue.js koncepcji.

Wiesz już, jak korzystać z podstawowych narzędzi Vue.js, aby stworzyć, spakować i uruchomić projekt w przeglądarce. Poznałeś także zasadę budowania aplikacji za pomocą komponentów, które łączą kod HTML i JavaScript. Kod JavaScript jest dzielony na sekcje, takie jak dane i metody. Nauczyłeś się także korzystać z dyrektyw i wiązań danych. Innymi słowy, odkrywasz zasady działania reaktywnego modelu danych, który znacznie ułatwia propagację zmian stanu w aplikacji.

Vue.js udostępnia wiele innych funkcji — co widać chociażby po rozmiarze tej książki — ale już teraz znasz podstawowe pojęcia, które rozwiniemy w kolejnych rozdziałach. Rozdział 2. jest poświęcony kontekstowi, w którym Vue.js funkcjonuje, a także strukturze i treści tej książki.

ROZDZIAŁ 2.



Zrozumieć Vue.js

Vue.js to otwarty, elastyczny i niezwykle funkcjonalny framework do tworzenia aplikacji działających po stronie klienta. Zasady rządzące tym frameworkiem zostały zaczerpnięte ze świata aplikacji serwerowych; zastosowanie ich wobec elementów języka HTML pozwoliło stworzyć podstawę do budowy bogatych aplikacji webowych (ang. *rich web applications*) w łatwy sposób. W tej książce wyjaśniam zasady działania Vue.js i omawiam różne funkcje przezeń dostarczane.

Ta książka a cykl wydawniczy Vue.js

Zespół tworzący Vue.js nieustajaco stara się, aby zmiany w API Vue.js były jak najmniejsze. Jest to mila odmiana w porównaniu z innymi frameworkami działającymi po stronie klienta, gdzie zmiany likwidujące kompatybilność wstępna są na porządku dziennym. Vue.js otrzymuje częste aktualizacje, ale z reguły nie wpływają one na działanie istniejących aplikacji. To świetna wiadomość pod kątem projektów, które będziesz tworzyć, a także przykładów w niniejszej książce.

Mimo wszystko zawsze istnieje możliwość, że przy której aktualizacji Vue.js przykłady zawarte w książce przestaną działać. Nieużyciwe byłoby prosić czytelników o zakup nowego wydania książki przy każdej takiej zmianie, zwłaszcza że większość funkcji Vue.js nie zmienia się nawet pomiędzy głównymi wydaniami. W związku z tym będę publikował odświeżone wersje oryginalnych kodów po każdym głównym wydaniu Vue.js na moim repozytorium na GitHubie pod adresem <https://github.com/Apress/pro-vue-js-2>.

Jest to swego rodzaju eksperyment, w związku z czym forma tych aktualizacji nie jest jeszcze do końca jasna, choćby dlatego, że nie wiem, co dokładnie będzie zawarte w kolejnych głównych wydanach Vue.js. Nie zmienia to faktu, że chcę wydłużyć czas, w którym ta książka będzie zawierała aktualne informacje, dzięki odświeżaniu kodów do niej dołączonych.

Nie mogę powiedzieć, co dokładnie będzie zawarte w tych aktualizacjach, jak szybko lub w jakiej formie się one ukażą. Mogę jedynie poprosić o przeglądanie od czasu do czasu strony repozytorium po wydaniu nowej wersji Vue.js. Jeśli masz pomysł na to, w jaki sposób usprawnić proces aktualizacji, wyślij e-mail na adam@adam-freeman.com i daj mi znać.

Czy warto korzystać z Vue.js?

Vue.js nie stanowi uniwersalnego rozwiązania wszystkich Twoich problemów, dlatego dobrze wiedzieć, do czego warto go używać, a kiedy rozsądnie jest rozejrzeć się za alternatywą. Vue.js daje możliwości dostępne nigdy wyłącznie dla twórców aplikacji natywnych, oferując je w pełni w przeglądarce. Generuje to sporo pracy dla przeglądarki, która musi uruchomić aplikację Vue.js, przetworzyć elementy dokumentu HTML, wykonać kod JavaScript, obsługiwać zdarzenia, a także zrealizować wszystkie zadania niezbędne do uruchomienia aplikacji Vue.js, m.in. te przedstawione w rozdziale 1.

Wszystko to zajmuje określona ilość czasu, a ta zależy od stopnia skomplikowania aplikacji Vue.js, samej przeglądarki, a także mocy obliczeniowej urządzenia. Z pewnością nie zauważysz opóźnienia, korzystając z najnowszych wersji przeglądarek na współczesnym komputerze, ale już w przypadku starszych przeglądarek działających na przestarzałych smartfonach proces inicjalizacji aplikacji w Vue.js może przebiegać dość wolno.

W związku z powyższym inicjalizacja aplikacji powinna odbywać się tak rzadko, jak to tylko możliwe. Większość treści aplikacji powinna być dostarczana do użytkownika w momencie, gdy jest potrzebna. To założenie trzeba uwzględnić w trakcie tworzenia aplikacji. Mówiąc ogólnie, istnieją dwa rodzaje aplikacji webowych: **wielostroncowe** (ang. *round-trip applications*) i **jednostroncowe** (ang. *single-page applications*, dalej zwane SPA).

Zasada działania aplikacji wielostroncowych

Przez długi czas aplikacje webowe działały według zasady modelu wielostroncowego. Zgodnie z tym modelem najpierw przeglądarka wysyła żądanie przesyłania początkowego dokumentu HTML z serwera. Działanie użytkownika — np. kliknięcie hiperłącza lub wysłanie formularza — generuje wysłanie kolejnego żądania, a w rezultacie — otrzymanie nowego dokumentu HTML. W tym modelu przeglądarka stanowi *de facto* tylko silnik renderujący treści HTML, a wszelka logika aplikacji i danych znajduje się po stronie serwera. Przeglądarka wykonuje szereg bezstanowych żądań HTTP, na które serwer odpowiada, generując dynamicznie kod HTML.

Obecnie cały czas tworzy się wiele aplikacji webowych tego rodzaju, zwłaszcza w przypadku projektów branżowych, ponieważ nakładają one niewielkie oczekiwania na przeglądarkę, a ponadto są najszerzej obsługiwane wśród klientów (przeglądarki). Niestety aplikacje wielostroncowe mają spore wady — użytkownik musi czekać na wyrenderowanie kolejnego dokumentu HTML, do przetwarzania żądań i zarządzania stanem aplikacji konieczna jest spora infrastruktura serwerowa, a jednocześnie przesyłanie kompletnych dokumentów HTML ze wszystkimi zasobami wymaga większej przepustowości łączna (za każdym razem jest przesyłane sporo identycznej treści, zwłaszcza kodu HTML). Vue.js nie jest dobrze dostosowane do tworzenia aplikacji wielostroncowych, ponieważ przeglądarka wykonuje proces inicjalizacji przy generowaniu każdego dokumentu HTML otrzymanego z serwera.

Zasada działania SPA

Aplikacje SPA działają nieco inaczej niż wielostroncowe. Najpierw, po przesyłaniu pierwszego żądania, również dochodzi do odesłania początkowego dokumentu HTML do przeglądarki, jednak po zakończeniu inicjalizacji wszelkie działania użytkownika prowadzą tylko do wysłania żądań Ajax. Odpowiedziami na te żądania czasami są fragmenty kodu HTML (zamiast całych dokumentów), a niekiedy nawet wyłącznie dane, które następnie są wstawiane/zamieniane w ramach istniejących elementów dokumentu HTML. Niezależnie od sytuacji nigdy nie dochodzi do zamiany całego dokumentu HTML. Oprócz tego należy pamiętać, że żądania Ajax są obsługiwane asynchronicznie (w tle), nawet jeśli sprowadza się to do wyświetlenia komunikatu „ładowanie danych”.

Aplikacje jednostroncowe idealnie pasują do Vue.js i innych frameworków działających po stronie klienta, takich jak Angular czy React. Wynika to z faktu, że inicjalizacja aplikacji musi odbyć się raz, a później aplikacja po prostu działa w przeglądarce, reagując na czynności użytkownika i pobierając niezbędne dane w tle.

Porównanie Vue.js z Reactem i Angularem

Vue.js ma dwóch głównych konkurentów: frameworki React i Angular. Mimo pewnych różnic wszystkie te trzy frameworki działają na podobnej zasadzie, robią to świetnie i mogą być używane do tworzenia bogatych i płynnych aplikacji działających po stronie klienta.

Jedną z kluczowych kwestii odróżniających Vue.js od pozostałych frameworków jest skupienie się wyłącznie na prezentacji treści HTML. Vue.js samo w sobie nie zawiera innych mechanizmów, takich jak asynchroniczna obsługa żądań HTTP czy trasowanie adresów URL. Nie jest to jednak istotne, ponieważ dodatkowe funkcje są obsługiwane za pomocą pakietów polecanych, a nawet rozwijanych przez zespół deweloperski Vue.js.

Prawdziwa różnica tkwi w doświadczeniu programistycznym niezbędnym do efektywnego używania frameworka. Angular do efektywnej pracy wymaga od programisty znajomości TypeScript, podczas gdy w Vue.js język ten jest wspierany, ale całkowicie opcjonalny. Ponadto Vue.js i React kierują się w stronę łączenia kodu HTML i JavaScriptu, co nie każdemu odpowiada.

Moja rada jest prosta: wybierz framework, który podoba Ci się najbardziej, ale jeśli po bliższym poznaniu dojdziesz do wniosku, że jednak Ci on nie odpowiada, nie wahaj się zmienić decyzji. Nie jest to może bardzo naukowe podejście, ale tak naprawdę żaden z wyborów nie jest zły, ponieważ większość zasadniczych koncepcji jest zaimplementowana podobnie we wszystkich trzech frameworkach.

Jeśli chcesz przeanalizować różnice między frameworkami, zajrzyj na stronę <https://vuejs.org/v2/guide/comparison.html>, która zawiera obszerne porównanie możliwości Vue.js z innymi frameworkami. Nie radzę zagłębiać się w bardo w drobiazgową analizę różnic, ponieważ wszystkie frameworki są dobre. W każdym z nich wykonasz duże i złożone projekty, a najlepszy framework to zawsze ten, który odpowiada Twojemu osobistemu stylowi programowania i w którym działasz najbardziej produktywnie.

Jak działa renderowanie po stronie serwera?

Renderowanie po stronie serwera (ang. *server-side rendering — SSR*) to kolejne pojęcie związane z aplikacjami SPA. Mechanizm ten ma na celu szybszą reakcję aplikacji w momencie, gdy użytkownik po raz pierwszy przechodzi pod dany adres URL. Dzięki SSR wyszukiwarki są w stanie lepiej indeksować treści aplikacji webowych.

SSR korzysta z technologii Node.js jako środowiska wykonawczego po stronie serwera w celu uruchomienia aplikacji webowej, aby wygenerować treść, która trafi do użytkownika. W tym momencie mamy do czynienia z aplikacją wielostopniową, w której każda interakcja powoduje wysłanie żądania HTTP do serwera. Serwer, dysponując uruchomioną aplikacją, odsyła kod HTML do przeglądarki użytkownika. Równolegle kod i treści wymagane do inicjalizacji aplikacji są pobierane w tle. Po zakończeniu inicjalizacji aplikacja staje się typową aplikacją SPA, a działania użytkownika są obsługiwane w kodzie działającym w przeglądarce.

Istnieje możliwość pobrania pakietów, które obsługują SSR w aplikacjach Vue.js, jednak nie zajmuje się nimi w tej książce. Renderowanie po stronie serwera jest procesem złożonym i ograniczonym do serwerów, które są w stanie wykonywać kod JavaScript, ponieważ muszą one uruchomić aplikację w imieniu użytkowników. Co ważne, w takich aplikacjach nie można korzystać z funkcji lub API, które są dostępne jedynie w przeglądarkach, ponieważ funkcje te nie będą dostępne na serwerze. Bezproblemowe przełączanie pomiędzy renderowaniem po stronie serwera i klienta może być trudne i mylące również dla użytkowników w przypadku wystąpienia problemów. Ograniczenia i problemy wynikające z SSR oznaczają, że nie jest to odpowiednie rozwiązanie dla większości projektów Vue.js i powinno być wdrażane niezwykle ostrożnie. Więcej na ten temat znajdziesz na stronie <https://vuejs.org/v2/guide/ssr.html>.

Złożoność aplikacji

Rodzaj aplikacji nie jest jedyną kwestią, którą trzeba rozważyć przed wyborem Vue.js jako technologii najlepiej dopasowanej do Twojego projektu. Ważna jest także jej złożoność — często słyszę od czytelników, którzy wybrali Vue.js, Angular lub React, że do ich projektu wystarczyłaby prostsza technologia. Frameworki takie jak Vue.js wymagają tego, by poświęcić nieco czasu na nauczenie się ich w odpowiednim stopniu (wystarczy spojrzeć na rozmiar tej książki). Taki wysiłek nie ma sensu, jeśli musisz jedynie zwalidować formularz lub wypełnić w sposób automatyczny pole typu *select*.

W świecie zdominowanym przez frameworki łatwo zapomnieć, że przeglądarki same w sobie dostarczają bogaty zbiór **API** (ang. *Application Programming Interface* — interfejs programowania aplikacji), który może być używany samodzielnie. Co więcej, z tych API korzysta również wewnętrznie Vue.js. Jeśli problem, który próbujesz rozwiązać jest prosty i niezależny od innych komponentów, możesz rozważyć używanie bezpośrednio API przeglądarki, zaczynając od API obiektowego modelu dokumentu (ang. **DOM** — *Document Object Model*). W dalszych rozdziałach przekonasz się, że czasami warto skorzystać bezpośrednio z API przeglądarki. Więcej informacji na ich temat znajdziesz na stronie <https://developer.mozilla.org/>.

Problemem API przeglądarek, zwłaszcza API DOM, jest nieco dziwny sposób użycia, a także fakt, że starsze przeglądarki implementują niektóre funkcje inaczej. Jeśli musisz korzystać ze starszych przeglądarek, dobrą alternatywą jest użycie biblioteki jQuery (<https://jquery.org/>), która pozwala na łatwą pracę z elementami HTML, a także wspiera obsługę zdarzeń, animacji czy asynchronicznych żądań HTTP.

Rozbudowane frameworki działające po stronie klienta, takie jak Vue.js, wymagają zbyt dużej inwestycji czasowej i zbyt wielu zasobów, aby stosować je w małych projektach. Ich użycie ma sens w projektach dużych, złożonych, gdzie występują rozbudowane procesy, różne rodzaje użytkowników, a także duże ilości danych do przetworzenia. W takich przypadkach praca bezpośrednio z API przeglądarek lub jQuery staje się trudna z uwagi na dużą ilość powstającego kodu, a także kwestię skalowania aplikacji. Funkcje udostępnione przez Vue.js pozwalają znacznie łatwiej tworzyć duże i złożone aplikacje bez konieczności zagłębiania się w nieczytelnym kodzie, co często staje się rzeczywistością w przypadku złożonych projektów tworzonych bez użycia jakiegokolwiek frameworka.

Co muszę wiedzieć?

Jeśli zdecydujesz się skorzystać z Vue.js w swoim projekcie, powinieneś znać podstawy tworzenia aplikacji webowych, rozumieć zasady działania języków HTML i CSS, a także mieć choćby podstawową wiedzę na temat języka JavaScript. Jeśli nie czujesz się do końca na siłach w tych kwestiach, to wiedz, że niezbędne zagadnienia są omówione w rozdziałach 3. i 4, jednak nie znajdziesz tam obszernego, kompletnego omówienia wszystkich elementów języka HTML czy właściwości języka CSS. Trudno omówić wszystko to w książce, która bądź co bądź traktuje o Vue.js. Więcej informacji na temat wspomnianych tutaj technologii znajdziesz na stronie <https://developer.mozilla.org/>.

Jak skonfigurować swoje środowisko programistyczne?

Jedynie narzędzia programistyczne, które są naprawdę niezbędne do rozpoczęcia tworzenia aplikacji w Vue.js, to te omówione w rozdziale 1. W kolejnych rozdziałach doinstalujemy jeszcze kilka dodatkowych pakietów. Jeśli jesteś w stanie zbudować aplikację z rozdziału 1., to jesteś gotowy do pracy z Vue.js przez resztę książki.

Jaki jest układ treści w tej książce?

Ta książka jest podzielona na trzy części, z których każda porusza pewien zbiór powiązanych ze sobą tematów.

Część I. Zaczynamy pracę z Vue.js

W pierwszej części tej książki przedstawiam informacje związane z początkiem programistycznej przygody z Vue.js. Znajdziesz w niej wiedzę na temat podstaw technologii Vue.js, a także technologii pokrewnych, takich jak HTML, CSS czy JavaScript. Dowiesz się, jak stworzyć swoją pierwszą aplikację, a także napiszesz nieco większą, bliższą rzeczywistości aplikację o nazwie *Sklep sportowy*.

Część II. Vue.js pod lupą

W drugiej części tej książki omówię najczęściej używane funkcje w Vue.js. Vue.js zawiera wiele wbudowanych mechanizmów, które omawiam szczegółowo. Oprócz tego przedstawiam metody dodawania własnego kodu i treści do projektu.

Część III. Zaawansowane funkcje Vue.js

W części trzeciej omówiono zaawansowane funkcje Vue.js, za pomocą których można tworzyć bardziej rozbudowane aplikacje. Przedstawiam także cykl życia aplikacji Vue.js, objaśniam współdzielony magazyn danych i demonstruję, jak wyświetlać fragmenty aplikacji na podstawie działań użytkownika. Pokażę Ci też, jak rozszerzyć wbudowane funkcje Vue.js i jak tworzyć testy jednostkowe w swoich projektach.

Czy znajdę tu dużo przykładów?

W tej książce znajdziesz mnóstwo przykładów. Najlepszą metodą nauki Vue.js jest uczenie się na przykładach, dlatego dołożyłem starań, aby w książce znalazło się ich bardzo wiele. Oprócz tego w książce zamieszczono też liczne zrzuty ekranu, dzięki czemu wiesz, co powinieneś otrzymać. Aby zmaksymalizować liczbę przykładów, staram się unikać powtarzania tego samego kodu raz po raz. W związku z tym, gdy wprowadzam nowy plik do projektu, przedstawiam jego pełną zawartość, jak w listingu 2.1. Dołączam nazwę pliku i katalog, w którym się znajduje, i pokazuję zmiany za pomocą pogrubionego kroju czcionki.

Listing 2.1. Pobieranie danych w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col"><error-display /></div>
    </div>
    <div class="row">
      <div class="col-8 m-3"><product-display /></div>
      <div class="col m-3"><product-editor /></div>
    </div>
  </div>
</template>
<script>
  import ProductDisplay from "./components/ProductDisplay";
  import ProductEditor from "./components/ProductEditor";
  import ErrorDisplay from "./components/ErrorDisplay";
  export default {
    name: 'App',
    components: { ProductDisplay, ProductEditor, ErrorDisplay },
    created() {
      this.$store.dispatch("getProductsAction");
    }
  }
</script>
```

Ten listing pochodzi z rozdziału 21. i przedstawia zawartość pliku *App.vue*, który znajduje się w katalogu *src*. Nie martw się tym, że nie wiesz, jakie jest przeznaczenie tego pliku ani co jest jego zawartością — na razie musisz jedynie wiedzieć, że ten listing przedstawia całą zawartość pliku, a zmiany, które musisz wprowadzić, aby podążyć za przykładem, są oznaczone pogrubieniem.

W pewnych sytuacjach niektóre pliki w aplikacji Vue.js mogą okazać się długie, a wprowadzane przez mnie zmiany są niewielkie. W takiej sytuacji zamiast pełnego pliku umieszczam wielokropek, aby oznaczyć, że mamy do czynienia jedynie z fragmentem pliku (listing 2.2).

Listing 2.2. Przygotowania do dynamicznego wyświetlania danych w pliku src/components/ProductEditor.vue

```
...
<script>
  let unwatcher;
  export default {
    data: function () {
      return {
        editing: false,
        product: {}
      }
    },
    methods: {
      save() {
        this.$store.dispatch("saveProductAction", this.product);
        this.product = {};
      },
      cancel() {
        this.$store.commit("selectProduct");
      },
      selectProduct(selectedProduct) {
        if (selectedProduct == null) {
          this.editing = false;
          this.product = {};
        } else {
          this.editing = true;
          this.product = {};
          Object.assign(this.product, selectedProduct);
        }
      }
    },
    created() {
      unwatcher = this.$store.watch(state =>
        state.selectedProduct, this.selectProduct);
      this.selectProduct(this.$store.state.selectedProduct);
    },
    beforeDestroy() {
      unwatcher();
    }
  }
</script>
...
```

Ten listing pochodzi z rozdziału 21. i przedstawia zmiany wprowadzone w znacznie większym pliku. Widząc taki listing, wiesz, że cała reszta pliku nie ulega zmianie.

Czasami zmiany muszą być wprowadzone w różnych fragmentach pliku, przez co trudno pokazać je na listingu częściowym. W takiej sytuacji omijam fragment istniejącego pliku (listing 2.3).

Listing 2.3. Dodawanie modułu w pliku *src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import PrefsModule from "./preferences";
import NavModule from "./navigation";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
  modules: {
    prefs: PrefsModule,
    nav: NavModule
  },
  state: {
    products: [],
    selectedProduct: null
  },
  //...inne funkcje magazynu danych zostały dla czytelności pominięte...
})
```

Zmiany wciąż są oznaczone pogrubieniem, a z kolei fragmenty pliku, które zostały ominięte w listingu, nie uległy żadnej zmianie.

Gdzie znajdę przykładowe kody?

Wszystkie przykładowe projekty znajdziesz w formie archiwum na serwerze FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.

Podsumowanie

W tym rozdziale omówilem rodzaje projektów, w których Vue.js jest dobrym wyborem, a także przedstawiłem alternatywy i bezpośrednią konkurencję. Wspomniałem także o tym, jakie treści są zawarte w dalszej części książki, i wskazałem, skąd pobrać przykładowe kody. W kolejnym rozdziale omawiam podstawowe funkcje języków HTML i CSS, z których to funkcji korzystam w tej książce w trakcie programowania w Vue.js.

ROZDZIAŁ 3.



Podstawy HTML i CSS

Programiści trafiają do świata aplikacji webowych na różne sposoby i nie zawsze znają podstawowe technologie, na których opiera się współczesna sieć. W tym rozdziale omawiam krótko podstawy języka HTML i biblioteki CSS Bootstrap, z których korzystam w przykładach w tej książce. W rozdziale 4. omawiam podstawy języka JavaScript, dzięki którym w pełni zrozumiesz resztę przykładów zawartych w niniejszej książce. Jeśli jesteś doświadczonym programistą, możesz pominąć początkowe rozdziały i przejść od razu do rozdziału 5., w którym wykorzystuję Vue.js do realizacji złożonych aplikacji.

-
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.
-

Przygotowania do rozdziału

W tym rozdziale potrzebny nam będzie prosty projekt Vue.js. Uzyskamy go, wywołując polecenie z listingu 3.1, które tworzy projekt o nazwie *htmlcssprimer*.

Listing 3.1. Tworzenie przykładowego projektu

```
vue create htmlcssprimer --default
```

Proces tworzenia nowego projektu może trochę potrwać, ponieważ konieczne jest pobranie i zainstalowanie wielu pakietów.

-
- **Uwaga** W trakcie pisania niniejszej książki pakiet @vue/cli był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratą dostępną pod adresem <https://github.com/Apress/pro-vue-js-2>.
-

Po utworzeniu projektu uruchom polecenia zawarte w listingu 3.2, dzięki którym zainstalujemy framework CSS Bootstrap wykorzystywany w tym rozdziale. Bootstrap posłuży nam do wprowadzenia zmian w treści aplikacji Vue.js.

Listing 3.2. Instalacja pakietu Bootstrap

```
cd htmlcssprimer
npm install bootstrap@4.0.0
```

Dodanie Bootstrapa do projektu wymaga dodania instrukcji przedstawionej w listingu 3.3 do pliku *src/main.js*. Zasadę działania instrukcji `import` omawiam w rozdziale 4., ale z pewnością już teraz widać, że nie jest to instrukcja skomplikowana.

Listing 3.3. Dodawanie instrukcji w pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
Vue.config.productionTip = false
import "bootstrap/dist/css/bootstrap.min.css";
new Vue({
  render: h => h(App)
}).$mount('#app')
```

Następnie zamieniam treść zastępczą w pliku *App.vue* na znaczniki `template` i `script` przedstawione w listingu 3.4 (znacznik `style` został całkowicie usunięty).

Listing 3.4. Zamiana treści w pliku src/App.vue

```
<template>
  <div>
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika Adam
    </h4>
    <div class="container-fluid p-4">
      <div class="row">
        <div class="col font-weight-bold">Zadanie</div>
        <div class="col-2 font-weight-bold">Czy wykonano?</div>
      </div>
      <div class="row" v-for="t in tasks" v-bind:key="t.action">
        <div class="col">{{t.action}}</div>
        <div class="col-2">{{t.done}}</div>
      </div>
    </div>
  </div>
</template>
<script>
export default {
  data: function () {
    return {
      tasks: [{ action: "Kup kwiaty", done: false },
               { action: "Znajdź buty", done: false },
               { action: "Odbierz bilety", done: true },
               { action: "Zadzwoń do Janka", done: false }]
    }
  }
}
</script>
```

Jest to uproszczona wersja aplikacji z rozdziału 1., zawierająca pewne dynamiczne mechanizmy, ale bez funkcji takich jak kończenie zadań czy dodawanie nowych elementów. Zapisz wszystkie zmiany i wykonaj polecenie z listingu 3.5 w katalogu *htmlcssprimer*, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 3.5. Uruchamianie narzędzi deweloperskich Vue.js

```
npm run serve
```

Początkowe przygotowanie projektu jak zawsze zajmie trochę czasu, ale wreszcie zobaczysz znajomy komunikat informujący, że aplikacja jest gotowa do działania. Otwórz nowe okno przeglądarki i przejdź pod adres <http://localhost:8080>, a zobaczysz efekt widoczny na rysunku 3.1.

Zadanie	Czy wykonano?
Kup kwiaty	false
Znajdź buty	false
Odbierz bilety	true
Zadzwoń do Janka	false

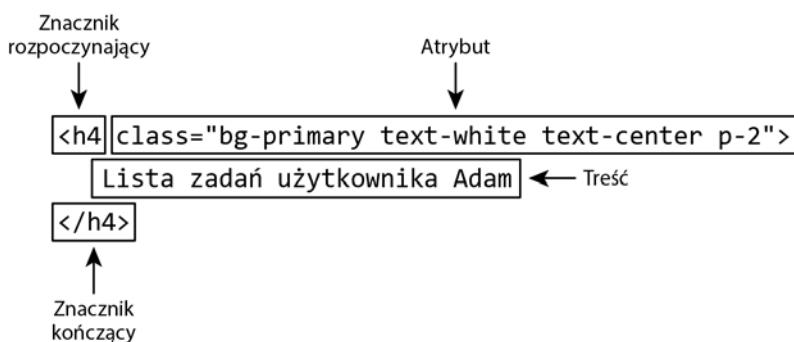
Rysunek 3.1. Uruchamianie przykładowej aplikacji

Jak działają elementy języka HTML?

Podstawowym pojęciem języka HTML jest **element**. Informuje on przeglądarkę o rodzaju treści, jaką należy w niej wyświetlić. Oto przykład elementu HTML:

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

Jak widać na rysunku 3.2, element składa się z kilku elementów — znacznika rozpoczynającego i kończącego, atrybutów i treści.



Rysunek 3.2. Budowa elementu HTML

Nazwa elementu (określana czasem jako **nazwa znacznika** lub po prostu **znacznik** — z ang. *tag*) to h4. Znacznik ten informuje przeglądarkę, że treść należy wyświetlić jako nagłówek. Istnieje szeroki zakres nagłówków, od h1 do h6, gdzie h1 oznacza najważniejsze informacje, h2 mniej istotne itd.

Definicja znacznika HTML wymaga umieszczenia nazwy znacznika w nawiasach ostrych (< i >). W ten sposób uzyskujemy **znacznik rozpoczęty**. **Znacznik kończący** wygląda tak samo, tyle że przed nazwą znacznika konieczne jest jeszcze dodanie znaku ukośnika (/).

Znacznik określa znaczenie elementu. Pełna lista standardowych znaczników jest określona w specyfikacji języka HTML. W tabeli 3.1 przedstawiam elementy, z których korzystam w listingu 3.4, a także najpopularniejsze elementy, które spotkasz w kolejnych rozdziałach. Pełną listę znaczników znajdziesz w specyfikacji języka HTML.

Tabela 3.1. Najpopularniejsze elementy języka HTML

Element	Opis
a	Oznacza hiperłącze (zwane też kotwicą — z ang. <i>anchor</i>), które po kliknięciu użytkownika powoduje przejście przeglądarki do nowego adresu URL lub innego miejsca w tym samym dokumencie.
button	Oznacza przycisk, używany często do wysyłania danych do serwera.
div	Element ogólnego przeznaczenia używany często do budowania struktury dokumentu pod kątem odpowiedniego wyświetlenia w przeglądarce.
h1 – h6	Nagłówki od pierwszego do szóstego stopnia.
input	Oznacza pole do pobrania pojedynczej informacji od użytkownika.
table	Oznacza tabelę, która wyświetla dane w formie wierszy i kolumn.
tbody	Oznacza treść, (dosłownie ciało) tabeli.
td	Oznacza komórkę tabeli w ramach wiersza.
template	Oznacza treść, która zostanie przetworzona za pomocą języka JavaScript. Komponenty Vue.js korzystają z szablonów do reprezentowania treści HTML. Element ten jest stosowany także w celu uniknięcia tworzenia nieprawidłowych dokumentów języka HTML.
th	Oznacza komórkę nagłówka w wierszu tabeli.
thead	Oznacza nagłówek tabeli.
tr	Oznacza wiersz tabeli.

Element a jego treść

Dowolna treść, która pojawi się pomiędzy znacznikiem otwierającym a zamkającym, stanowi jego zawartość. Element może zawierać tekst (np. *Lista zadań użytkownika Adam*) lub inne elementy języka HTML. Oto przykład z listingu 3.4, w którym element zawiera inne elementy języka HTML:

```
...
<div class="row">
  <div class="col font-weight-bold">Zadanie</div>
  <div class="col-2 font-weight-bold">Czy wykonano?</div>
</div>
...
```

Element zewnętrzny nosi nazwę **rodzica** (ang. *parent*), natomiast elementy znajdujące się w nim to jego **dzieci** (ang. *children*). Możliwość tworzenia hierarchii elementów stanowi istotną funkcję języka HTML, która pozwala na tworzenie skomplikowanych układów graficznych. Związek rodzic – dziecko można uogólnić, tj. pojedynczy element może być przodkiem wielu innych elementów, które z kolei mogą być jego potomkami.

Ograniczenia zawartości elementów języka HTML

Niektóre elementy wprowadzają pewne ograniczenia co do rodzajów elementów, które mogą być ich dziećmi. Elementy `div` mogą zawierać dowolny inny element, dzięki czemu łatwo jest go ostylować; z drugiej strony niektóre elementy mają bardzo specyficzne zastosowanie, stąd obecność ograniczeń. Element `tbody` może na przykład zawierać tylko jeden lub więcej elementów `tr`, z których każdy reprezentuje jeden wiersz tabeli.

- **Wskazówka** Nie musisz uczyć się wszystkich elementów języka HTML i związków pomiędzy nimi. Wszystkiego nauczysz się w trakcie pracy z przykładami w kolejnych rozdziałach. Ponadto większość edytorów wyświetla ostrzeżenie, gdy stworzysz nieprawidłową strukturę dokumentu HTML.

Elementy puste

Niektóre elementy nie mogą nic zawierać. Nazywamy je **elementami pustymi** (ang. *void*, także *self-closing*) i zapisujemy bez oddzielnego znacznika zamykającego, np.:

```
...
<input />
...
```

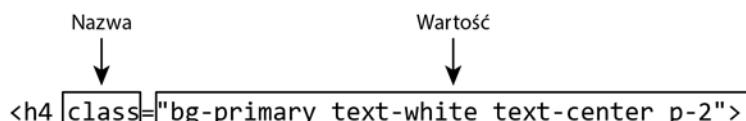
Element pusty składa się z pojedynczego znacznika, przy czym ukośnik jest dodawany przed zamykającym nawiastem ostrym (`>`). Przedstawiony powyżej element jest najczęściej używanym przykładem elementu pustego; jest on stosowany do pobierania danych od użytkownika z formularzy języka HTML. Wiele przykładów elementów HTML znajdziesz w kolejnych rozdziałach.

Jak działają atrybuty?

Dodatkowe informacje na temat znaczników można przekazać za pomocą **atributów**, które dodaje się do elementów języka HTML. Oto przykład zastosowania atrybutu do elementu `h4` (rysunek 3.2):

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

Atrybuty definiuje się w ramach znacznika rozpoczętym. Większość atrybutów składa się z nazwy i wartości przedzielonych znakiem równości (rysunek 3.3).



Rysunek 3.3. Nazwa i wartość atrybutu

Nazwa przykładowego atrybutu to `class`. Ten atrybut jest używany do grupowania powiązanych elementów, dzięki czemu można uzyskać ich spójny wygląd. Z tego powodu atrybut `class` jest używany również i w tym przykładzie, a jego wartość stanowi kilka klas, które wchodzą w skład pakietu Bootstrap, opisywanego przeze mnie w dalszej części rozdziału.

Stosowanie literałów tekstowych w atrybutach

Działanie Vue.js opiera się m.in. na atrybutach elementów języka HTML. W większości przypadków wartości atrybutów są ewaluowane jako wyrażenia języka JavaScript, tak jak w poniższym przykładzie (pochodzącym z listingu 3.4):

```
...
<div class="row" v-for="t in tasks" v-bind:key="t.action">
...

```

Wartości atrybutów zawierają krótkie fragmenty kodu JavaScript. Wartość atrybutu `v-for` jest wyrażeniem, które spowoduje przejście po elementach tablicy `tasks` i przypisanie ich do tymczasowej zmiennej `t`. Tablica `tasks` jest dostarczona w elemencie `script` w listingu 3.4. W pewnych sytuacjach konieczne będzie jednak dostarczenie konkretnej wartości (a nie odczytanie zmiennej określonej w elemencie `script`), co będzie wymagać uwzględnienia dodatkowych cudzysłów. Dzięki temu JavaScript potraktuje dany ciąg znaków jako literał, a nie zmienną, np.:

```
...
<h3 v-on:click="name = 'Clicked'>{{ name }}</h3>
...
```

Ten element pochodzi z rozdziału 14., a wartość atrybutu stanowi wyrażenie, które przypisuje ciąg znaków `Clicked` do właściwości `name`. Literał jest określony za pomocą cudzysłów pojedynczych (znak `'`), co sprawia, że Vue.js traktuje go dosłownie, nie próbując znaleźć zmiennej `Clicked` w ramach elementu `script`.

Stosowanie atrybutów bez wartości

Nie wszystkie atrybuty wymagają wartości. W niektórych przypadkach wystarczy pojawić się nazwy w celu osiągnięcia określonego zachowania. Oto przykład elementu, który zawiera taki atrybut (znajdziesz go w rozdziale 25.):

```
...
<transition enter-active-class="animated fadeIn" leave-active-class=" animated fadeOut"
mode="out-in" appear appear-active-class="animated zoomIn">
    <router-view />
</transition>
...
```

Ten element zawiera wiele atrybutów, ale warto zwrócić uwagę na atrybut `appear`, który nie ma wartości. Sama obecność nazwy atrybutu powoduje, że zostanie osiągnięty określony efekt.

Analiza przykładowego dokumentu HTML

Jeśli chcesz przeanalizować kod HTML strony internetowej, możesz skorzystać z opcji *Wyświetl źródło strony*, dostępnej w menu kontekstowym przeglądarki Google Chrome (kliknij prawym przyciskiem myszy treść strony, aby je wyświetlić). W innych przeglądarkach taka opcja jest dostępna w podobny sposób, choć z reguły nazywa się nieco inaczej. W przypadku naszej aplikacji takie postępowanie nie da oczekiwaneego rezultatu — nie zobaczysz kodu HTML listy zadań. Zamiast tego zobaczysz kod podobny do poniższego:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width,initial-scale=1.0">
        <link rel="icon" href="/favicon.ico">
```

```

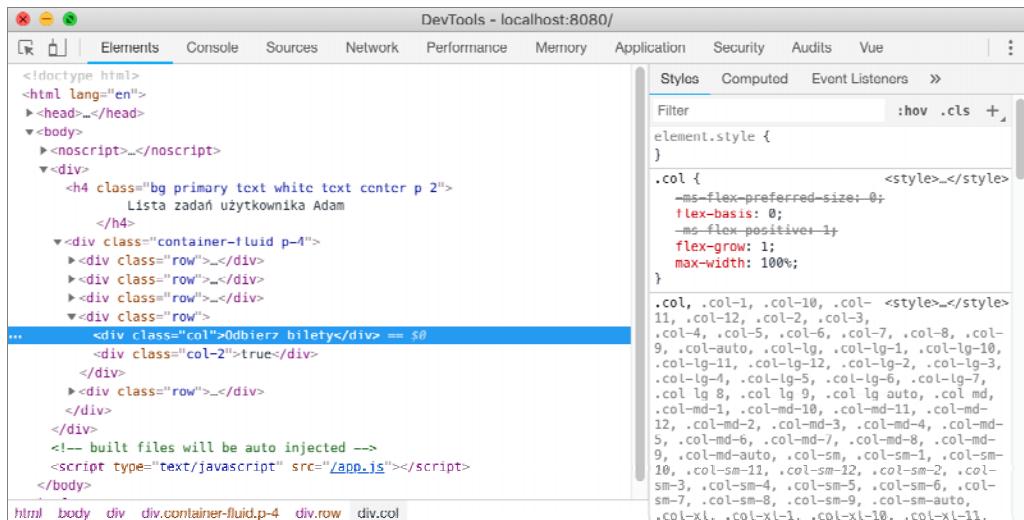
<title>htmlcssprimer</title>
<link as="script" href="/app.js" rel="preload">
</head>
<body>
<noscript>
<strong>
    We're sorry but htmlcssprimer doesn't work properly without
    JavaScript enabled. Please enable it to continue.
</strong>
</noscript>
<div id="app"></div>
<!-- built files will be auto injected -->
<script type="text/javascript" src="/app.js"></script></body>
</html>

```

Aby wyświetlić elementy HTML wygenerowane przez Vue.js, musisz uruchomić narzędzia deweloperskie przeglądarki. Po uruchomieniu aplikacji dochodzi do wywołania kodu JavaScript i w rezultacie otrzymujemy treść wyświetlzoną w oknie przeglądarki.

Większość przeglądarek zawiera narzędzia deweloperskie dostępne po wcisnięciu klawisza *F12* (dlatego często nazywa się je narzędziami F12), ale nierzadko można też skorzystać z opcji dostępnej z menu kontekstowego. W przypadku przeglądarki Google Chrome opcja ta nosi nazwę *Zbadaj*. Po jej wybraniu zostaną uruchomione narzędzia deweloperskie ze szczególnym uwzględnieniem elementu HTML, który kliknęłaś, uruchamiając menu kontekstowe.

Rysunek 3.4 przedstawia narzędzia deweloperskie tuż po kliknięciu w przeglądarce prawym przyciskiem myszy elementu tekstowego *Odbierz bilety* i wybraniu z menu kontekstowego opcji *Zbadaj*. Rysunek zawiera elementy HTML wygenerowane przez przeglądarkę, w tym ich treść, atrybuty i szczegóły stylów CSS, które ich dotyczą.



Rysunek 3.4. Badanie elementu HTML

Widok wyświetlony w narzędziach deweloperskich F12 zmienia się na bieżąco. Oznacza to, że zmiany w dokumencie HTML, które zajdą w wyniku działania aplikacji, zostaną wyświetlone w ramach tych narzędzi. W ten sposób możesz łatwo zrozumieć zachowanie aplikacji, zwłaszcza w sytuacji, gdy nie wszystko funkcjonuje po Twojej myśli.

Jak działa Bootstrap?

Elementy języka HTML informują przeglądarkę o rodzaju treści, które chcemy przedstawić, jednak nie określają sposobu ich wyświetlania. Do tego celu służy technologia CSS (ang. *Cascading Style Sheets* — kaskadowe arkusze stylów). CSS zawiera rozbudowany zestaw **właściwości** (ang. *properties*), które mogą być wykorzystywane do określenia każdego aspektu wyświetlania elementu. Za pomocą **selektorów** (ang. *selectors*) możemy z kolei określić, w jaki sposób właściwości mogą być stosowane do elementów.

Jednym z głównych problemów technologii CSS są różnice w interpretacji między przeglądarkami. Nie są one duże, jednak mogą powodować, że ten sam kod HTML na różnych urządzeniach może mieć różnych wygląd. Nie jest łatwo zdiagnozować, a także naprawić tego rodzaju problemy, niemniej w tym miejscu pomagają nam właśnie frameworki CSS — dzięki nim jesteśmy w stanie stylować znaczniki HTML w spójny sposób.

Najpopularniejszym frameworkm CSS jest z pewnością Bootstrap. Został on stworzony przez programistów Twittera, ale od momentu powstania ciepło przyjęta go społeczność open source. Bootstrap składa się ze zbioru klas CSS, które można stosować do wybranych elementów w celu spójnego stylowania. Oprócz tego framework ten zawiera liczne usprawnienia — nie korzystam jednak z nich w tej książce. Bootstrap często pojawia się w moich projektach — działa jednolicie w różnych przeglądarkach i jest prosty w użyciu. Korzystam ze stylów Bootstrapa, ponieważ dzięki nim jestem w stanie osiągnąć zadowalający efekt bez konieczności definiowania własnego kodu CSS. Bootstrap oferuje znacznie więcej, niż przedstawiam w niniejszej książce — dodatkowe informacje znajdziesz na stronie <http://getbootstrap.com/>.

Nie chcę zagłębiać się za bardzo w szczegóły związane z Bootstrapem, ponieważ nie jest on istotą niniejszej książki. Warto jednak wiedzieć, które fragmenty przykładów są związane stricte z Vue.js, a które z Bootstrapem.

Stosowanie podstawowych klas Bootstrapa

Style Bootstrapa stosujemy za pomocą atrybutu `class`, który pozwala na grupowanie powiązanych elementów. Atrybut `class` nie jest używany wyłącznie do stosowania stylów CSS; jest to jednak najczęstszy przypadek użycia. Atrybut ten ma decydujący wpływ na sposób działania frameworków takich jak Bootstrap. Oto przykładowy element HTML z atrybutem `class` (listing 3.4):

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

Atrybut `class` przypisuje do elementu `h4` cztery klasy oddzielone spacjami: `bg-primary`, `text-white`, `text-center` i `p-2`. Klasy te odpowiadają zbiorom stylów zdefiniowanych w Bootstrapie (tabela 3.2).

Tabela 3.2. Klasa elementu h4

Nazwa	Opis
<code>bg-primary</code>	Ta klasa stosuje kontekst stylu.
<code>text-white</code>	Ta klasa stosuje styl, który ustawia kolor tekstu elementu na biały.
<code>text-center</code>	Ta klasa stosuje styl, który pozycjonuje zawartość elementu na środku, w poziomie.
<code>p-2</code>	Ta klasa stosuje styl, który dodaje przestrzeń wokół elementu, co opisano w punkcie „ <i>Stosowanie marginesów i odstępów</i> ”.

Stosowanie klas kontekstowych

Jedną z zalet frameworków CSS takich jak Bootstrap jest możliwość uproszczenia procesu tworzenia spójnego szablonu (skórki) w aplikacji. Bootstrap korzysta z **kontekstów stylu** (ang. *style contexts*), które pozwalają na spójne stylowanie powiązanych elementów. Konteksty te zostały opisane w tabeli 3.3. Są one używane w nazwach klas, które aplikują style Bootstrapa do elementów.

Tabela 3.3. Konteksty stylu Bootstrapa

Nazwa	Opis
primary	Ten kontekst jest używany do oznaczenia głównego działania lub obszaru treści.
secondary	Ten kontekst jest używany do oznaczenia dodatkowych obszarów treści.
success	Ten kontekst jest używany do określenia pozytywnego zakończenia operacji.
info	Ten kontekst jest używany do przedstawienia dodatkowych informacji.
warning	Ten kontekst jest używany do przedstawienia ostrzeżeń.
danger	Ten kontekst jest używany do przedstawienia poważnych ostrzeżeń.
muted	Ten kontekst jest używany do zmniejszenia znaczenia treści.
dark	Ten kontekst jest używany do zwiększenia kontrastu przez zastosowanie ciemnego koloru.
white	Ten kontekst jest używany do zwiększenia kontrastu przez zastosowanie białego koloru.

Bootstrap dostarcza klasy, które pozwalają stosować konteksty stylu do różnych rodzajów elementów. Element h4, z którym zaczęłem pracę w tym podrozdziale, otrzymał klasę bg-primary, która ustawia kolor tła elementu, aby określić, że jest on związany z zasadniczym działaniem aplikacji. Inne klasy są stosowane wobec konkretnych elementów, np. btn-primary — klasa ta jest używana do ustawiania elementów button i a, dzięki czemu wyglądają one jak przyciski spójne z resztą głównego kontekstu. Niektóre z klas kontekstowych muszą być stosowane razem z innymi klasami, które ustawiają podstawowy styl elementu (np. klasa btn, która jest łączona z klasą btn-primary).

Stosowanie marginesów i odstępów

Bootstrap zawiera szereg klas pomocniczych, które pozwalają na ustawienie **odstępów** (ang. *padding*, zwanych też odstępami wewnętrznymi), określających obszar między granicą elementu a jego treścią, a także **marginesów** (ang. *margin*, zwanych też odstępami zewnętrznymi), które definiują obszar między granicą elementu a innymi elementami. Dzięki zastosowaniu tych klas w całej aplikacji zachowane są spójne przestrzenie między elementami.

Nazwy klas są konstruowane według ściśle określonego wzoru. Oto element h4 z listingu 3.4:

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

Klasy, które wprowadzają marginesy i odstępy, funkcjonują zgodnie z precyzyjnym schematem: na początku pojawia się litera m (dla marginesów) i p (dla odstępów). Następnie, opcjonalnie, pojawia się litera określająca konkretne krawędzie (t dla górnej krawędzi, b dla dolnej, l dla lewej i r dla prawej). Po znaku minus następuje liczba, która określa przestrzeń do zastosowania (0 — brak przestrzeni; kolejne wartości 1, 2, 3, 4 i 5 oznaczają coraz większe ilości przestrzeni). Jeśli w nazwie klasy nie występuje litera, która określa krawędzie, dany odstęp lub margines zostanie zastosowany do wszystkich krawędzi. W tym przypadku klasa p-2 oznacza, że element h4 otrzymuje odstęp poziomu drugiego dla wszystkich swoich krawędzi.

Stosowanie Bootstrapa do tworzenia siatki

Bootstrap dostarcza klasę stylów, które mogą być używane do tworzenia różnorodnych układów opartych na siatce (ang. *grid layout*), od jedno- do dwunastokolumnowych, także ze wsparciem w zakresie tworzenia układów responsywnych, w których siatka zmienia się w zależności od szerokości ekranu. Z układu siatki korzystam w wielu przykładach, w tym w komponencie przedstawionym w listingu 3.4, który prezentuje elementy listy zadań do zrobienia:

```
...
<div class="container-fluid p-4">
  <div class="row">
    <div class="col font-weight-bold">Task</div>
    <div class="col-2 font-weight-bold">Done</div>
  </div>
<div class="row" v-for="t in tasks" v-bind:key="t.action">
  <div class="col">{{t.action}}</div>
  <div class="col-2">{{t.done}}</div>
</div>
</div>
...
```

Układ siatki Bootstrapa jest prosty w użyciu. Główny element `div` otrzymuje klasę `container` (lub `container-fluid`, jeśli chcesz zająć całą dostępną przestrzeń). Kolumnę definiuje się przez zastosowanie klasy `row` do elementu `div`, dzięki czemu układ siatki zostanie zastosowany wobec treści zawartej w elemencie `div`.

Każdy wiersz zawiera 12 kolumn, a Twoim zadaniem jest określenie, ile kolumn zajmie każdy element-dziecko. W tym celu przypisujesz mu klasę o nazwie `col-`, gdzie po znaku minus musisz podać liczbę kolumn. Klasa `col-1` oznacza, że element zajmie jedną kolumnę, `col-2` — dwie, a `col-12` — że element zajmie cały wiersz. Pominiecie liczby kolumn (przypisanie klasy `col`) oznacza, że Bootstrap zarezerwuje dla elementu wszystkie pozostałe w danym wierszu kolumny.

Stosowanie Bootstrapa do stylowania tabel

Bootstrap wspiera stylowanie elementów `table` i ich zawartości — korzystam z tej możliwości w dalszych rozdziałach. Tabela 3.4 zawiera kluczowe klasy Bootstrapa używane w pracy z tabelami.

Tabela 3.4. Klasy Bootstrapa dla tabel

Nazwa	Opis
<code>table</code>	Stosuje ogólne style do elementu <code>table</code> i jego wierszy.
<code>table-striped</code>	Stosuje alternatywny styl wierszy w treści tabeli.
<code>table-bordered</code>	Stosuje obramowanie do wszystkich wierszy i kolumn.
<code>table-sm</code>	Likwiduje przestrzeń w tabeli, aby stworzyć bardziej zwarte układy.

Wszystkie klasy są stosowane bezpośrednio do elementu `table` (listing 3.6), w którym układ siatki zastąpiłem tabelą.

Listing 3.6. Stosowanie układu tabelarnego w pliku `src/App.vue`

```
<template>
  <div>
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika Adam
    </h4>
```

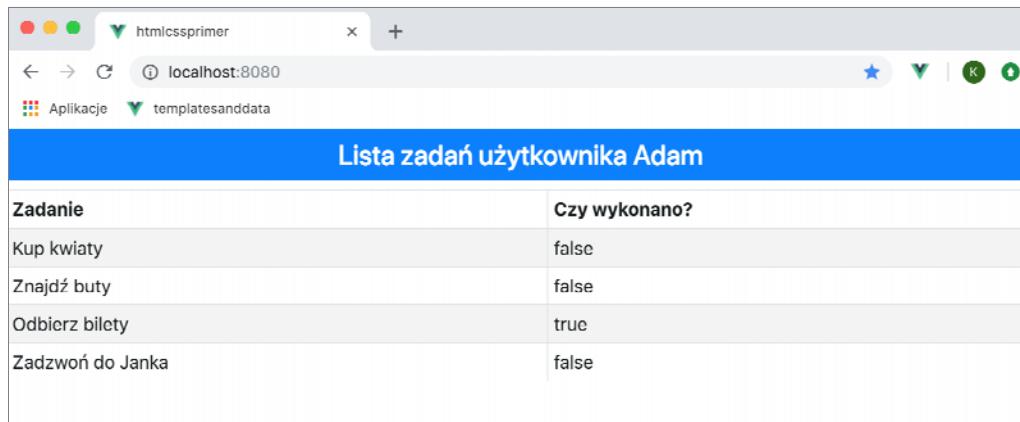
```

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr><th>Zadanie</th><th>Czy wykonano?</th></tr>
    </thead>
    <tbody>
        <tr v-for="t in tasks" v-bind:key="t.action">
            <td>{{t.action}}</td>
            <td>{{t.done}}</td>
        </tr>
    </tbody>
</table>
</div>
</template>
<script>
    export default {
        data: function () {
            return {
                tasks: [{ action: "Kup kwiaty", done: false },
                        { action: "Znajdź buty", done: false },
                        { action: "Odbierz bilety", done: true },
                        { action: "Zadzwoń do Janka", done: false }]
            }
        }
    }
</script>

```

-
- **Wskazówka** Zwróć uwagę, że w definicji tabeli w listingu 3.6 zastosowałem element `thead`. Przeglądarki automatycznie dodają elementy `tr`, będące bezpośrednimi potomkami elementu `table`, do elementu `tbody`, jeśli takowy nie został użyty. Przy korzystaniu z Bootstrapa takie zachowanie może spowodować dziwne efekty, dlatego definiując tabelę, zawsze dobrze jest utworzyć od razu kompletny zestaw znaczników.
-

Rysunek 3.5 przedstawia efekt zastosowania tabeli zamiast siatki do wyświetlania listy zadań.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080'. The main content area features a blue header bar with the text 'Lista zadań użytkownika Adam'. Below it is a table with four rows. The table has two columns: 'Zadanie' and 'Czy wykonano?'. The data rows are:

Zadanie	Czy wykonano?
Kup kwiaty	false
Znajdź buty	false
Odbierz bilety	true
Zadzwoń do Janka	false

Rysunek 3.5. Stylowanie tabeli HTML

Stosowanie Bootstrapa do stylowania formularzy

Bootstrap zawiera także style dla formularzy, dzięki czemu mogą one być stylowane spójnie z resztą aplikacji. W listingu 3.7 dodałem elementy formularza do przykładowej aplikacji.

Listing 3.7. Dodawanie elementów formularza do pliku src/App.vue

```
<template>
  <div>
    <h4 class="bg-primary text-white text-center p-2">
      Lista zadań użytkownika Adam
    </h4>
    <table class="table table-striped table-bordered table-sm">
      <thead>
        <tr><th>Zadanie</th><th>Czy wykonano?</th></tr>
      </thead>
      <tbody>
        <tr v-for="t in tasks" v-bind:key="t.action">
          <td>{{t.action}}</td>
          <td>{{t.done}}</td>
        </tr>
      </tbody>
    </table>
    <div class="form-group m-2">
      <label>Nowy element:</label>
      <input v-model="newItemText" class="form-control" />
    </div>
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="addNewTodo">
        Dodaj
      </button>
    </div>
  </div>
</template>
<script>
  export default {
    data: function () {
      return {
        tasks: [{ action: "Kup kwiaty", done: false },
                 { action: "Znajdź buty", done: false },
                 { action: "Odbierz bilety", done: true },
                 { action: "Zadzwoń do Janka", done: false }],
        newItemText: ""
      }
    },
    methods: {
      addNewTodo() {
        this.tasks.push({
          action: this.newItemText,
          done: false
        });
        this.newItemText = "";
      }
    }
  }
</script>
```

Podstawowe style formularzy można uzyskać za pomocą klasy `form-group`, zastosowanej do elementu `div`, zawierającego elementy `input` i `label`. Element `input` otrzymuje także klasę `form-control`. Bootstrap styluje elementy, dzięki czemu element `label` jest pokazany nad elementem `input`, a z kolei element `input` otrzymuje 100% przestrzeni poziomej (rysunek 3.6).

Zadanie	Czy wykonano?
Kup kwiaty	false
Znajdź buty	false
Odbierz bilety	true
Zadzwoń do Janka	false

Nowy element:

Dodaj

Rysunek 3.6. Stylowanie elementów formularza

Podsumowanie

W tym rozdziale omówiłem pokrótkie podstawy języka HTML i frameworka CSS Bootstrap. Znajomość technologii HTML i CSS jest konieczna, aby efektywnie tworzyć aplikacje webowe. Moim zdaniem najlepiej jest zdobyć te umiejętności w praktyce. Opisy przykładów w tym rozdziale pozwolą Ci kontynuować lekturę kolejnych rozdziałów i analizować przykłady w nich zawarte. W następnym rozdziale omówię podstawy języka JavaScript w zakresie wykorzystywanym w tej książce.

ROZDZIAŁ 4.



Elementarz JavaScriptu

W tym rozdziale omówię najważniejsze, z perspektywy Vue.js, funkcje języka JavaScript. Nie jest to z pewnością kompletny opis tego języka. Koncentruję się jedynie na najistotniejszych aspektach, które są niezbędne do zrozumienia omawianych przykładów. Tabela 4.1 przedstawia streszczenie tego rozdziału.

Tabela 4.1. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Podaj instrukcje, które zostaną wykonane w przeglądarce.	Zastosuj wyrażenia języka JavaScript.	4.5
Wstrzymaj się z wykonaniem instrukcji do momentu, gdy będą potrzebne.	Zastosuj funkcje języka JavaScript.	4.6 – 4.8, 4.11 – 4.12
Zdefiniuj funkcje o zmiennej liczbie parametrów.	Zastosuj parametry domyślne i parametry reszty (ang. <i>default/rest parameters</i>).	4.9 – 4.10
Zadeklaruj funkcje zwięzłe.	Skorzystaj z funkcji strzałkowych (ang. <i>fat arrow functions</i>).	4.14
Zdefiniuj zmienne i stałe.	Skorzystaj ze słów kluczowych <code>let</code> i <code>const</code> .	4.15 – 4.16
Zastosuj prymitywne typy języka JavaScript.	Użyj słów kluczowych <code>string</code> , <code>number</code> i <code>boolean</code> .	4.17 – 4.18, 4.20
Zdefiniuj łańcuchy znaków, które zawierają inne wartości.	Zastosuj łańcuchy szablonowe.	4.19
Wykonaj instrukcje warunkowo.	Zastosuj słowa kluczowe <code>if</code> , <code>else</code> i <code>switch</code> .	4.21
Porównaj według wartości i tożsamości.	Skorzystaj z operatorów równości i identyczności.	4.22 – 4.23
Skonwertuj typy.	Skorzystaj ze słów kluczowych związanych z konwersją typów.	4.24 – 4.26
Zgrupuj powiązane elementy.	Zdefiniuj tablicę.	4.27 – 4.28
Odczytaj lub zmień wartość w tablicy.	Skorzystaj z indeksu.	4.29 – 4.30
Wylicz zawartość tablicy.	Skorzystaj z metody <code>forEach</code> lub pętli <code>for</code> .	4.31

Tabela 4.1. Podsumowanie rozdziału — ciąg dalszy

Problem	Rozwiążanie	Listing
Rozwiń zawartość tablicy.	Skorzystaj z operatora rozwinięcia (ang. <i>spread operator</i>).	4.32 – 4.33
Przetwórz zawartość tablicy.	Skorzystaj z wbudowanej metody.	4.34
Zgrupuj powiązane informacje w pojedynczą jednostkę danych.	Zdefiniuj obiekt.	4.35 – 4.37
Zdefiniuj operację, która może być wykonywana na wartościach należących do obiektu.	Zdefiniuj metodę.	4.38 – 4.39
Skopiuj właściwości i wartość z jednego do drugiego obiektu.	Skorzystaj z metody <code>Object.assign</code> .	4.40
Zgrupuj powiązane funkcje.	Utwórz moduł języka JavaScript.	4.41 – 4.49
Obsłuż operację asynchroniczną.	Zdefiniuj obietnicę (<code>Promise</code>), a także skorzystaj z operatorów <code>async</code> i <code>await</code> .	4.50 – 4.54

Przygotowania do rozdziału

W tym rozdziale muszę skorzystać z prostego projektu Vue.js. W tym celu wykonam polecenie z listingu 4.1 — utworzę projekt o nazwie *jsprimer*.

Listing 4.1. Tworzenie przykładowego projektu

```
vue create jsprimer --default
```

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.

Projekt zostanie utworzony, pakiety zostaną pobrane, a narzędzia deweloperskie — pobrane i zainstalowane. Cały proces może zająć trochę czasu.

- **Uwaga** W trakcie pisania niniejszej książki pakiet `@vue/cli` był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratum dostępną pod adresem <https://github.com/Apress/pro-vue-js-2>.

Po utworzeniu projektu otwórz wybrany edytor tekstowy i zamień zawartość pliku `src/main.js` na pojedynczą instrukcję z listingu 4.2. W tym rozdziale koncentruję się na języku JavaScript, a nie Vue.js, dlatego kod inicjalizacji aplikacji Vue.js nie będzie nam potrzebny.

Listing 4.2. Zamiana zawartości pliku `src/main.js`

```
console.log("Cześć!");
```

Nasz projekt wymaga wprowadzenia zmiany w konfiguracji, aby przesłonić domyślne ustawienia w projekcie. Dodaj instrukcje przedstawione w listingu 4.3 do pliku `package.json`, który przechowuje niezbędne ustawienia (szczegółowo opisuję je w rozdziale 10.).

Listing 4.3. Zmiana konfiguracji projektu w pliku jsprimer/package.json

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/essential",
    "eslint:recommended"
  ],
  "rules": {
    "no-console": "off",
    "no-declare": "off",
    "no-unused-vars": "off"
  },
  "parserOptions": {
    "parser": "babel-eslint"
  }
},
"postcss": {
  "plugins": {
    "autoprefixer": {}
  }
},
...
}
```

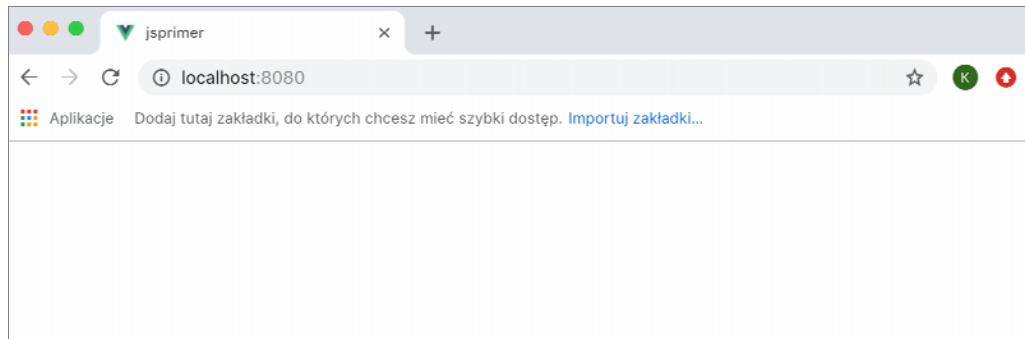
W ten sposób wyłączamy ostrzeżenia powstające w wyniku użycia funkcji języka JavaScript przedstawionych w tym rozdziale, generowane przez linter tego języka. Szczegółowo opisuję je w rozdziale 10.

Zapisz zmiany w plikach *main.js* i *package.json*, a następnie otwórz wiersz poleceń (terminal) i wykonaj polecenia z listingu 4.4, aby uruchomić narzędzia deweloperskie Vue.js. Mimo że w tym rozdziale nie korzystamy z funkcji Vue.js, zamierzam odwołać się do tych samych narzędzi, dzięki czemu łatwiej będzie nam wykonywać kod JavaScript w przeglądarce.

Listing 4.4. Uruchamianie narzędzi deweloperskich Vue.js

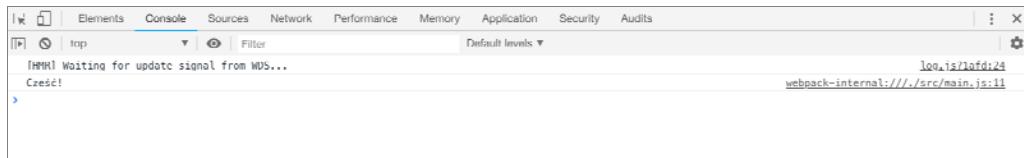
```
cd jsprimer
npm run serve
```

Po inicjalizacji projektu zostanie wyświetlony komunikat o zakończeniu przygotowania aplikacji. Otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, a zobaczysz puste okno, jak na rysunku 4.1.



Rysunek 4.1. Uruchomienie przykładowej aplikacji

Po uruchomieniu narzędzi deweloperskich (za pomocą klawisza *F12*) i zapoznaniu się z zakładką *Console* zobaczyś, że kod z listingu 4.2 spowodował powstanie prostego wyniku (rysunek 4.2).



Rysunek 4.2. Komunikat w konsoli przeglądarki

Wszystkie przykłady w tym rozdziale generują tekst na standardowym wyjściu (w tym przypadku — w konsoli przeglądarki). W związku z tym, zamiast zrzutów ekranu, przedstawiam je w postaci zwykłego tekstu:

```
[HMR] Waiting for update signal from WDS...
Cześć!
```

Pierwszy wiersz komunikatu jest generowany przez narzędzie deweloperskie, które automatycznie odświeża okno w momencie wykrycia zmiany w katalogu *src*. W związku z tym nie będę uwzględniał go w kolejnych przykładach.

Stosowanie instrukcji

Podstawowym elementem kodu JavaScript jest **instrukcja** (ang. *statement*). Każda instrukcja reprezentuje pojedyncze polecenie i z reguły kończy się znakiem średnika (;). Średnik nie jest wymagany, jednak dzięki niemu kod jest znacznie łatwiejszy do zrozumienia, a ponadto możemy umieszczać wiele instrukcji w pojedynczym wierszu tekstu. W listingu 4.5 dodałem kilka instrukcji do pliku *main.js*.

Listing 4.5. Dodawanie instrukcji języka JavaScript do pliku *src/main.js*

```
console.log("Cześć!");
console.log("Jabłka.");
console.log("To jest instrukcja.");
console.log("To też jest instrukcja.");
```

Przeglądarka wykonuje polecenia w kolejności. W tym przypadku każda z instrukcji generuje komunikat w konsoli. Oto efekt wywołania powyższego kodu:

```
Cześć!
Jabłka.
To jest instrukcja.
To też jest instrukcja.
```

Tworzenie i używanie funkcji

Gdy przeglądarka otrzymuje zestaw instrukcji JavaScript, jest on wywoływany w kolejności ich występowania — od pierwszej do ostatniej, tak jak w poprzednim przykładzie. Instrukcje w pliku *main.js*, generujące komunikaty w konsoli, są wykonywane jedna po drugiej. Jeśli jednak chcesz uniknąć natychmiastowego wykonania fragmentu kodu, możesz napisać go w ramach funkcji. Dzięki temu przeglądarka nie wykona kodu dopóty, dopóki nie napotka wywołania funkcji, jak w listingu 4.6.

Listing 4.6. Definicja funkcji JavaScript w pliku *src/main.js*

```
const myFunc = function () {
    console.log("Ta instrukcja jest wewnątrz funkcji.");
};
console.log("Ta instrukcja jest na zewnątrz funkcji.");
myFunc();
```

Definicja funkcji jest prosta — po słowie `const` umieszczamy nazwę funkcji, następnie znak równości (`=`) i słowo kluczowe `function`. Dalej umieszczamy nawiasy okrągłe, po których następuje para nawiasów klamrowych. Pomiędzy nawiasami klamrowymi możesz umieścić instrukcje, które mają zostać wykonane w ramach funkcji.

W tym listingu tworzymy funkcję o nazwie `myFunc`, która zawiera jedną instrukcję (ponownie wysyłamy tekst do konsoli przeglądarki). Instrukcja ta nie zostanie wywołana aż do momentu, gdy gdzieś nie nastąpi wywołanie funkcji, np.:

```
...
myFunc();
...
```

Po zapisaniu pliku *main.js* zaktualizowany plik zostanie wysłany do przeglądarki, a w rezultacie otrzymamy następujący efekt:

```
Ta instrukcja jest na zewnątrz funkcji.  
Ta instrukcja jest wewnątrz funkcji.
```

Jak widać, instrukcja wewnątrz funkcji nie jest wykonywana od razu. Należy jednak zwrócić uwagę, że w tym konkretnym przypadku zadeklarowanie funkcji nie dało nam istotnej korzyści — wywołanie funkcji znalazło się przecież zaraz za jej deklaracją. Funkcje przydają się znacznie bardziej, gdy ich wywołania mają miejsce w wyniku reakcji na jakieś zdarzenie, takie jak działanie użytkownika.

Funkcje można także definiować bezpośrednio, nie przypisując ich jawnie do zmiennej (listing 4.7).

Listing 4.7. Definicja funkcji w pliku *src/main.js*

```
function myFunc() {
    console.log("Ta instrukcja jest wewnątrz funkcji.");
}
console.log("Ta instrukcja jest na zewnątrz funkcji.");
myFunc();
```

Kod zadziała identycznie jak ten z listingu 4.6, ale taka konstrukcja jest bardziej znana większości programistów i dlatego też (jak również z uwagi na zasady tworzenia aplikacji Vue.js) w ten sposób definiuję funkcje w niniejszej książce.

Stosowanie nowoczesnych mechanizmów języka JavaScript

JavaScript został znacząco unowocześniony na przestrzeni ostatnich kilku lat. Do języka tego dodano szereg przydatnych rozszerzeń i użytecznych, wbudowanych funkcji, np. do obsługi tablic. Nie wszystkie przeglądarki obsługują najnowsze zmiany, dlatego narzędzia deweloperskie Vue.js zawierają pakiet Babel, który pozwala na przekształcenie kodu z najnowszej wersji języka JavaScript w starszą, która z powodzeniem będzie działać nawet w starszych przeglądarkach. Dzięki temu można pisać nowoczesny kod bez obaw o kompatybilność ze starszymi przeglądarkami. Babel dotyczy jednak tylko języka JavaScript, a nie dodatkowych API, z których czasami chcemy skorzystać, np. API lokalnej pamięci (Local Storage), z którego korzystałem w rozdziale 1.

W tym przypadku musisz zastanowić się nad wyborem wersji przeglądarki, której chcesz obsługiwać.

Niezwykle pomocna w tym celu okazuje się strona caniuse.com, która w łatwy sposób przedstawia wersje przeglądarek obsługujące dane API lub rozszerzenie.

Definicja funkcji z parametrami

JavaScript pozwala na tworzenie funkcji z parametrami (listing 4.8).

Listing 4.8. Definicja funkcji z parametrami w pliku src/main.js

```
function myFunc(name, weather) {
    console.log("Cześć, " + name + "!");
    console.log("Dziś jest " + weather + ".");
}
myFunc("Adam", "słonecznie");
```

Do funkcji myFunc dodałem dwa parametry — name i weather. JavaScript jest językiem dynamicznie typowanym, co oznacza m.in., że nie musimy określać typów parametrów w definicji funkcji.Więcej na temat dynamicznego typowania znajdziesz w dalszej części rozdziału, podczas omawiania przeze mnie zmiennych w języku JavaScript. Wywołanie funkcji z parametrami wymaga podania ich jako argumenty, np.:

```
...
myFunc("Adam", "słonecznie");
...
```

Efekt wywołania funkcji jest następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Stosowanie parametrów domyślnych i parametru reszty

Liczba argumentów przekazanych w wywołaniu funkcji nie musi zgadzać się z liczbą parametrów w jej definicji. Jeśli wywołasz funkcję z mniejszą liczbą argumentów (w porównaniu z liczbą parametrów), pozostałe parametry otrzymają specjalną wartość `undefined`. Wywołanie funkcji z większą liczbą argumentów spowoduje zignorowanie tych, które wykraczają poza liczbę parametrów zdefiniowanych w funkcji.

Konsekwencją takiej sytuacji jest brak możliwości utworzenia dwóch funkcji o tej samej nazwie z różną liczbą parametrów. Takie zachowanie nosi nazwę **polimorfizmu** i choć jest obsługiwane w językach takich jak Java czy C# nie istnieje w JavaScriptie. Jeśli więc zdefiniujesz dwie funkcje o tej samej nazwie, druga definicja po prostu zastąpi pierwszą.

W przypadku gdy liczba parametrów funkcji nie zgadza się z liczbą argumentów przy wywołaniu, możesz skorzystać z dwóch dodatkowych mechanizmów. **Parametry domyślne** (ang. *default parameters*) są używane, gdy w wywołaniu podano mniej parametrów. Dzięki temu można dostarczyć wartości domyślne dla parametrów, które nie otrzymały argumentów w momencie wywołania (listing 4.9).

Listing 4.9. Stosowanie parametru domyślnego w pliku src/main.js

```
function myFunc(name, weather = "słonecznie") {
    console.log("Cześć, " + name + "!");
    console.log("Dziś jest " + weather + ".");
}
myFunc("Adam");
```

Parametr `weather` otrzymuje wartość domyślną `słonecznie`, a zatem w przypadku, gdy funkcja zostanie wywołana z jednym argumentem, uzyskamy następujący efekt:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Parametr reszty (ang. *rest parameters*) pozwala na przechwycenie wszelkich argumentów wykraczających poza listę zdefiniowanych parametrów (listing 4.10).

Listing 4.10. Stosowanie parametru reszty w pliku src/main.js

```
function myFunc(name, weather, ...extraArgs) {
    console.log("Cześć, " + name + "!");
    console.log("Dziś jest " + weather + ".");
    for (let i = 0; i < extraArgs.length; i++) {
        console.log("Dodatkowy argument:" + extraArgs[i]);
    }
}
myFunc("Adam", "słonecznie", "raz", "dwa", "trzy");
```

Parametr reszty musi być zadeklarowany jako ostatni, a jego nazwa jest poprzedzona wielokropkiem (trzema kropkami). Parametr reszty stanowi tablicę, w której zostaną umieszczone wszelkie dodatkowe argumenty. W tym przypadku wydruk będzie następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
Dodatkowy argument: raz
Dodatkowy argument: dwa
Dodatkowy argument: trzy
```

Tworzenie funkcji zwracających wyniki

Zwracanie wyników z funkcji jest możliwe za pomocą słowa kluczowego `return`. Listing 4.11 przedstawia funkcję, która zwraca wynik.

Listing 4.11. Zwracanie wyniku w funkcji w pliku src/main.js

```
function myFunc(name) {
    return ("Cześć, " + name + "!");
}
console.log(myFunc("Adam"));

Ta funkcja definiuje jeden parametr, a następnie korzysta z jego wartości w celu wygenerowania wyniku. Wywołuję funkcję i przekazuję jej wynik jako argument do funkcji console.log:
```

```
...
console.log(myFunc("Adam"));
...
```

Zwróć uwagę, że nie musisz deklarować typu zwracanej wartości ani samego faktu, że zamierzasz cokolwiek zwrócić. Wynik działania będzie następujący:

```
Cześć, Adam!
```

Przekazywanie funkcji przez argument

JavaScript pozwala na traktowanie funkcji jako obiektów, dzięki czemu możemy przekazać funkcję w postaci argumentu do innej funkcji (listing 4.12).

Listing 4.12. Przekazywanie funkcji przez argument w pliku src/main.js

```
function myFunc(nameFunction) {
    return ("Cześć, " + nameFunction() + "!");
}
console.log(myFunc(function () {
    return "Adam";
}));
```

Funkcja myFunc wprowadza parametr o nazwie nameFunction, który następnie jest wywoływany w treści funkcji. Wynik wywołania jest dołączony do łańcucha znaków. Efekt jest następujący:

Cześć, Adam!

Funkcje można łączyć ze sobą, tworząc skomplikowane konstrukcje z prostych i łatwo testowalnych fragmentów kodu (listing 4.13).

Listing 4.13. Łączenie wywołań funkcji w łańcuch (plik src/main.js)

```
function myFunc(nameFunction) {
    return ("Cześć, " + nameFunction() + "!");
}
function printName(nameFunction, printFunction) {
    printFunction(myFunc(nameFunction));
}
printName(function () { return "Adam" }, console.log);
```

Efekt będzie następujący:

Cześć, Adam!

Stosowanie funkcji strzałkowych

Funkcje strzałkowe (ang. *arrow functions, fat arrow functions*; znane też jako **wyrażenia lambda** — *lambda expressions*) stanowią odmienną metodę definiowania funkcji, zwłaszcza gdy konieczne jest przekazanie funkcji w formie argumentu do innej funkcji. Listing 4.14 zastępuje funkcje z poprzedniego przykładu funkcjami strzałkowymi.

Listing 4.14. Zastosowanie funkcji strzałkowych w pliku src/main.js

```
const myFunc = (nameFunction) => ("Cześć, " + nameFunction() + "!");
const printName = (nameFunction, printFunction) =>
    printFunction(myFunc(nameFunction));
printName(function () { return "Adam" }, console.log);
```

Funkcje te działają identycznie z tymi z listingu 4.13. Funkcja strzałkowa składa się z trzech części — parametrów wejściowych, znaków równości i „większy niż” ($=>$, czyli strzałka), a także wyniku działania. Słowo kluczowe `return` i nawiasy klamrowe są konieczne tylko, jeśli funkcja strzałkowa składa się z więcej niż jednej instrukcji. Dodatkowe przykłady funkcji strzałkowych znajdziesz w dalszej części rozdziału i książce.

-
- **Ostrzeżenie** Funkcje strzałkowe nie mogą być stosowane wszędzie w Vue.js, dlatego zwróć uwagę na dalsze przykłady i zapamiętaj, gdzie można z nich korzystać.
-

Zmienne i typy

Słowo kluczowe `let` służy do definiowania zmiennych, a także przypisywania wartości do zmiennej w pojedynczej instrukcji, w przeciwieństwie do słowa kluczowego `const`, odpowiadającego za tworzenie stałych (których wartości nie mogą być zmieniane).

Gdy korzystasz ze słowa `let` lub `const`, zmienna bądź stała są dostępne jedynie w tym obszarze kodu, w którym zostały zadeklarowane — obszar ten nosi nazwę **zasięgu** (ang. *scope*) zmiennej/stałej.

Funkcjonowanie zasięgów tłumaczy listing 4.15.

Listing 4.15. Zastosowanie słowa kluczowego let w deklaracji zmiennych (plik src/main.js)

```
function messageFunction(name, weather) {
    let message = "Cześć, Adam!";
    if (weather == "słonecznie") {
        let message = "To ładny dzień.";
        console.log(message);
    } else {
        let message = "Dziś jest " + weather + ".";
        console.log(message);
    }
    console.log(message);
}
messageFunction("Adam", "pochmurno");
```

W tym przykładzie słowo let pojawia się w trzech instrukcjach. Zasięg każdej zmiennej jest ograniczony do obszaru kodu, w którym jest ona zdefiniowana, przez co wynik działania kodu jest następujący:

```
Dziś jest pochmurno.  
Cześć, Adam!
```

Warto pamiętać, że istnieje jeszcze jedno słowo kluczowe, za pomocą którego można deklarować zmienne — var. Słowa kluczowe let i const są stosunkowo nowym dodatkiem do języka JavaScript — zostały wprowadzone, aby rozwiązać problemy wynikające ze stosowania słowa var. Listing 4.16 różni się od listingu 4.15 zastąpieniem słowa let przez var.

Listing 4.16. Zamiana słowa let na var w deklaracji zmiennych (src/main.js)

```
function messageFunction(name, weather) {
    var message = "Cześć, Adam!";
    if (weather == "słonecznie") {
        var message = "To ładny dzień.";
        console.log(message);
    } else {
        var message = "Dziś jest " + weather + ".";
        console.log(message);
    }
    console.log(message);
}
messageFunction("Adam", "pochmurno");
```

Stosowanie słów let i const

Warto stosować słowo kluczowe const zawsze, gdy nie chcemy zmieniać wartości. Dzięki temu, jeśli przypadkowo zmodyfikujesz taką wartość, wbrew założeniom, otrzymasz błąd. Z drugiej strony rzadko korzystam z tej reguły, ponieważ ciężko odzwyczać mi się od używania słowa var, a ponadto na co dzień programuję w różnych językach i staram się unikać konstrukcji, które nie funkcjonują w innych językach. Jeśli jednak dopiero zaczynasz przygodę z JavaScriptem, zdecydowanie zalecam stosowanie słów const i let zamiast naśladowania moich starych, niedobrych nawyków.

Po zapisaniu zmian zobaczysz następujący wynik:

```
Dziś jest pochmurno.  
Dziś jest pochmurno.
```

Błędne działanie wynika z faktu, że słowo var tworzy zmienną w zasięgu funkcji, przez co wszystkie instrukcje odwołujące się do nazwy zmiennej message dotyczą tej samej zmiennej. Taka sytuacja może prowadzić do nieoczekiwanych zachowań nawet dla doświadczonych programistów JavaScript, dlatego tak bardzo zachęcam do stosowania słowa let.

Zmienne a domknięcia

Gdy definiujesz funkcję wewnętrz r innej funkcji — tworząc w ten sposób funkcję **zewnętrzną i wewnętrzną** — wewnętrzna funkcja jest w stanie uzyskać dostęp do zmiennych zadeklarowanych w funkcji zewnętrznej, korzystając z mechanizmu **domknięcia** (ang. *closure*), np.:

```
function myFunc(name) {
  let myLocalVar = "słonecznie";
  let innerFunction = function () {
    return ("Cześć, " + name + "! Dziś jest " + myLocalVar + ".");
  }
  return innerFunction();
}
console.log(myFunc("Adam"));
```

W tym przykładzie funkcja wewnętrzna ma dostęp do zmiennych lokalnych funkcji zewnętrznej, w tym jej parametrów. Mechanizm domknięć pozwala na uniknięciu przekazywania wszystkich istotnych informacji z funkcji zewnętrznej do wewnętrznej za pomocą jawnie określonych parametrów. Trzeba jednak zachować ostrożność, ponieważ regularne używanie typowych nazw zmiennych, takich jak counter czy index, może doprowadzić do przypadkowego, niechcianego użycia zmiennej z funkcji zewnętrznej.

Typy prymitywne

JavaScript wprowadza trzy typy prymitywne: string, number i boolean. Być może ta lista nie jest zbyt długa, ale typy te dają sporo elastyczności.

- **Wskazówka** Przyznam się do pewnego uproszczenia. Możesz spotkać się jeszcze z trzema innymi wartościami prymitywnymi. Zmienne zadeklarowane, ale bez przypisanej wartości, otrzymują specjalną wartość undefined, podczas gdy wartość null oznacza, że zmienna nie ma żadnej konkretnej wartości (podobnie jak w innych językach programowania). Ostatnim typem prymitywnym jest Symbol, który reprezentuje niezmienialną wartość będącą unikatowym identyfikatorem — ten typ nie jest jednak obecnie jeszcze zbyt popularny.

Obsługa wartości logicznych

Typ boolean składa się z dwóch wartości: true i false. Listing 4.17 przedstawia przykład użycia ich obu, jednak zazwyczaj z wyrażeniami logicznymi można spotkać się w warunkach, np. w instrukcji if. W tym przypadku nie załączam wydruku z konsoli.

Listing 4.17. Definicja wartości logicznych (plik src/main.js)

```
let firstBool = true;
let secondBool = false;
```

Obsługa łańcuchów znaków

Łańcuchy znaków, czyli po prostu teksty, definiuje się, korzystając z pojedynczych lub podwójnych cudzysłówów (listing 4.18).

Listing 4.18. Definicja zmiennych tekstowych (*src/main.js*)

```
let firstString = "To jest tekst.";
let secondString = 'I to też.';
```

Znaki cudzysłówów muszą do siebie pasować, np. nie możesz zacząć łańcucha znaków pojedynczym cudzysłowem, a zakończyć podwójnym. Również tutaj nie załączam wydruku z konsoli. Typ *string* w JavaScriptie zawiera szereg przydatnych metod i właściwości, z których te najbardziej przydatne przedstawiam w tabeli 4.2.

Tabela 4.2. Przydatne metody i właściwości typu *string*

Nazwa	Opis
<code>length</code>	Ta właściwość określa liczbę znaków w łańcuchu.
<code>charAt(<i>indeks</i>)</code>	Ta metoda zwraca znak znajdujący się pod podanym indeksem.
<code>concat(<i>łańcuch</i>)</code>	Ta metoda tworzy nowy łańcuch znaków, który łączy tekst będący argumentem metody z tekstem, z którego poziomu wywołano tę metodę.
<code>indexOf(<i>łańcuch</i>, <i>początek</i>)</code>	Ta metoda zwraca indeks pierwszego wystąpienia argumentu <i>łańcuch</i> lub <code>-1</code> , jeśli nie został on znaleziony. Opcjonalny argument <i>początek</i> określa indeks, od którego zostanie rozpoczęte poszukiwanie.
<code>replace(<i>łańcuch</i>, <i>nowy Łańcuch</i>)</code>	Ta metoda zamienia wszystkie wystąpienia łańcucha <i>łańcuch</i> na łańcuch <i>nowy Łańcuch</i> .
<code>slice(<i>początek</i>, <i>koniec</i>)</code>	Ta metoda zwraca fragment łańcucha znaków pomiędzy indeksami <i>początek</i> i <i>koniec</i> .
<code>split(<i>łańcuch</i>)</code>	Ta metoda dzieli łańcuch na tablicę łańcuchów oddzielonych łańcuchem <i>łańcuch</i> .
<code>toUpperCase()</code> , <code>toLowerCase()</code>	Te metody zwracają łańcuchy znaków, w których wszystkie znaki są wielkie lub małe.
<code>trim()</code>	Ta metoda zwraca łańcuch, w którym wszystkie wiodące i kończące łańcuch białe znaki są usunięte.

Łańcuchy szablonowe

Typowym zadaniem, przed którym staje programista, jest połączenie treści statycznej z dynamicznie generowanymi danymi. Tradycyjnym podejściem była konkatencja łańcuchów znaków, z której korzystałem już wiele razy do tej pory:

```
...
let message = "Dziś jest " + weather + ".";
...
```

JavaScript obsługuje także łańcuchy (napisy) szablonowe, które pozwalają na dynamiczne wstawianie danych w sposób znacznie mniej podatny na błędy (listing 4.19).

Listing 4.19. Przykład użycia łańcuchów szablonowych

```
function messageFunction(weather) {
  let message = `Dziś jest ${weather}.`;
  console.log(message);
```

```

    }
messageFunction("słonecznie");

```

Łańcuchy szablonowe są umieszczane wewnętrz grawisów (``). Wartości dynamiczne są oznaczane za pomocą nawiasów klamrowych, poprzedzonych znakiem dolara. Poniższy łańcuch dołącza do łańcucha zmiennej `weather`:

```

...
let message = `Dziś jest ${weather}.`;
...

```

W rezultacie otrzymamy tekst:

Dziś jest słonecznie.

Obsługa liczb

Typ `number` jest używany do obsługi liczb całkowitych i zmiennoprzecinkowych (rzeczywistych). Przykład użycia przedstawia listing 4.20.

Listing 4.20. Przykłady użycia typu number (src/main.js)

```

let daysInWeek = 7;
let pi = 3.14;
let hexValue = 0xFFFF;

```

Nie musisz określać jawnie rodzaju liczby, z jakiego chcesz skorzystać. Wystarczy podać wartość, a JavaScript zadziała zgodnie z oczekiwaniemi. W tym przypadku zadeklarowałem dwie liczby całkowite i jedną zmiennoprzecinkową, korzystając przy tym z zapisu `0x`, który pozwala na zapisanie liczby w formacie szesnastkowym.

Operatory języka JavaScript

JavaScript wprowadza obszerny zbiór operatorów. Zestawienie najważniejszych zawiera tabela 4.3.

Tabela 4.3. Przydatne operatory języka JavaScript

Operator	Opis
<code>++, --</code>	Pre-/postinkrementacja, pre-/postdekrementacja.
<code>+, -, *, /, %</code>	Dodawanie, odejmowanie, mnożenie, dzielenie, reszta z dzielenia.
<code><, <=, >, >=</code>	Mniejszy niż, mniejszy lub równy, większy niż, większy lub równy.
<code>==, !=</code>	Równość, różność.
<code>==:, !==</code>	Identyczność, nieidentyczność.
<code>&&, </code>	Logiczne I, logiczne LUB (operator <code> </code> jest używany do obsługi wartości <code>null</code>).
<code>=</code>	Przypisanie.
<code>+</code>	Konkatenacja (złączenie) łańcuchów znaków.
<code>?:</code>	Trójargumentowy operator warunkowy.

Instrukcje warunkowe

Wiele operatorów języka JavaScript działa w połączeniu z instrukcjami warunkowymi. W tej książce często stosuję instrukcję `if/else` i `switch`. Listing 4.21 przedstawia przykład użycia obu w znany dla programistów sposób.

Listing 4.21. Przykład użycia instrukcji warunkowych (*src/main.js*)

```
let name = "Adam";
if (name == "Adam") {
    console.log("Imię to Adam.");
} else if (name == "Jacek") {
    console.log("Imię to Jacek.");
} else {
    console.log("Imię jest inne.");
}
switch (name) {
    case "Adam":
        console.log("Imię to Adam.");
        break;
    case "Jacek":
        console.log("Imię to Jacek.");
        break;
    default:
        console.log("Imię jest inne.");
        break;
}
```

W efekcie otrzymamy następujący wydruk:

```
Imię to Adam.
Imię to Adam.
```

Operator równości a operator identyczności

Operatory równości i identyczności mają szczególne znaczenie. Operator równości podejmie próbę sprawdzenia obu operandów do tego samego typu i dopiero wtedy wykona porównanie. Zachowanie to jest przydatne, o ile wiesz, co się dzieje. Listing 4.22 przedstawia operator równości w praktyce.

Listing 4.22. Zastosowanie operatora równości w pliku *src/main.js*

```
let firstVal = 5;
let secondVal = "5";
if (firstVal == secondVal) {
    console.log("Operandy są takie same.");
} else {
    console.log("Operandy NIE są takie same.");
}
```

Wynik będzie następujący:

```
Operandy są takie same.
```

JavaScript konwertuje operandy na ten sam typ, po czym porównuje tak uzyskane wartości. Krótko mówiąc, operator równości porównuje wartości niezależnie od typu. Jeśli chcesz mieć pewność, że wartości i ich typy są takie same, skorzystaj z operatora identyczności (`==`, trzy znaki równości zamiast dwóch, jak w listingu 4.23).

Listing 4.23. Operator identyczności w pliku src/main.js

```
let firstVal = 5;
let secondVal = "5";
if (firstVal === secondVal) {
    console.log("Operandy są takie same.");
} else {
    console.log("Operandy NIE są takie same.");
}
```

W tym przykładzie operator identyczności zauważa różnice między typami, przez co wynik będzie następujący

Operandy NIE są takie same.

Jawna konwersja typów

Operator konkatenacji łańcuchów znaków (+) ma wyższy priorytet od operatora dodawania liczb (również +), przez co w JavaScriptcie w pierwszej kolejności dojdzie do złączenia łańcuchów, a nie do wykonania dodawania. Rodzi to wątpliwości, zwłaszcza biorąc pod uwagę, że JavaScript konwertuje typy wyrażeń w dość swobodny sposób (listing 4.24).

Listing 4.24. Konkatenacja jest ważniejsza od dodawania (src/main.js)

```
let myData1 = 5 + 5;
let myData2 = 5 + "5";
console.log("Wynik 1:" + myData1);
console.log("Wynik 2:" + myData2);
```

Efekt wykonania kodu jest następujący:

Wynik 1: 10
Wynik 2: 55

Drugi wynik może wydać się mylący. Mimo iż naszym celem w obu przypadkach było dodanie liczb, do gry włączył się operator konkatenacji (w drugim przypadku mamy do czynienia z łańcuchem znaków), który doprowadził do złączenia, a nie dodania operandów. Aby uniknąć takich sytuacji, konieczne jest jawne skonwertowanie wartości przed wykonaniem żądanej operacji.

Konwersja liczb na łańcuchy znaków

Jeśli pracujesz ze zmiennymi liczbowymi, ale chcesz potraktować je jako łańcuch znaków, możesz skorzystać z metody `toString` (listing 4.25).

Listing 4.25. Zastosowanie metody `number.toString` (src/main.js)

```
let myData1 = (5).toString() + String(5);
console.log("Wynik:" + myData1);
```

Zwróci uwagę, że wartość liczbową została umieszczona w nawiasach okrągłych i dopiero wtedy wywoałem na niej metodę `toString`. Wynika to z faktu, że prymitywna wartość liczbową musi zostać poddana konwersji na typ `number` — dopiero wtedy można wywołać metody tego typu. Alternatywnie można przekazać literal liczbowy do funkcji `String`. Oba podejścia dają ten sam efekt, czyli wykonanie konwersji liczby na łańcuch znaków. Operator + w przypadku takich wartości będzie złączał (konkatenował) operandy. Wynik działania kodu jest następujący:

Wynik: 55

Zawsze można skorzystać też z innych metod, które dają nieco więcej kontroli nad tekstową reprezentacją liczby. Metody te opisuję pokrótko w tabeli 4.4. Wszystkie one są zdefiniowane w typie number.

Tabela 4.4. Użyteczne metody liczbowo-tekstowe

Metoda	Opis
<code>toString()</code>	Ta metoda zwraca tekstową reprezentację dziesiętną liczby.
<code>toString(2), toString(8), toString(16)</code>	Te metody zwracają tekstowe reprezentacje liczby w formacie dwójkowym, ósemkowym lub heksadecymalnym (szesnastkowym).
<code>toFixed(<i>n</i>)</code>	Ta metoda zwraca tekstową reprezentację liczby rzeczywistej z <i>n</i> cyframi po przecinku.
<code>toExponential(<i>n</i>)</code>	Ta metoda zwraca tekstową reprezentację liczby z wykorzystaniem zapisu naukowego (wykładniczego), z jedną cyfrą przed separatorem dziesiętnym i <i>n</i> cyframi po nim.
<code>toPrecision(<i>n</i>)</code>	Ta metoda zwraca tekstową reprezentację liczby z <i>n</i> cyframi znaczącymi z wykorzystaniem zapisu naukowego (jeśli jest konieczny).

Konwersja łańcuchów znaków na liczby

Podobny mechanizm pozwala na konwersję łańcuchów znaków na liczby, dzięki czemu możemy wykonywać zwykłe dodawanie. W tym celu korzystamy z funkcji Number (listing 4.26).

Listing 4.26. Konwersja łańcuchów znaków na liczby w pliku *src/main.js*

```
let firstVal = "5";
let secondVal = "5";
let result = Number(firstVal) + Number(secondVal);
console.log("Wynik:" + result);
```

Efekt jest następujący:

Wynik: 10

Funkcja Number działa w niezwykle ściśły sposób, przetwarzając (parsując) wartości tekstowe. Zawsze możesz też skorzystać z funkcji parseInt i parseFloat. Wszystkie trzy metody są opisane w tabeli 4.5.

Tabela 4.5. Funkcje przydatne do konwersji tekstów na liczby

Metoda	Opis
<code>Number(<i>łańcuch</i>)</code>	Ta metoda parsuje łańcuch znaków w celu uzyskania liczby całkowitej lub rzeczywistej.
<code>parseInt(<i>łańcuch</i>)</code>	Ta metoda parsuje łańcuch znaków w celu uzyskania liczby całkowitej.
<code>parseFloat(<i>łańcuch</i>)</code>	Ta metoda parsuje łańcuch znaków w celu uzyskania liczby całkowitej lub rzeczywistej.

Obsługa tablic

Tablice w JavaScriptie działają tak jak w innych językach programowania. Listing 4.27 przedstawia metodę tworzenia tablicy i wypełniania jej zawartością.

Listing 4.27. Tworzenie tablicy i uzupełnianie jej treścią (src/main.js)

```
let myArray = new Array();
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

Tablica została utworzona za pomocą zapisu `new Array()`. W ten sposób otrzymaliśmy pustą tablicę, którą następnie przypisalem do zmiennej `myArray`. W następnych instrukcjach przypisuję wartości do kolejnych indeksów tablicy (dla tego listingu nie ma wydruku).

W tej sytuacji warto pamiętać o kilku kwestiach. Przede wszystkim nie musiałem podawać rozmiaru tablicy w momencie jej tworzenia. JavaScript sam zadba o zmianę rozmiaru tablicy. Ponadto nie musiałem określić typu danych, które znajdują się w tablicy. Tablice w JavaScriptie mogą przechować dane różnych typów. W tym przykładzie w tablicy znalazły się liczba, napis i wartość logiczna.

Literały tablicowe

Literał tablicowy to konstrukcja, która pozwala na utworzenie i wypełnienie treścią tablicy w jednej instrukcji (listing 4.28).

Listing 4.28. Przykład użycia literała tablicowego (src/main.js)

```
let myArray = [100, "Adam", true];
```

W tym przykładzie zmienna `myArray` otrzymuje nową tablicę, zawierającą elementy umieszczone pomiędzy nawiasami kwadratowymi (przykład nie zawiera wydruku).

Odczyt i modyfikacja zawartości tablicy

Aby odczytać element o danym indeksie, należy skorzystać z nawiasów kwadratowych, umieszczając indeks pomiędzy nimi (listing 4.29).

Listing 4.29. Odczyt danych z tablicy za pomocą indeksu (src/main.js)

```
let myArray = [100, "Adam", true];
console.log(`Indeks 0: ${myArray[0]}`);
```

Modyfikacja wartości jest możliwa przez przypisanie nowej wartości do elementu o danym indeksie. Również i w tym przypadku można przypisać wartość zupełnie nowego typu. Oto efekt działania powyższego kodu:

Indeks 0: 100.

Listing 4.30 przedstawia przykład modyfikacji treści tablicy.

Listing 4.30. Modyfikacja zawartości tablicy w pliku src/main.js

```
let myArray = [100, "Adam", true];
myArray[0] = "Wtorek";
console.log(`Index 0: ${myArray[0]}`);
```

W tym przykładzie pierwszy element został zastąpiony tekstem, w związku z czym efekt działania kodu jest następujący:

Indeks 0: Wtorek

Przeglądanie zawartości tablicy

Przeglądanie zawartości tablicy jest możliwe za pomocą pętli `for` lub metody `forEach`, która otrzymuje funkcję wykonywaną dla każdego elementu tablicy. Oba podejścia pokazano w listingu 4.31.

Listing 4.31. Przeglądanie zawartości tablicy (src/main.js)

```
let myArray = [100, "Adam", true];
for (let i = 0; i < myArray.length; i++) {
    console.log(`Indeks ${i}: ${myArray[i]}`);
}
console.log("---");
myArray.forEach((value, index) => console.log(`Indeks ${index}: ${value}`));
```

Pętla `for` działa identycznie jak analogiczne konstrukcje w innych językach. Liczbę elementów tablicy można pobrać dzięki właściwości `length`.

Funkcja przekazana do metody `forEach` przyjmuje dwa argumenty: aktualny element tablicy do przetworzenia i położenie elementu w tablicy. W powyższym listingu korzystam z notacji strzałkowej w wywołaniu metody `forEach`, i to właśnie w takich sytuacjach jej zastosowanie ma największy sens. Oto efekt działania kodu:

```
Indeks 0: 100
Indeks 1: Adam
Indeks 2: true
---
Indeks 0: 100
Indeks 1: Adam
Indeks 2: true
```

Operator rozwinięcia

Operator rozwinięcia (ang. *spread operator*) służy do przekształcenia tablicy w ciąg niezależnych argumentów funkcji. Listing 4.32 definiuje funkcję, która przyjmuje wiele argumentów, a jest wywoływana na dwa sposoby — bez użycia i z użyciem operatora rozwinięcia.

Listing 4.32. Przykład użycia operatora rozwinięcia (src/main.js)

```
function printItems(numValue, stringValue, boolValue) {
    console.log(`Number: ${numValue}`);
    console.log(`String: ${stringValue}`);
    console.log(`Boolean: ${boolValue}`);
}
let myArray = [100, "Adam", true];
printItems(myArray[0], myArray[1], myArray[2]);
printItems(...myArray);
```

Operator rozwinięcia to wielokropek (zapisywany jako trzy kropki jedna po drugiej). Jego zastosowanie spowoduje rozpakowanie tablicy i przekazanie jej elementów jako odrębnych argumentów.

```
...
printItems(...myArray);
...
```

Operator rozwinięcia pozwala na łatwe łączenie tablic (listing 4.33).

Listing 4.33. Złączanie tablic w pliku *src/main.js*

```
let myArray = [100, "Adam", true];
let myOtherArray = [200, "Robert", false, ...myArray];
myOtherArray.forEach((value, index) => console.log(`Indeks ${index}: ${value}`));
```

Dzięki operatorowi rozwinięcia jestem w stanie przekazać tablicę `myArray` jako pojedynczy element w taki sposób, że jej elementy zostaną dołączone do nowo tworzonej tablicy `myOtherArray` (efekt poniżej):

```
Indeks 0: 200
Indeks 1: Robert
Indeks 2: false
Indeks 3: 100
Indeks 4: Adam
Indeks 5: true
```

Wbudowane metody do obsługi tablic

Typ `Array` zawiera szereg metod przydatnych w pracy z tablicami. Najważniejsze metody przedstawiono w tabeli 4.6.

Skoro wiele metod opisanych w tabeli 4.6 tworzy nową tablicę, metody mogą być łączone w łańcuch wywołań w celu przetwarzania danych (listing 4.34).

Listing 4.34. Przetwarzanie tablicy w pliku *src/main.js*

```
let products = [
  { name: "Kapelusz", price: 24.5, stock: 10 },
  { name: "Kajak", price: 289.99, stock: 1 },
  { name: "Piłka", price: 10, stock: 0 },
  { name: "Buty do biegania", price: 116.50, stock: 20 }
];
let totalValue = products
  .filter(item => item.stock > 0)
  .reduce((prev, item) => prev + (item.price * item.stock), 0);
console.log(`Wartość łączna: ${totalValue.toFixed(2)} PLN`);
```

Korzystam z metody `filter` w celu znalezienia elementów, których wartość atrybutu `stock` jest większa od zera, po czym stosuję metodę `reduce`, aby obliczyć całkowitą wartość produktów w koszyku. Efekt:

```
Wartość łączna: 2864.99 PLN
```

Obsługa obiektów

Tworzenie obiektów jest możliwe na wiele sposobów. Listing 4.35 przedstawia jeden z przykładów.

Listing 4.35. Przykład tworzenia obiektu (*src/main.js*)

```
let myData = new Object();
myData.name = "Adam";
myData.weather = "słonecznie";
console.log(`Cześć, ${myData.name}!`);
console.log(`Dziś jest ${myData.weather}.`);
```

Tabela 4.6. Przydatne metody obsługi tablic

Metoda	Opis
<code>concat(<i>innąTablicą</i>)</code>	Ta metoda zwraca nową tablicę stanowiącą konkatenację tablicy, z której poziomu została wywołana, i tablicy przekazanej w formie argumentu. Istnieje możliwość przekazania wielu tablic w formie argumentów.
<code>join(<i>separator</i>)</code>	Ta metoda łączy wszystkie elementy w tablicy w jeden łańcuch znaków. Argument <i>separator</i> jest używany jako łącznik pomiędzy elementami tablicy.
<code>pop()</code>	Ta metoda usuwa i jednocześnie zwraca ostatni element tablicy.
<code>shift()</code>	Ta metoda usuwa i jednocześnie zwraca pierwszy element tablicy.
<code>push(<i>element</i>)</code>	Ta metoda dodaje element na końcu tablicy.
<code>unshift(<i>element</i>)</code>	Ta metoda wstawia element na początku tablicy.
<code>reverse()</code>	Ta metoda zwraca nową tablicę zawierającą elementy w odwrotnym porządku.
<code>slice(<i>początek</i>, <i>koniec</i>)</code>	Ta metoda zwraca fragment tablicy.
<code>sort()</code>	Ta metoda sortuje tablicę. Można także dołączyć funkcję do porównania elementów, dzięki której będzie możliwe wykonywanie sortowania według własnych kryteriów.
<code>splice(<i>indeks</i>, <i>liczba</i>)</code>	Ta metoda usuwa zadaną liczbę elementów, począwszy od elementu o podanym indeksie.
<code>every(<i>test</i>)</code>	Ta metoda wywołuje podaną funkcję dla wszystkich elementów tablicy i zwraca <code>true</code> , jeśli wszystkie wywołania tej funkcji również zwróciły wartość <code>true</code> (<code>false</code> w przeciwnym razie).
<code>some(<i>test</i>)</code>	Ta metoda wywołuje podaną funkcję dla wszystkich elementów tablicy i zwraca <code>true</code> , jeśli minimum jedno wywołanie tej funkcji zakończyło się zwróceniem wartości <code>true</code> (<code>false</code> w przeciwnym razie).
<code>filter(<i>test</i>)</code>	Ta metoda zwraca nową tablicę zawierającą elementy, dla których przekazana w argumencie funkcja zwraca wartość <code>true</code> .
<code>find(<i>test</i>)</code>	Ta metoda zwraca pierwszy element tablicy, dla którego przekazana w argumencie funkcja zwraca wartość <code>true</code> .
<code>findIndex(<i>test</i>)</code>	Ta metoda zwraca indeks pierwszego elementu tablicy, dla którego przekazana w argumencie funkcja zwraca wartość <code>true</code> .
<code>forEach(<i>wywołanie</i>)</code>	Ta metoda wywołuje funkcję <i>wywołanie</i> dla każdego elementu tablicy (jak opisano w powyższym przykładzie).
<code>includes(<i>wartość</i>)</code>	Ta metoda zwraca wartość <code>true</code> , jeśli tablica zawiera określona wartość.
<code>map(<i>wywołanie</i>)</code>	Ta metoda zwraca nową tablicę, zawierającą elementy będące wynikami wywołania funkcji <i>wywołanie</i> na elementach tablicy źródłowej.
<code>reduce(<i>wywołanie</i>)</code>	Ta metoda zwraca wartość będącą wynikiem obliczeń przeprowadzanych na wszystkich elementach tablicy źródłowej.

Obiekt tworzę za pomocą wywołania `new Object()`, a następnie przypisuję wynik (nowo utworzony obiekt) do zmiennej `myData`. Po utworzeniu obiektu definiuję właściwości przez przypisanie do nich wartości:

```
...
myData.name = "Adam";
...
```

Przed wykonaniem tej instrukcji mój obiekt nie zawierał właściwości name. Po jej wykonaniu właściwość istnieje i ma wartość Adam. Odczytanie wartości jest możliwe również przez podanie nazwy zmiennej i nazwy właściwości przedzielonych kropką, np.:

```
...
console.log(`Cześć, ${myData.name}!`);
...
```

Efekt jest następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Literały obiektowe

Obiekt i jego właściwości można zdefiniować w jednej operacji, korzystając z literału (listing 4.36).

Listing 4.36. Tworzenie obiektu za pomocą literału (src/main.js)

```
let myData = {
  name: "Adam",
  weather: "słonecznie"
};
console.log(`Cześć, ${myData.name}!`);
console.log(`Dziś jest ${myData.weather}.`);
```

Każda właściwość, którą chcesz zdefiniować, jest oddzielona od jej wartości znakiem dwukropka (:), a właściwości od siebie są oddzielone przecinkiem (,). Efekt jest identyczny z poprzednim przykładem:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Zmienne jako właściwości obiektów

Jeśli skorzystasz ze zmiennej jako właściwości obiektu, JavaScript skorzysta z nazwy zmiennej jako nazwy właściwości, a z wartością zmiennej — jako wartości właściwości (listing 4.37).

Listing 4.37. Zmienna jako literal obiektowy (src/main.js)

```
let name = "Adam"
let myData = {
  name,
  weather: "słonecznie"
};
console.log(`Cześć, ${myData.name}!`);
console.log(`Dziś jest ${myData.weather}.`);
```

Zmienna name jest używana w celu dodania właściwości do obiektu myData. Właściwość nosi nazwę name, a jej wartość to Adam. Jest to niezwykle użyteczny sposób tworzenia obiektu zwłaszcza, gdy łączysz wiele danych w jeden obiekt (takie zachowanie znajdziesz również w kolejnych przykładach). Kod z listingu 4.37 generuje następujący efekt:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Stosowanie funkcji jako metod

Jedną z możliwości, jaką lubię w JavaScriptie najbardziej, jest dodawanie funkcji do obiektów. Funkcja zdefiniowana w ramach obiektu nosi nazwę metody. Listing 4.38 pokazuje przykład dodania metody do obiektu.

Listing 4.38. Dodawanie metody do obiektu w pliku src/main.js

```
let myData = {
    name: "Adam",
    weather: "słonecznie",
    printMessages: function () {
        console.log(`Cześć, ${myData.name}!`);
        console.log(`Dziś jest ${myData.weather}.`);
    }
};
myData.printMessages();
```

W tym przykładzie korzystam z funkcji, aby utworzyć metodę `printMessages`. Zwróć uwagę, że odwołanie się do właściwości zdefiniowanych w ramach tego obiektu wymaga użycia słowa kluczowego `this`. Zastosowanie funkcji jako metody powoduje niejawne przekazanie obiektu, w którym funkcja została zadeklarowana, pod postacią zmiennej `this`. Efekt działania kodu jest następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Metody można definiować także bez słowa kluczowego `function` (listing 4.39).

Listing 4.39. Definicja metody w pliku src/main.js

```
let myData = {
    name: "Adam",
    weather: "słonecznie",
    printMessages() {
        console.log(`Cześć, ${myData.name}!`);
        console.log(`Dziś jest ${myData.weather}.`);
    }
};
myData.printMessages();
```

Moim zdaniem jest to bardziej naturalny sposób tworzenia metod, dlatego korzystam z niego w wielu przykładach w tej książce. Efekt jest następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Kopiowanie właściwości pomiędzy obiekttami

W przykładach w kolejnych rozdziałach kopuję wszystkie właściwości z jednego obiektu do drugiego, aby pokazać różne funkcje Vue.js. JavaScript udostępnia w tym celu metodę `Object.assign` (listing 4.40).

Listing 4.40. Kopiowanie właściwości obiektu (src/main.js)

```
let myData = {
    name: "Adam",
    weather: "słonecznie",
    printMessages() {
```

```

        console.log(`Cześć, ${myData.name}`);
        console.log(`Dziś jest ${myData.weather}.`);
    }
};

let secondObject = {};
Object.assign(secondObject, myData);
secondObject.printMessages();

```

W tym przykładzie tworzymy obiekt, który nie ma właściwości, i stosujemy metodę `Object.assign` do skopiowania właściwości (wraz z wartościami) z obiektu `myData`. Efekt jest następujący:

```
Cześć, Adam!
Dziś jest słonecznie.
```

Moduły w języku JavaScript

Wszystkie dotychczasowe przykłady w tym rozdziale były zawarte w jednym pliku JavaScript. Jest to rozwiązanie właściwe dla prostszych przypadków, ale złożona aplikacja webowa może zawierać dużą ilość kodu, praktycznie niemożliwą do umieszczenia w pojedynczym pliku.

Aby podzielić aplikację na dużą liczbę możliwych do zarządzania fragmentów, JavaScript obsługuje mechanizm modułów. Każdy moduł zawiera kod, z którego mogą korzystać inne fragmenty aplikacji. W kolejnych punktach omawiam różne praktyki tworzenia i używania modułów.

Tworzenie i używanie modułów

Moduły na ogół definiuje się w odrębnych katalogach, dlatego utworzyłem podkatalog `src/math`s i dodałem tam plik `sum.js` (listing 4.41).

Listing 4.41. Zawartość pliku `sum.js` w katalogu `src/math`s

```

export default function(values) {
    return values.reduce((total, val) => total + val, 0);
}

```

Plik `sum.js` zawiera funkcję, która przyjmuje tablicę argumentów. Dzięki funkcji `reduce()` jesteśmy w stanie obliczyć sumę elementów tej tablicy. Nie to jest jednak istotą tego przykładu — tworzymy funkcję w odrębnym pliku po to, aby potraktować go jak moduł.

W listingu 4.41 pojawiają się dwa istotne słowa kluczowe, często spotykane przy tworzeniu modułów. Słowo `export` oznacza, że dana funkcja będzie dostępna poza modułem. Domyslnie funkcje zadeklarowane w module są prywatne, tak więc zastosowanie słowa `export` jest konieczne, jeśli chcemy z nich skorzystać na zewnątrz modułu. Słowo `default` zaś jest używane, gdy moduł udostępnia jeden mechanizm, tak jak w tym przypadku. Razem słowa `export` i `default` zapewniają, że tylko jedna funkcja zostanie udostępniona na zewnątrz aplikacji.

Stosowanie prostego modułu JavaScript

Kolejnym słowem kluczowym, które musimy poznać, jest `import`. Skorzystałem ze słowa `import`, aby uzyskać dostęp do funkcji zadeklarowanej przed chwilą w pliku `main.js` (listing 4.42).

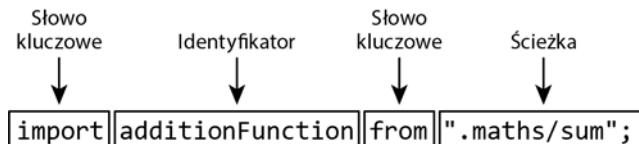
Listing 4.42. Zastosowanie modułu w praktyce

```

import additionFunction from "./maths/sum";
let values = [10, 20, 30, 40, 50];
let total = additionFunction(values);
console.log(`Łącznie: ${total}`);

```

Słowo kluczowe `import` jest używane, aby odwołać się do zewnętrznego modułu. Można z niego korzystać na kilka sposobów, ale w przypadku modułów tworzonych własnoręcznie najczęściej będziesz korzystać ze składni opisanej na rysunku 4.3.



Rysunek 4.3. Dodawanie zależności do modułu w pliku JavaScript

Po słowie kluczowym `import` występuje identyfikator, czyli nazwa, za pomocą której będziemy odwoływać się do danej funkcji. W naszym przypadku funkcja nosi nazwę `additionFunction`.

- **Wskazówka** Zwróć uwagę, że instrukcja `import` z podanym identyfikatorem powoduje jawne narzucenie nazwy, pod którą dana funkcja będzie znana. Wiele instrukcji `import` odwołujących się do tego samego modułu może stosować różne nazwy w celu odwołania się do tej samej funkcji.

Za identyfikatorem znajduje się słowo kluczowe `from`, po którym umieszcza się położenie (ścieżkę) modułu. Warto zwrócić uwagę zwłaszcza na ostatnią część instrukcji, ponieważ później zapisana ścieżka skutkuje nieco innym zachowaniem podczas importowania.

W czasie budowania aplikacji narzędzia Vue.js wykryją instrukcję `import` i dołączą funkcję z pliku `sum.js` do pliku w języku JavaScript, który to plik zostanie odesłany do przeglądarki w celu uruchomienia aplikacji. Identyfikator użyty w instrukcji `import` jest używany do uzyskania dostępu do funkcji w module, analogicznie jak w przypadku funkcji zdefiniowanych lokalnie.

```

...
let total = additionFunction(values);
...

```

Jeśli prześledzisz konsolę przeglądarki, zobaczyś, że listing 4.42 wygeneruje następujący wynik:

Łącznie: 150

Położenie modułów JavaScript

Położenie modułu zmienia sposób, w jaki narzędzia odpowiedzialne za budowanie będą go szukać podczas generowania pliku JavaScript wysyłanego do przeglądarki. W przypadku modułów definiowanych własnoręcznie położenie jest określone za pomocą ścieżki względnej, rozpoczynając od jednej lub dwóch kropek, w zależności od tego, czy ścieżka jest określana względem katalogu bieżącego, czy też jego rodzica. W listingu 4.42 położenie zaczyna się od kropki.

```

...
import additionFunction from "./maths/sum";
...

```

To położenie informuje narzędzia, że istnieje zależność w module `sum`, która znajduje się w katalogu `maths`. Ten katalog znajduje się z kolei w tym samym katalogu co plik, z którego poziomu wykonujemy instrukcję `import` (zwróć uwagę, że pomijamy rozszerzenie pliku w lokalizacji).

Alternatywą dla określania ścieżek w sposób wzajemny do bieżącego pliku jest podanie ścieżki względnej do katalogu projektu:

```
...
import additionFunction from "@/maths/sum";
...
```

Jeżeli położenie (ścieżka) jest poprzedzona znakiem @, moduł jest lokalizowany względem katalogu *src*.

Jeśli ominiesz początkowe kropki i znak @, instrukcja import będzie szukać zależności w katalogu *node_modules*, gdzie instalowane są moduły w trakcie konfiguracji projektu. Jest to dobre miejsce dla wszelkich pakietów pochodzących od zewnętrznych twórców, np. Vue.js, stąd też spotkasz się z konstrukcjami w postaci:

```
...
import Vue from "vue";
...
```

Położenie w tym przypadku nie zaczyna się od kropki, tak więc moduł vue będzie poszukiwany w katalogu *node_modules* projektu. Pakiet ten dostarcza standardowe funkcje aplikacji Vue.js.

Tworzenie wielu mechanizmów w jednym module

Moduły mogą zawierać więcej niż jedną funkcję lub wartość, co pozwala na grupowanie powiązanych funkcji. W ramach przykładu utworzyłem plik *operations.js* w katalogu *src/math*s i dodałem do niego kod z listingu 4.43.

Listing 4.43. Zawartość pliku *operations.js* w katalogu *src/math*s

```
export function multiply(values) {
    return values.reduce((total, val) => total * val, 1);
}
export function subtract(amount, values) {
    return values.reduce((total, val) => total - val, amount);
}
export function divide(first, second) {
    return first / second;
}
```

Ten moduł definiuje trzy funkcje, do których zostało zastosowane słowo kluczowe export. W przeciwieństwie do poprzedniego przykładu słowo kluczowe default nie jest używane, a każda funkcja ma swoją nazwę. W związku z tym przy importowaniu modułu musimy zastosować nieco inne podejście (listing 4.44).

Listing 4.44. Zastosowanie modułu w pliku *src/main.js*

```
import additionFunction from "./maths/sum";
import { multiply, subtract } from "./maths/operations";
let values = [10, 20, 30, 40, 50];
console.log(`Suma: ${additionFunction(values)} `);
console.log(`Iloczyn: ${multiply(values)} `);
console.log(`Różnica: ${subtract(1000, values)} `);
```

Nawiąsy, które występują po słowie kluczowym import, otaczają listę funkcji, z których chcę skorzystać — w tym przypadku multiply i subtract, oddzielonych przecinkiem. Deklaruję zależności od funkcji, których potrzebuję, stąd omijam funkcję divide. Efekt jest następujący:

```
Suma: 150
Iloczyn: 12000000
Różnica: 850
```

Zmiana nazw importowanych funkcji

Jak można zauważyc, funkcje importowane z modułu mają swoje nazwy. Mimo to możemy nadać im własne, korzystając ze słowa kluczowego as (listing 4.45).

Listing 4.45. Zastosowanie aliasu modułu w pliku *src/main.js*

```
import additionFunction from "./maths/sum";
import { multiply, subtract as minus } from "./maths/operations";
let values = [10, 20, 30, 40, 50];
console.log(`Suma: ${additionFunction(values)}~`);
console.log(`Iloczyn: ${multiply(values)}~`);
console.log(`Różnica: ${minus(1000, values)}~`);
```

Zastosowałem słowo kluczowe as, aby zaimportować funkcję subtract pod nazwą minus. Efekt działania jest taki sam jak w listingu 4.44.

Importowanie całego modułu

Wyliczenie wszystkich nazw funkcji w module może stać się niezwykle żmudne w przypadku skomplikowanych modułów. Bardziej eleganckim rozwiązaniem jest zaimportowanie wszystkich funkcji dostarczonych przez moduł, jak w listingu 4.46.

Listing 4.46. Importowanie całego modułu w pliku *src/main.js*

```
import additionFunction from "./maths/sum";
import * as ops from "./maths/operations";
let values = [10, 20, 30, 40, 50];
console.log(`Suma: ${additionFunction(values)}~`);
console.log(`Iloczyn: ${ops.multiply(values)}~`);
console.log(`Różnica: ${ops.subtract(1000, values)}~`);
```

Gwiazdka oznacza, że zostaną zaimportowane wszystkie elementy modułu. Słowo kluczowe as w tym przypadku pozwala na określenie identyfikatora, pod którym będą dostępne te operacje (np. ops.multiply, ops.subtract). Efekt ponownie będzie taki jak w listingu 4.44.

Łączenie wielu plików w jeden moduł

Moduły mogą być deklarowane w wielu plikach, które następnie są łączone za pośrednictwem pliku *index.js*. Ten plik z kolei udostępnia zawartość modułu całej aplikacji. Do katalogu *src/math*s dodaję plik *index.js*, przedstawiony w listingu 4.47.

Listing 4.47. Zawartość pliku *src/math/index.js*

```
import addition from "./sum";
export function mean(values) {
    return addition(values)/values.length;
}
export { addition };
export * from "./operations";
```

Ten plik zaczyna się, tak jak wcześniejsze przykłady, instrukcją import, która określa zależność od funkcji z pliku *sum.js*. Jest to konieczne z uwagi na obecną w tym pliku definicję funkcji mean (średniej).

Instrukcja eksportuje funkcję z pliku *sum.js*, dzięki czemu może być używana na zewnątrz modułu. Nie muszę określać położenia tej funkcji, ponieważ została ona zaimportowana. W takiej sytuacji funkcja jest ujęta w nawiasy klamrowe.

Ostatnia instrukcja eksportuje wszystkie funkcje z pliku *operations.js* bez konieczności ich uprzedniego importu. Takie zachowanie jest przydatne, jeśli chcemy udostępnić funkcje na zewnątrz modułu, ale nie musimy korzystać z nich w pliku *index.js*.

Dzięki plikowi *index.js* jesteśmy w stanie zaimportować wszystkie mechanizmy w pliku *main.js* za pomocą jednej instrukcji (listing 4.48).

Listing 4.48. Importowanie całego modułu (*src/main.js*)

```
import * as math from "./maths";
let values = [10, 20, 30, 40, 50];
console.log(`Suma: ${math.addition(values)})`);
console.log(`Iloczyn: ${math.multiply(values)})`);
console.log(`Różnica: ${math.subtract(1000, values)})`);
console.log(`Średnia: ${math.mean(values)})`);
```

Położenie określone w instrukcji `import` nie wskazuje na plik. W tym przykładzie wszystkie mechanizmy udostępniane przez moduł są dostępne za pomocą identyfikatora `math` bez konieczności podawania, w którym pliku się one znajdują. Efekt działania tego kodu jest następujący:

```
Suma: 150
Iloczyn: 12000000
Różnica: 850
Średnia: 30
```

Importowanie pojedynczych funkcji z wieloplikowego modułu

Nie musisz importować wszystkich funkcji z modułu, nawet jeśli składa się on z wielu plików. W listingu 4.49 zmieniam instrukcję `import`, dzięki czemu mogę zaimportować tylko funkcje, z których rzeczywiście korzystam. Dodatkowo zastosowałem słowo kluczowe `as`, aby raz jeszcze pokazać możliwość zmiany nazwy pliku.

Listing 4.49. Importowanie wybranych funkcji z modułu wieloplikowego (*src/main.js*)

```
import { addition as add, multiply, subtract, mean as average } from "./maths";
let values = [10, 20, 30, 40, 50];
console.log(`Suma: ${add(values)})`);
console.log(`Iloczyn: ${multiply(values)})`);
console.log(`Różnica: ${subtract(1000, values)})`);
console.log(`Średnia: ${average(values)})`);
```

W ten sposób połączymy funkcje z poprzednich listingów, wybierając cztery funkcje z modułu i zmieniając nazwy dwóch z nich. Oto efekt:

```
Suma: 150
Iloczyn: 12000000
Różnica: 850
Średnia: 30
```

Zasady działania obietnic

Obietnica (ang. *promise*) to aktywność działająca w tle, która z założenia zostanie zakończona w przyszłości. Typowym przykładem użycia obietnic jest wykonywanie w sposób asynchroniczny (nieblokujący) żądań HTTP, które otrzymują wynik w momencie pobrania odpowiedzi z serwera WWW.

Problemy z asynchronicznym wykonywaniem operacji

Żądanie HTTP jest jednym z najpopularniejszych przykładów użycia asynchronicznych operacji, gdy konieczne jest pobranie danych wymaganych przez użytkownika. Niebawem przejdę do obsługi żądania HTTP, jednak zaczniemy od czegoś prostszego — w listingu 4.50 dodaję do pliku *maths/index.js* funkcję, która działa w sposób asynchroniczny.

Listing 4.50. Dodawanie funkcji do pliku *src/maths/index.js*

```
import addition from "./sum";
export function mean(values) {
    return addition(values)/values.length;
}
export { addition };
export * from "./operations";
export function asyncAdd(values) {
    setTimeout(() => {
        let total = addition(values);
        console.log(`Suma asynchroniczna: ${total}`);
        return total;
    }, 500);
}
```

Funkcja `setTimeout` wykonuje funkcję asynchronicznie, po upływie określonego interwału. W listingu funkcja `asyncAdd` otrzymuje parametr, który dalej zostaje przekazany do funkcji `addition` (ale dopiero po upływie 500 milisekund), tworząc operację w tle, która nie zostanie zakończona od razu. Jest to oczywiście przykład, który ma za zadanie reprezentować bardziej praktyczne operacje, takie jak realizacja żądań HTTP. W listingu 4.51 zmodyfikowałem plik *main.js*, pokazując przykład użycia funkcji `asyncAdd`.

Listing 4.51. Wykonywanie operacji w tle w pliku *src/main.js*

```
import { asyncAdd } from "./maths";
let values = [10, 20, 30, 40, 50];
let total = asyncAdd(values);
console.log(`Suma łączna: ${total}`);
```

Problem w powyższym przykładzie polega na tym, że wynik funkcji `asyncAdd` nie zostanie wygenerowany przed zakończeniem działania instrukcji w pliku *main.js*. W związku z tym efekt działania aplikacji będzie następujący:

```
Suma łączna: undefined
Suma asynchroniczna: 150
```

Przeglądarka wykonuje instrukcje w pliku *main.js*, co powoduje wywołanie funkcji `asyncAdd`. Następnie przeglądarka przechodzi do kolejnej instrukcji w pliku *main.js*, która zapisuje komunikat do konsoli, korzystając z wartości dostarczonej przez `asyncAdd`. Dzieje się to niestety przed faktycznym zakończeniem działania zadania asynchronicznego, przez co otrzymujemy w wyniku wartość `undefined`. Zadanie asynchroniczne po pewnym czasie się kończy, ale jest to zdecydowanie za późno dla pliku *main.js*.

Przykład z użyciem obietnic

Rozwiążanie przedstawionego problemu wymaga skorzystania z mechanizmu, który pozwoli obserwować zadanie asynchroniczne. Dzięki temu będziemy w stanie poczekać z wyświetleniem wyniku do momentu zakończenia zadania. Taką rolę w JavaScriptie pełni obietnica, z której korzystam w listingu 4.52.

Listing 4.52. Zastosowanie obietnicy w pliku *src/maths/index.js*

```
import addition from "./sum";
export function mean(values) {
    return addition(values)/values.length;
}
export { addition };
export * from "./operations";
export function asyncAdd(values) {
    return new Promise((callback) => {
        setTimeout(() => {
            let total = addition(values); console.log(`Suma asynchroniczna: ${total}`);
            callback(total);
        }, 500);
    });
}
```

Powyższy przykład nie należy do najprostszych. Słowo kluczowe `new` służy do utworzenia obiektu `Promise`, który przyjmuje funkcję przeznaczoną do obserwowania. Funkcja otrzymuje w formie parametru wywołanie zwrotne (ang. *callback*). Wywołanie to zostanie wykonane po zakończeniu zadania asynchronicznego. Wywołanie przyjmuje jako argument wynik działania zadania. Wykonanie wywołania zwrotnego nazywa się często rozwiązaniem (lub spełnieniem) obietnicy.

Obiekt typu `Promise`, zwrócony przez funkcję `asyncAdd`, pozwala na obserwowanie zadania asynchronicznego, dzięki czemu możemy zareagować na jego zakończenie (listing 4.53).

Listing 4.53. Obserwowanie obietnicy (*src/main.js*)

```
import { asyncAdd } from "./maths";
let values = [10, 20, 30, 40, 50];
asyncAdd(values).then(total => console.log(`Suma łączna: ${total}`));
```

Metoda `then` przyjmuje funkcję, która zostanie uruchomiona po wykonaniu wywołania zwrotnego. Wynik przekazany do wywołania zwrotnego trafia do funkcji `then`. Oznacza to, że suma nie zostanie wypisana w konsoli przeglądarki do momentu zakończenia zadania asynchronicznego. Efekt będzie następujący:

```
Suma asynchroniczna: 150
Suma łączna: 150
```

Uproszczenie kodu asynchronicznego

JavaScript oferuje dwa słowa kluczowe — `async` i `await` — które pozwalają na obsługę operacji asynchronicznych bez jawnego korzystania z obietnic. Stosuj je w listingu 4.54.

Listing 4.54. Zastosowanie słów kluczowych `await` i `async` w pliku *src/main.js*

```
import { asyncAdd } from "./maths";
let values = [10, 20, 30, 40, 50];
async function doTask() {
    let total = await asyncAdd(values);
    console.log(`Suma łączna: ${total}`);
}
doTask();
```

-
- **Ostrzeżenie** Warto pamiętać, że użycie słów `async` i `await` nie zmienia sposobu działania aplikacji. Operacja wciąż jest wykonywana asynchronicznie, a wynik nie jest dostępny do momentu jej zakończenia. Słowa kluczowe upraszczają pracę z asynchronicznym kodem, dzięki czemu nie musimy stosować metody `then`.
-

Wspomniane słowa kluczowe można stosować jedynie w odniesieniu do funkcji, stąd obecność funkcji `doTask` w listingu. Słowo `async` oznacza, że dana funkcja w swoim działaniu wykorzystuje obietnice. Słowo `await` jest używane, gdy wywołujemy funkcję zwracającą obietnicę. Dzięki słowu `await` wynik działania obietnicy zostanie zwrócony i przypisany do zmiennej `total`. Dopiero wtedy zostaną wyświetcone kolejne instrukcje. Oto efekt:

```
Suma asynchroniczna: 150
```

```
Suma łączna: 150
```

Podsumowanie

W tym rozdziale omówiliśmy podstawowe zagadnienia języka JavaScript, skupiając się na jego zasadniczych mechanizmach, które ułatwią Ci pracę z `Vue.js`. W kolejnym rozdziale przejdziemy do tworzenia bardziej zaawansowanego i bliższego rzeczywistości projektu o nazwie *Sklep sportowy*.

ROZDZIAŁ 5.

Sklep sportowy: prawdziwa aplikacja

W większości rozdziałów w tej książce zajmujemy się niewielkimi przykładami, skoncentrowanymi na poszczególnych funkcjach Vue.js. W ten sposób można łatwo poznać zasady działania poszczególnych części Vue.js, ale czasami brakuje szerszego kontekstu i trudno jest połączyć funkcje poznane w jednym rozdziale z funkcjami z innego. W związku z tym w tym i kolejnych rozdziałach zajmiemy się utworzeniem bardziej złożonej aplikacji.

Moja aplikacja *Sklep sportowy* (*SportsStore*) spełni założenia typowego sklepu internetowego. Utworzę internetowy katalog produktów, który będzie można przeglądać według kategorii i strony. Oprócz tego w sklepie znajdzie się koszyk, do którego użytkownicy będą dodawać lub z którego będą usuwać produkty, a także widok rozliczenia (*checkout*), gdzie klient wprowadzi adres dostawy i potwierdzi zamówienie. Oprócz tego sklep będzie zawierać obszar administracyjny, który pozwoli na zarządzanie katalogiem produktów — oczywiście tylko dla administratorów. Na zakończenie dowiesz się, jak przygotować aplikację do wdrożenia.

Celem tego rozdziału jest przedstawienie procesu tworzenia aplikacji w Vue.js jak najbliższego tworzeniu rzeczywistych aplikacji. Chcę się skupić na Vue.js, rzecz jasna, dlatego upraszczam kwestie związane z integracją z zewnętrznymi systemami takimi jak warstwa serwerowa czy obsługa płatności.

Przykład sklepu sportowego pojawia się w różnych moich książkach, ponieważ chcę w ten sposób pokazać, jak w różnych językach i frameworkach można osiągnąć ten sam efekt. Nie musisz znać moich innych książek, aby zrozumieć ten rozdział, ale jeśli je masz, warto zestawić różnice np. z książką *Pro ASP.NET Core MVC 2* lub *Pro Angular*.

Funkcje Vue.js, z których skorzystam w sklepie sportowym, są omówione w kolejnych rozdziałach. Zamiast powtarzać wszystko w tym miejscu, przekażę Ci tyle, abyś zrozumiał ogólne założenia aplikacji. W kolejnych rozdziałach dowiesz się wszystkiego w szczegółach.

-
- **Ostrzeżenie** Nie oczekuj, że wszystko zrozumiesz od razu — Vue.js składa się z wielu rozmaitych części, a aplikacja *Sklep sportowy* ma pokazać Ci, jak współpracują one ze sobą, bez konieczności zagłębiania się w szczegóły już teraz. Jeśli dany temat Cię nurtuje, zapoznaj się z drugą częścią tej książki w wybranym zakresie i powróć do lektury tego rozdziału.
-

Tworzenie projektu Sklep sportowy

Na początku musimy oczywiście stworzyć projekt. Otwórz terminal (wiersz poleceń), przejdź do wybranej lokalizacji i wykonaj polecenie z listingu 5.1.

Listing 5.1. Tworzenie projektu Sklep sportowy

```
vue create sportsstore --default
```

- **Uwaga** W trakcie pisania niniejszej książki pakiet @vue/cli był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratą dostępną pod adresem <https://github.com/Apress/pro-vue-js-2>.

Projekt zostanie utworzony, a pakiety do wszystkich narzędzi deweloperskich zostaną pobrane i zainstalowane, co może zajść trochę czasu.

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.
-

Dodawanie dodatkowych pakietów

Vue.js opiera się na funkcjach, które są dostarczane w formie pakietów opcjonalnych — niektóre z nich są rozwijane przez główny zespół Vue.js, a inne są rozwijane przez zewnętrzne zespoły. Większość pakietów wymaganych do tworzenia aplikacji w Vue.js jest dodawana standardowo, ale w przypadku tego projektu musimy dodać kilka pakietów sami. Wykonaj polecenia z listingu 5.2, aby przejść do katalogu *sportsstore* i dodać niezbędne pakiety (narzędzie npm może zainstalować wiele pakietów jednym poleceniem, ale tutaj postanowiłem wyodrębnić każdy pakiet, aby nazwy i wersje były bardziej czytelne).

Listing 5.2. Dodawanie pakietów

```
cd sportsstore
npm install axios@0.18.0
npm install vue-router@3.0.1
npm install vuex@3.0.1
npm install vuelidate@0.7.4
npm install bootstrap@4.0.0
npm install font-awesome@4.7.0
npm install --save-dev json-server@0.12.1
npm install --save-dev jsonwebtoken@8.1.1
npm install --save-dev faker@4.1.0
```

Wykonując polecenia, należy pamiętać o podaniu numerów wersji. W trakcie instalacji mogą pojawić się komunikaty o niespełnionych zależnościach, ale możesz je zignorować. Znaczenie każdego pakietu w naszym projekcie objaśniono w tabeli 5.1. Niektóre pakiety są instalowane z argumentem `--save-dev`, co oznacza, że są instalowane wyłącznie na etapie tworzenia aplikacji i nie będą częścią jej produkcyjnej wersji.

- **Uwaga** Pakiety `vue-router` i `vuex` mogą być instalowane automatycznie jako elementy szablonu, jednak dodaj je osobno, aby pokazać, w jaki sposób można dołączyć je samodzielne. Nie ma nic złego w korzystaniu z narzędzi projektowych w celu inicjalizacji projektu, ale warto wiedzieć, jak działa projekt w Vue.js, zwłaszcza gdy coś zaczyna funkcjonować nieprawidłowo.

Tabela 5.1. Dodatkowe pakiety wymagane w projekcie Sklep sportowy

Nazwa	Opis
axios	Pakiet Axios służy do wykonywania żądań HTTP do usług sieciowych, które dostarczą naszej aplikacji danych. Axios działa nie tylko z Vue.js — jest to częsty wybór do obsługi protokołu HTTP w wielu frameworkach. Szczegółowy opis Axiosa znajdziesz w rozdziale 19.
vue-router	Ten pakiet pozwala na uzależnienie zawartości widoku od aktualnego adresu URL przeglądarki.
vuex	Ten pakiet jest używany do tworzenia współdzielonego magazynu danych, który ułatwia zarządzanie danymi w projekcie Vue.js. Magazyny Vuex szczegółowo opisuję w rozdziale 20.
vuelidate	Ten pakiet pomaga walidować dane, które użytkownicy wprowadzają w elementach formularza — więcej na ten temat w rozdziale 6.
bootstrap	Pakiet Bootstrap zawiera style CSS, które zostaną użyte do ostylowania dokumentu HTML prezentowanego użytkownikowi.
font-awesome	Pakiet Font Awesome zawiera bibliotekę ikon, z których skorzystamy w sklepie do oznaczenia funkcji ważnych dla użytkownika.
json-server	Ten pakiet udostępnia łatwe w użyciu usługi sieciowe zgodne ze stylem architektury REST. To właśnie wspomniany pakiet otrzyma żądania HTTP wykonane za pomocą Axiosa. Szerzej omawiam ten pakiet w punkcie „Przygotowanie REST-owej usługi sieciowej”.
jsonwebtoken	Ten pakiet jest używany do generowania tokenów autoryzacyjnych, dzięki którym będzie możliwy dostęp do administracyjnych funkcji sklepu (więcej w rozdziale 7.).
faker	Ten pakiet jest używany do generowania testowych danych w celu sprawdzenia możliwości obsługi dużej ilości danych (więcej w rozdziale 7.).

Dołączanie arkuszy stylów CSS do aplikacji

Pakiety Bootstrap i Font Awesome wymagają dodania instrukcji import do pliku *main.js*, w którym znajduje się nadrzędna konfiguracji aplikacji Vue.js. Instrukcje import, pokazane w listingu 5.3, zapewniają, że zawartość tych pakietów zostanie dołączona do aplikacji przez narzędzia deweloperskie Vue.js.

Listing 5.3. Dołączanie pakietów do pliku *src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
Vue.config.productionTip = false
import "bootstrap/dist/css/bootstrap.min.css";
import "font-awesome/css/font-awesome.min.css"
new Vue({
  render: h => h(App)
}).$mount('#app')
```

■ **Wskazówka** Nie martw się pozostałymi instrukcjami w pliku *main.js*. Ich zadaniem jest inicjalizacja aplikacji Vue.js. Omówię je szczegółowo w rozdziale 9. — wiedza na ich temat nie jest niezbędna do tego, aby zacząć przygodę z Vue.js.

Te instrukcje pozwolą mi wykorzystywać możliwości CSS oferowane przez dodane pakiety w całej aplikacji.

Przygotowanie REST-owej usługi sieciowej

Aplikacja *Sklep sportowy* skorzysta z asynchronicznych żądań HTTP w celu pobrania danych udostępnionych przez REST-ową usługę sieciową. Jak objaśniam w rozdziale 19., **REST** (ang. *REpresentational State Transfer* — transfer stanu za pomocą reprezentacji) to podejście do projektowania usług sieciowych, które zakłada zastosowanie metody (czasownika — ang. *verb*) protokołu HTTP razem z adresem URL w celu ustalenia obiektów danych, których dotyczy dana operacja.

Pakiet json-server, który dodałem przed chwilą do projektu, stanowi świetne narzędzie do szybkiego generowania usług sieciowych na podstawie danych w formacie JSON lub innej postaci kodu JavaScript. Aby upewnić się, że istnieje pewien stan, do którego da się przywrócić projekt, zamierzam skorzystać z funkcji, która pozwala na utworzenie REST-owej usługi sieciowej na podstawie kodu JavaScript. Zrestartowanie usługi sieciowej spowoduje więc przywrócenie zestawu danych do stanu początkowego. Utworzyłem plik *data.js* w katalogu *sportsstore*, a następnie dodałem kod z listingu 5.4.

Listing 5.4. Zawartość pliku *sportsstore/data.js*

```
var data = [{ id: 1, name: "Kajak", category: "Sporty wodne",
    description: "Kajak dla jednej osoby", price: 275 },
{ id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
    description: "Skuteczna i modna", price: 48.95 },
{ id: 3, name: "Piłka nożna", category: "Piłka nożna",
    description: "Zakceptowana przez federację pod względem wagi i wymiarów",
    price: 19.50 },
{ id: 4, name: "Chorągiewki do rogów boiska", category: "Piłka nożna",
    description: "Odmień swoje boisko, nadając mu profesjonalny sznyt", price: 34.95 },
{ id: 5, name: "Stadion", category: "Piłka nożna",
    description: "Stadion na 35 000 miejsc", price: 79500 },
{ id: 6, name: "Myśląca czapeczka", category: "Szachy",
    description: "Zwiększa wydajność mózgu o 75%", price: 16 },
{ id: 7, name: "Chwiejne krzesło", category: "Szachy",
    description: "Potajemnie doprowadź przeciwnika do irytacji", price: 29.95 },
{ id: 8, name: "Szachownica", category: "Szachy",
    description: "Zabawa dla całej rodziny", price: 75 },
{ id: 9, name: "Król(u) złoty", category: "Szachy",
    description: "Pozłacana, zdobiona diamentami figura króla", price: 1200 }]

module.exports = function () {
    return {
        products: data,
        categories: [...new Set(data.map(p => p.category))].sort(),
        orders: []
    }
}
```

Powyższy plik jest modelem JavaScript, który eksportuje funkcję domyślną. Oprócz tego w pliku znajdują się dwie kolekcje, które będą udostępniane przez REST-ową usługę sieciową. Kolekcja *products* zawiera produkty podlegające wyprzedaży, a kolekcja *categories* zawiera nazwy kategorii, pobrane na bazie wartości właściwości *category*. Kolekcja *orders* będzie zawierać zamówienia złożone przez klientów (obecnie jest ona pusta).

Dane przechowywane w REST-owej usłudze sieciowej muszą być chronione, aby zwykły użytkownik nie był w stanie zmieniać danych o produktach lub stanu zamówień. Pakiet json-server nie zawiera żadnych mechanizmów uwierzytelniania, dlatego do pliku *sportsstore/authMiddleware.js* dodaję kod przedstawiony w listingu 5.5.

Listing 5.5. Zawartość pliku sportsstore/authMiddleware.js

```
const jwt = require("jsonwebtoken");
const APP_SECRET = "myappsecret";
const USERNAME = "admin";
const PASSWORD = "secret";
module.exports = function (req, res, next) {
  if ((req.url == "/api/login" || req.url == "/login")
    && req.method == "POST") {
    if (req.body != null && req.body.name == USERNAME
      && req.body.password == PASSWORD) {
      let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
      res.json({ success: true, token: token });
    } else {
      res.json({ success: false });
    }
    res.end();
    return;
  } else if (((req.url.startsWith("/api/products")
    || req.url.startsWith("/products"))
    || (req.url.startsWith("/api/categories")
      || req.url.startsWith("/categories")))) && req.method != "GET")
  || ((req.url.startsWith("/api/orders")
    || req.url.startsWith("/orders")) && req.method != "POST")) {
    let token = req.headers["authorization"];
    if (token != null && token.startsWith("Bearer<")) {
      token = token.substring(7, token.length - 1);
      try {
        jwt.verify(token, APP_SECRET);
        next();
        return;
      } catch (err) { }
    }
    res.statusCode = 401;
    res.end();
    return;
  }
  next();
}
```

Powyższy kod analizuje żądania HTTP przesłane do REST-owej usługi sieciowej i implementuje podstawowe mechanizmy bezpieczeństwa. Jest to kod działający po stronie serwera, niezwiązany bezpośrednio z Vue.js, dlatego nie martw się, jeżeli nie do końca go rozumiesz. Proces uwierzytelniania i autoryzacji wyjaśniam w rozdziale 7.

- **Ostrzeżenie** Nie korzystaj z kodu zawartego w listingu 5.5 do innych celów niż aplikacja *Sklep sportowy*. Kod ten zawiera słabe hasła, osadzone w nim bezpośrednio. W tym projekcie nie jest to problem — jest on bowiem jedynie ćwiczeniowy, jednak w rzeczywistych projektach takich rozwiązań należy unikać.

Musimy zmienić również zawartość pliku *package.json*, aby móc uruchamiać pakiet json-server z wiersza poleceń (listing 5.6).

Listing 5.6. Dodawanie skryptu do pliku package.json w katalogu sportsstore

```
{
  "name": "sportsstore",
  "version": "0.1.0",
```

```

"private": true,
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "lint": "vue-cli-service lint",
  "json": "json-server data.js -p 3500 -m authMiddleware.js"
},
"dependencies": {
  "axios": "^0.18.0",
  "bootstrap": "^4.0.0",
  "font-awesome": "^4.7.0",
  "vue": "^2.5.16",
  "vue-router": "^3.0.1",
  "vuex": "^3.0.1"
},
//...pozostałe ustawienia zostały pominięte w listingu...
}

```

Plik *package.json* jest używany do konfigurowania projektu i związków z nim narzędzi. Sekcja *scripts* zawiera polecenia, które mogą być wykonywane za pomocą pakietów dodanych do projektu.

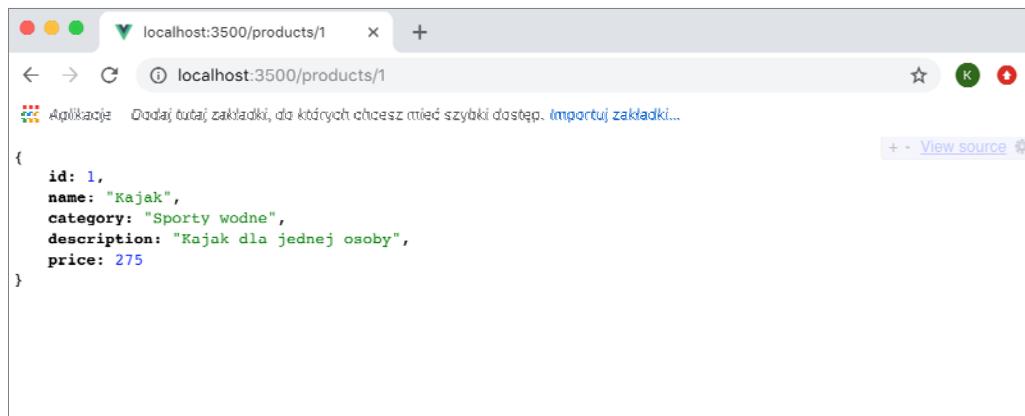
Uruchamianie narzędzi projektowych

Po zakończeniu konfiguracji projektu możemy uruchomić narzędzia deweloperskie i upewnić się, że wszystko działa, jak należy. Otwórz terminal (wiersz poleceń), przejdź do katalogu *sportsstore* i wykonaj polecenie z listingu 5.7, aby uruchomić usługę sieciową.

Listing 5.7. Uruchamianie usługi sieciowej Sklep sportowy

```
npm run json
```

Otwórz okno przeglądarki i przejdź pod adres <http://localhost:3500/products/1>, aby sprawdzić, czy usługa sieciowa działa prawidłowo (efekt powinien być zbliżony do tego z rysunku 5.1).



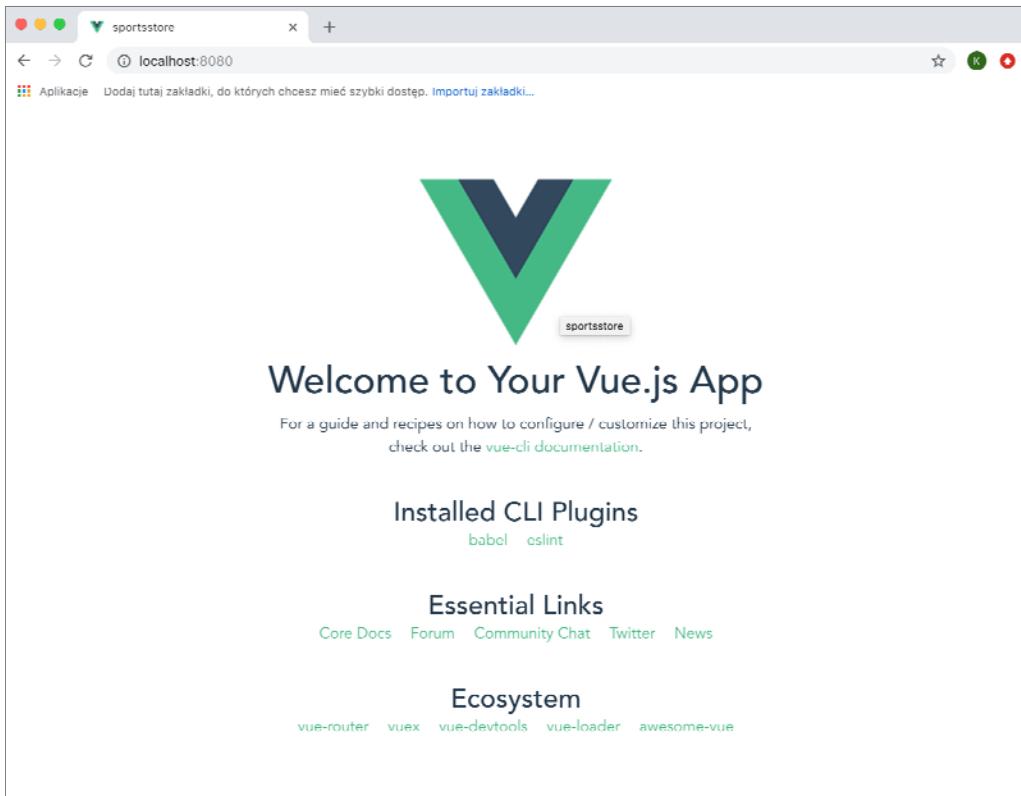
Rysunek 5.1. Testowanie usługi sieciowej

Nie zatrzymując usługi sieciowej, otwórz drugi wiersz poleceń, przejdź do katalogu *sportsstore* i wykonaj polecenie z listingu 5.8, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 5.8. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

W tym momencie zostanie uruchomiony serwer deweloperski HTTP, nastąpi typowy proces inicjalizacji, aż wreszcie otrzymasz komunikat świadczący o uruchomieniu aplikacji. Przejdz na stronę `http://localhost:8080` i upewnij się, że widzisz efekt taki jak na rysunku 5.2, czyli typową treść zastępczą dostępną tuż po utworzeniu projektu.



Rysunek 5.2. Uruchamianie aplikacji

- **Wskazówka** Port 8080 jest portem domyślnym. Jeśli jednak uruchomienie serwera na tym porcie nie będzie możliwe, narzędzia Vue.js wybiorą inny port. Jeśli tak się stanie, możesz zatrzymać proces, który pierwotnie zajął port 8080, lub przejść pod inny adres URL, podany w konsoli po uruchomieniu projektu Vue.js.

Tworzenie magazynu danych

Najlepszym obszarem do rozpoczęcia prac nad nowym projektem jest obsługa danych. We wszystkich projektach (może poza tymi najprostszymi) do tworzenia magazynu danych warto stosować pakiet Vuex.

Magazyn danych (ang. *data store*) pozwala na współdzielanie danych w całej aplikacji, udostępniając repozytorium i zapewniając, że wszystkie elementy aplikacji korzystają z tych samych, aktualnych danych.

- **Wskazówka** Vuex nie jest jedynym pakietem, który można zastosować do zarządzania danymi w aplikacji Vue.js. Został on jednak opracowany przez zespół Vue.js i jest bardzo dobrze zintegrowany z pozostałymi komponentami ze świata Vue.js. Poza bardzo specyficznymi sytuacjami warto stosować w projektach Vue.js pakiet Vuex.

Magazyny danych Vuex definiuje się z reguły jako odrębne moduły JavaScript, we własnym katalogu. W związku z tym utworzyłem katalog *src/store* (zgodnie z popularną konwencją) i dodałem plik *index.js* o treści przedstawionej w listingu 5.9.

Listing 5.9. Zawartość pliku src/store/index.js

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
const testData = [];
for (let i = 1; i <= 10; i++) {
  testData.push({
    id: i, name: `Produkt ${i}`, category: `Kategoria ${i % 3}`,
    description: `To jest Produkt ${i}`, price: i * 50
  })
}
export default new Vuex.Store({
  strict: true,
  state: {
    products: testData
  }
})
```

Instrukcje import deklarują zależności od bibliotek Vue.js i Vuex. Vuex jest rozpowszechniany jako wtyczka technologii Vue.js, dzięki czemu można łatwo skorzystać z niego w całej aplikacji. Zasady działania wtyczek objaśniam w rozdziale 26. Na razie musisz pamiętać, że każda wtyczka musi być włączona za pomocą wywołania metody *Vue.use*. Jeśli zapomnisz skorzystać z tej metody, funkcje magazynu danych nie będą dostępne w pozostałojej części aplikacji.

Magazyn danych tworzymy za pomocą słowa kluczowego *new* w odniesieniu do typu *Vuex.Store*. W tym wywołaniu przekazujemy obiekt konfiguracji składający się (w tym przypadku) z dwóch właściwości. Właściwość *state* służy do określenia danych przechowywanych w magazynie. Aby uruchomić magazyn danych, skorzystałem z pętli *for* w celu wygenerowania danych testowych (przypisanych do właściwości *products*). W dalszej części rozdziału (w podrozdziale „Zastosowanie REST-owej usługi sieciowej”) zamienię te dane na informacje pobrane z usługi sieciowej.

Właściwość *strict* nie jest tak oczywista i ma związek z nietypowym sposobem działania pakietu Vuex. Wartości danych są przeznaczone tylko do odczytu. Ich zmiana jest możliwa jedynie za pomocą mutacji (ang. *mutation*). Mutacja jest metodą języka JavaScript, która zmienia dane. Z przykładami mutacji spotkasz się niebawem, gdy zaczynę dodawać funkcje do aplikacji *Sklep sportowy*. Zastosowanie właściwości *strict* pozwoli na wyświetlenie ostrzeżenia, gdy zapomnisz skorzystać z mutacji, zamiast tego zmieniając dane bezpośrednio — to będzie się zdarzać, gdy zaczniesz więcej pracować z Vuex.

Dodanie magazynu danych do aplikacji wymaga dodania instrukcji z listingu 5.10 do pliku *main.js*, który stanowi nasz centralny punkt konfiguracji aplikacji.

Listing 5.10. Dodawanie magazynu danych do pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
Vue.config.productionTip = false
```

```
import "bootstrap/dist/css/bootstrap.min.css";
import "font-awesome/css/font-awesome.min.css"
import store from "./store";
new Vue({
  render: h => h(App),
  store
}).$mount('#app')
```

Instrukcja `import` deklaruje zależność od modułu magazynu danych i przypisuje go do identyfikatora `store`. Dodanie właściwości `store` do obiektu konfiguracji da możliwość zastosowania funkcji magazynu danych w całej aplikacji, o czym przekonasz się w miarę dodawania funkcji do projektu naszego sklepu.

- **Ostrzeżenie** Typowym błędem jest dodanie instrukcji `import` przy jednoczesnym pominięciu właściwości `store` w obiekcie konfiguracji. W ten sposób powstanie błędów, ponieważ funkcje magazynu danych nie zostaną uwzględnione w aplikacji.

Tworzenie magazynu produktów

Magazyn danych dostarcza wystarczająco dużą część infrastruktury całej aplikacji, aby rozpocząć od najważniejszej, z punktu widzenia użytkownika, funkcji — magazynu produktów. Wszystkie sklepy internetowe udostępniają użytkownikowi listę produktów do kupienia — u nas nie może być inaczej. Podstawowa struktura sklepu zawiera układ dwukolumnowy, w którym przyciski kategorii pozwolą na przefiltrowanie listy, a tabela przedstawi listę wszystkich produktów (rysunek 5.3).



Rysunek 5.3. Podstawowa struktura naszego sklepu

Podstawowym składnikiem aplikacji Vue.js jest komponent. Komponenty są definiowane w plikach z rozszerzeniem `.vue`. Rozpoczniemy od utworzenia pliku `Store.vue` w katalogu `src/components` (listing 5.11).

Listing 5.11. Zawartość pliku `src/components/Store.vue`

```
<template>
<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SKLEP SPORTOWY</a>
    </div>
  </div>
  <div class="row">
    <div class="col-3 bg-info p-2">
      <h4 class="text-white m-2">Kategorie</h4>
    </div>
    <div class="col-9 bg-success p-2">
```

```
<h4 class="text-white m-2">Produkty</h4>
</div>
</div>
</div>
</template>
```

Ten komponent zawiera tylko element template, z którego skorzystałem, by zdefiniować podstawowy układ aplikacji. W tym celu zdefiniowałem kilka elementów HTML z dołączonymi klasami Bootstrapa (opisanego w rozdziale 3.). Na razie w tym szablonie nie ma nic specjalnego, jednak odzwierciedla on układ przedstawiony na rysunku 5.3 i stanowi podstawę do dalszych prac. W listingu 5.12 zamieniam zawartość pliku *App.vue*, dzięki czemu jestem w stanie zamienić treść wygenerowaną w momencie tworzenia projektu na komponent magazynu utworzony w listingu 5.11.

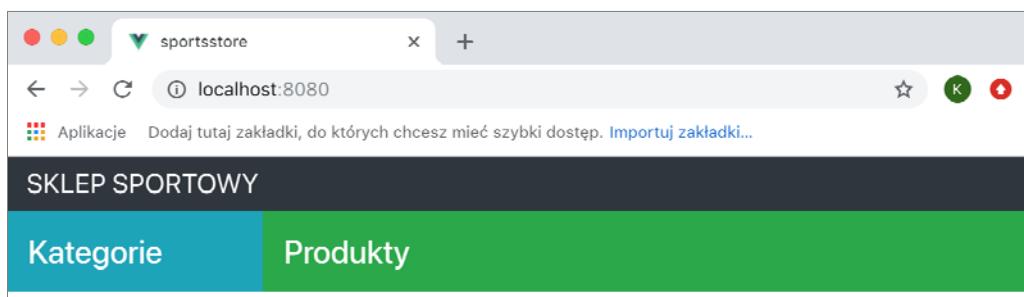
Listing 5.12. Zamiana zawartości pliku *src/App.vue*

```
<template>
  <store />
</template>
<script>
import Store from './components/Store';
export default {
  name: 'app',
  components: { Store }
}
</script>
```

Aplikacja Vue.js z reguły zawiera szereg komponentów. W większości projektów komponent App (zdefiniowany w pliku *App.vue*) odpowiada za wybór komponentu, który w danej chwili zostanie przedstawiony użytkownikowi. Ten mechanizm zostanie pokazany, gdy dodam do projektu koszyk i ekran podsumowania zamówienia w rozdziale 6. Na razie komponent Store jest jedynym, jaki utworzyliśmy, dlatego tylko on może być pokazany użytkownikowi.

Instrukcja import w elemencie script wprowadza zależność od komponentu z listingu 5.11 i przypisuje ją do identyfikatora Store. Następnie obiekt trafia do właściwości components, informując Vue.js o zastosowaniu komponentu Store w ramach komponentu App.

Po przetworzeniu szablonu komponentu App element HTML o nazwie store (z listingu 5.12) zostanie zastąpiony treścią szablonu z listingu 5.11, co da efekt jak na rysunku 5.4.



Rysunek 5.4. Dodawanie własnego komponentu do aplikacji

Tworzenie listy produktów

Kolejnym krokiem jest utworzenie komponentu, który wyświetli listę produktów użytkownikowi. Do katalogu *src/components* dodaję plik *ProductList.vue* o treści przedstawionej w listingu 5.13.

Listing 5.13. Zawartość pliku *src/components/ProductList.vue*

```
<template>
  <div>
    <div v-for="p in products" v-bind:key="p.id" class="card m-1 p-1 bg-light">
      <h4>
        {{p.name}}
        <span class="badge badge-pill badge-primary float-right">
          {{ p.price }}
        </span>
      </h4>
      <div class="card-text bg-white p-1">{{ p.description }}</div>
    </div>
  </div>
</template>
<script>
import { mapState } from "vuex";
export default {
  computed: {
    ...mapState(["products"])
  }
}
</script>
```

Element `script` importuje z pakietu vuex funkcję `mapState`, która pozwala na dostęp do danych w magazynie. Różne operacje można wykonać za pomocą różnych funkcji Vuex — `mapState` służy do utworzenia powiązania pomiędzy komponentem a danymi w magazynie. Funkcja `mapState` jest używana z operatorem rozwinięcia, ponieważ w ten sposób możemy powiązać wiele właściwości magazynu danych w jednej operacji (choć w tym przykładzie skorzystamy jedynie z właściwości stanu `products`). Właściwości stanu magazynu danych (ang. *data store state properties*) są dołączane jako właściwości obliczane (`computed`) komponentu — to zagadnienie omawiam w rozdziale 11.

Vue.js korzysta z **dyrektyw** (ang. *directives*) w celu modyfikowania elementów języka HTML. W tym listingu używam dyrektywy `v-for`, która duplikuje element i jego zawartość dla każdego elementu z tablicy.

```
...
<div v-for="p in products" v-bind:key="p.id" class="card m-1 p-1 bg-light">
  ...

```

Efektem działania funkcji `mapState` jest możliwość zastosowania właściwości `products` w dyrektywie `v-for` w celu uzyskania dostępu do danych w magazynie. Możemy mieć pewność, że dla każdego produktu zostanie zastosowany ten sam zbiór elementów HTML. Każdy produkt jest przypisywany tymczasowo do zmiennej `p`, z której skorzystam w celu dostosowania treści elementów generowanych dla każdego produktu, np. w ten sposób:

```
...
<div class="card-text bg-white p-1">{{ p.description }}</div>
...
```

Podwójne nawiasy klamrowe (`{{ i }}`) oznaczają wiązanie danych, dzięki któremu Vue.js jest w stanie podstawić wartość do elementu HTML w momencie renderowania widoku dla użytkownika. Wiązania danych zostały szczegółowo wyjaśnione w rozdziale 11., a dyrektywa `v-for` — w rozdziale 13. Powyższy kod spowoduje wstawienie do znacznika `div` wartości właściwości `description` aktualnie przetwarzanego produktu.

- **Wskazówka** Dyrektywa `v-for` znajduje się obok dyrektywy `v-bind`, która tworzy atrybut o wartości będącej wartością danych lub fragmentem kodu JavaScript. W tym przypadku tworzymy atrybut `key`, dzięki któremu dyrektywa `v-for` jest w stanie efektywnie reagować na zmiany w danych aplikacji (więcej na ten temat w rozdziale 13.).

Dodawanie listy produktów do aplikacji

Każdy komponent dodany do projektu musi zostać przed użyciem zarejestrowany. W listingu 5.14 rejestruję komponent ProductList w ramach komponentu Store, dzięki czemu mogę usunąć treść zastępczą i wstawić rzeczywistą listę produktów.

Listing 5.14. Rejestrowanie komponentu w pliku src/components/Store.vue

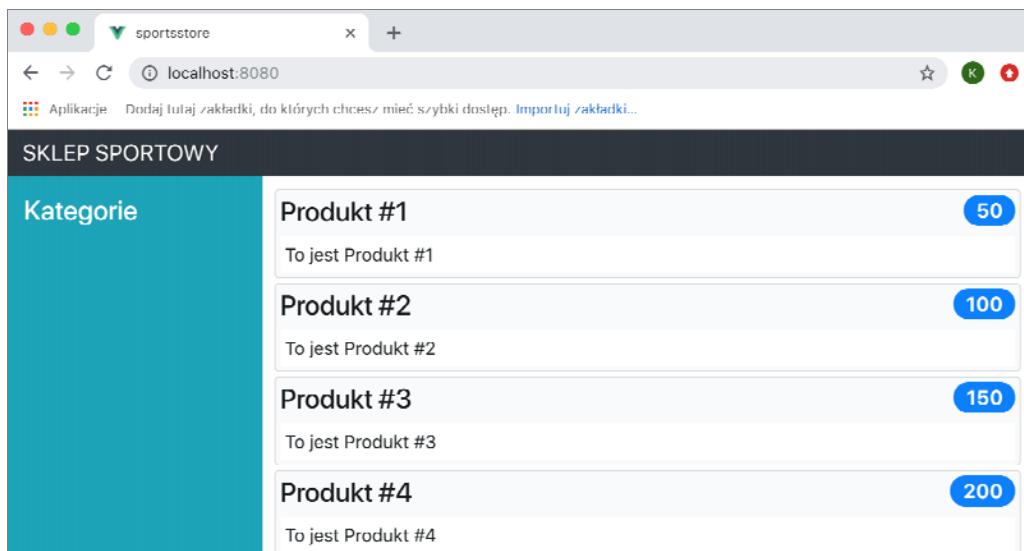
```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col bg-dark text-white">
        <a class="navbar-brand">SKLEP SPORTOWY </a>
      </div>
    </div>
    <div class="row">
      <div class="col-3 bg-info p-2">
        <h4 class="text-white m-2">Kategorie</h4>
      </div>
      <div class="col-9 p-2 ">
        <product-list />
      </div>
    </div>
  </div>
</template>
<script>
import ProductList from './ProductList';
export default {
  components: { ProductList }
}
</script>
```

Gdy komponenty są używane razem, tworzy się między nimi pewien związek. W tym przykładzie komponent Store jest rodzicem (ang. *parent*) komponentu ProductList. Można też powiedzieć, że komponent ProductList jest dzieckiem (ang. *child*) komponentu Store. W powyższym listingu do zarejestrowania komponentu skorzystałem z tego samego związku co w przypadku rejestracji komponentu Store w całej aplikacji: zimportowałem komponent-dziecko i dodałem go do właściwości `components` rodzica. Dzięki temu jestem w stanie dodać własny element HTML, aby wstawić komponent-dziecko w szablonie komponentu-rodzica. W listingu 5.14 wstawilem komponent `ProductList`, korzystając z elementu `product-list`. Vue.js automatycznie przetwarza taką wieloczęściową nazwę na odpowiednią nazwę komponentu (choć mogliby również nazwać znacznik `ProductList` lub `productList`).

W rezultacie komponent App wstawia zawartość komponentu Store, który z kolei wstawią zawartość komponentu ProductList, co daje efekt jak na rysunku 5.5.

Przetwarzanie cen

Po skonfigurowaniu podstawowej listy mogę zacząć dodawać kolejne funkcje do aplikacji. Na początku trzeba zająć się cenami. Obecnie ceny są wyświetlane jak zwykłe liczby, a nie jak kwoty pieniędzy. Komponenty Vue.js wprowadzają pojęcie **filtrów**, czyli funkcji, które pozwalają formatować wartości danych. W listingu 5.15 dodaję do komponentu ProductList filtr `currency`, który wyświetli wartość w formie kwot w złotówkach.



Rysunek 5.5. Wyświetlanie listy produktów

Listing 5.15. Definiowanie filtru w pliku src/components/ProductList.vue

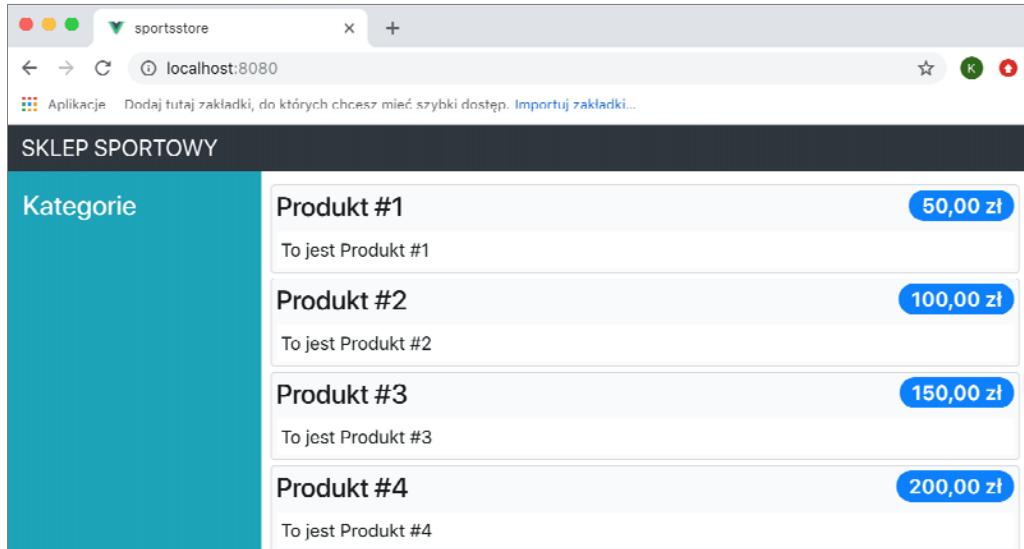
```
<template>
  <div>
    <div v-for="p in products" v-bind:key="p.id" class="card m-1 p-1 bg-light">
      <h4>
        {{p.name}}
        <span class="badge badge-pill badge-primary float-right">
          {{ p.price | currency }}
        </span>
      </h4>
      <div class="card-text bg-white p-1">{{ p.description }}</div>
    </div>
  </div>
</template>
<script>
import { mapState } from "vuex";
export default {
  computed: {
    ...mapState(["products"])
  },
  filters: {
    currency(value) {
      return new Intl.NumberFormat("pl-PL",
        { style: "currency", currency: "PLN" }).format(value);
    }
  }
}
</script>
```

Funkcje komponentu są zgrupowane za pomocą właściwości w obiekcie, zadeklarowanym w elemencie `script`. Komponent `ProductList` wprowadza dwie właściwości: `computed`, która daje dostęp do danych z magazynu, a także `filters`, w której definiujemy filtry. W tym przykładzie pojawia się jeden filtr, `currency`, który korzysta z funkcji lokalizacyjnych JavaScriptu, aby sformatować wartość liczbową jako kwotę w polskich złotych.

Aby skorzystać z filtru w szablonie, należy po nazwie wartości podać nazwę filtru, przedzielone znakiem pionowej kreski (|):

```
...
<span class="badge badge-pill badge-primary float-right">
  {{ p.price | currency }}
</span>
...
```

Po zapisaniu zmian w pliku *ProductList.vue* przeglądarka zostanie odświeżona, a ceny będą wyświetlane tak jak na rysunku 5.6.



Rysunek 5.6. Zastosowanie filtru do formatowania kwot pieniędzy

Obsługa stronicowania listy produktów

Produkty są wyświetlane w formie pojedynczej listy. W miarę zwiększania liczby dostępnych produktów korzystanie z niej może stać się dla użytkownika udręczą. Aby uczynić listę łatwiejszą w użyciu, dodam obsługę stronicowania. Na każdej stronie będzie wyświetlana ograniczona liczba produktów, a użytkownik będzie w stanie przechodzić pomiędzy kolejnymi stronami za pomocą mechanizmu nawigacji. Na początku musimy rozszerzyć magazyn danych, aby umieścić w nim rozmiar strony i numer aktualnie zaznaczonej strony (listing 5.16).

Listing 5.16. Przygotowanie do obsługi stronicowania w pliku src/store/index.js

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
const testData = [];
for (let i = 1; i <= 10; i++) {
  testData.push({
    id: i, name: `Produkt ${i}`, category: `Kategoria ${i % 3}`,
    description: `To jest Produkt ${i}`, price: i * 50
  })
}
```

```

export default new Vuex.Store({
  strict: true,
  state: {
    products: testData, productsTotal: testData.length,
    currentPage: 1,
    pageSize: 4
  },
  getters: {
    processedProducts: state => {
      let index = (state.currentPage - 1) * state.pageSize;
      return state.products.slice(index, index + state.pageSize);
    },
    pageCount: state => Math.ceil(state.productsTotal / state.pageSize)
  },
  mutations: {
    setCurrentPage(state, page) {
      state.currentPage = page;
    },
    setPageSize(state, size) {
      state.pageSize = size;
      state.currentPage = 1;
    }
  }
})

```

Nowe elementy w magazynie danych obsługują validację. Dzięki nim będziemy w stanie zaprezentować kluczowe mechanizmy oferowane przez Vuex. Pierwsze zmiany znajdują się we właściwości state. Definiuję w niej liczbę produktów, numer aktualnie wybranej strony i liczbę produktów do pokazania na jednej stronie.

Sekcja getters zawiera właściwości, które są obliczane za pomocą właściwości state. W listingu 5.16 sekcja getters wprowadza właściwość processedProducts, która zwraca wyłącznie produkty do wyświetlenia na bieżącej stronie, a także właściwość pageCount, która określa, ile jest możliwych do wyświetlenia stron produktów.

Sekcja mutations zawiera metody, które zmieniają wartości minimum jednej właściwości stanu. W tym listingu mamy do czynienia z dwoma mutacjami: setCurrentPage zmienia wartość właściwości currentPage, a setPageSize — właściwości pageSize.

Podział na standardowe i obliczane właściwości danych często pojawia się podczas tworzenia aplikacji Vue.js, ponieważ dzięki niemu możliwe jest efektywne wykrywanie zmian. Gdy zmiana ulega właściwości danych, Vue.js może określić jej wpływ na właściwość obliczaną i nie musi przeliczać wartości, o ile zmiana nie uległa bazowemu źródłu danych. Ten koncept jest rozwinięty jeszcze bardziej przez magazyny danych Vuex. Zmiana danych jest w nich możliwa jedynie w mutacjach — nie zaś przez proste przypisanie nowej wartości. Może to wydać się dziwne, gdy zaczynasz pracę z magazynami danych, jednak szybko przyzwyczaisz się do takiego sposobu pracy. Podążanie tą drogą pozwoli Ci skorzystać z ciekawych możliwości, takich jak śledzenie zmian czy cofanie lub ponowne wykonywanie operacji za pomocą wtyczki do przeglądarki o nazwie Vue Devtools, opisanej w rozdziale 1.

- **Wskazówka** Zwróć uwagę, że zarówno metody pobierające (getters), jak i mutacje z listingu 5.16 są zadeklarowane jako funkcje, które otrzymują obiekt state jako pierwszy parametr. Obiekt ten jest używany do pobrania wartości zdefiniowanych w sekcji state magazynu danych, która nie jest dostępna bezpośrednio. Więcej szczegółów i przykładów zawiera rozdział 20.

Skoro dodałem dane i mutacje do magazynu danych, mogę utworzyć komponent, który z nich korzysta. W związku z tym dodaję plik `src/components/PageControls.vue` (listing 5.17).

Listing 5.17. Zawartość pliku *src/components/PageControls.vue*

```
<template>
  <div v-if="pageCount > 1" class="text-right">
    <div class="btn-group mx-2">
      <button v-for="i in pageNumbers" v-bind:key="i"
              class="btn"
              v-bind:class="{ 'btn-primary': i == currentPage, 'btn-secondary': i != currentPage }">
        {{ i }}
      </button>
    </div>
  </div>
</template>
<script>
  import { mapState, mapGetters } from "vuex";
  export default {
    computed: {
      ...mapState(["currentPage"]),
      ...mapGetters(["pageCount"]),
      pageNumbers() {
        return [...Array(this.pageCount + 1).keys()].slice(1);
      }
    }
  }
</script>
```

Nie wszystkie funkcje stronicowania są już gotowe, ale z pewnością jest ich wystarczająco dużo, aby zacząć. Komponent korzysta z funkcji `mapState` i `mapGetters`, aby uzyskać dostęp do właściwości `currentPage` i `pageCount` magazynu danych. Nie wszystko musi być zdefiniowane w magazynie danych — komponent wprowadza funkcję `pageNumbers`, która korzysta z właściwości `pageCount` do wygenerowania sekwencji kolejnych liczb naturalnych. Sekwencja ta służy do wyświetlenia przycisków w szablonie, reprezentujących kolejne strony produktów. W tym celu korzystamy z dyrektywy `v-for`, z którą spotkaliśmy się wcześniej przy generowaniu zbioru elementów na liście produktów.

- **Wskazówka** Dane wymagane przez więcej niż jeden element aplikacji powinny być umieszczane w magazynie danych, podczas gdy dane ograniczone tylko do jednego komponentu powinny być zdefiniowane w elemencie `script`. Dane związane ze stronicowaniem umieścimy w magazynie, ponieważ wykorzystuję je do pobrania danych za pomocą usługi sieciowej w rozdziale 7.

Wcześniej omówiłem dyrektywę `v-bind`, która służy do definiowania atrybutu w elemencie HTML — wartością tego atrybutu może być wartość danych lub fragment kodu JavaScript. W listingu 5.17 zastosowałem dyrektywę `v-bind` do określenia wartości atrybutu `class`:

```
...
<button v-for="i in pageNumbers" v-bind:key="i" class="btn btn-secondary" v-bind:class="
  '{ 'btn-primary': i == currentPage, 'btn-secondary': i != currentPage }">
...
...
```

Vue.js dostarcza przydatne funkcje, które pozwalają na kontrolę klasy elementu. Klasę `btn-primary` otrzymujemy jedynie przycisk reprezentujący aktualnie przeglądaną stronę z danymi (więcej na ten temat w rozdziale 12.). Dzięki temu aktywny przycisk wygląda inaczej niż pozostałe, co daje użytkownikowi czytelną informację, na której stronie się on znajduje.

Dodanie komponentu paginacji do aplikacji wymaga dołączenia instrukcji `import` w celu dodania zależności. Następnie dodałem paginację do właściwości komponentu-rodzica (listing 5.18).

Listing 5.18. Zastosowanie paginacji w pliku src/components/ProductList.vue

```
<template>
  <div>
    <div v-for="p in products" v-bind:key="p.id" class="card m-1 p-1 bg-light">
      <h4>
        {{p.name}}
        <span class="badge badge-pill badge-primary float-right">
          {{ p.price | currency }}
        </span>
      </h4>
      <div class="card-text bg-white p-1">{{ p.description }}</div>
    </div>
    <page-controls />
  </div>
</template>
<script>
import { mapGetters } from "vuex";
import PageControls from "./PageControls";
export default {
  components: { PageControls },
  computed: {
    ...mapGetters({ products: "processedProducts" })
  },
  filters: {
    currency(value) {
      return new Intl.NumberFormat("pl-PL", { style: "currency", currency: "PLN"
        }).format(value);
    }
  }
}
</script>
```

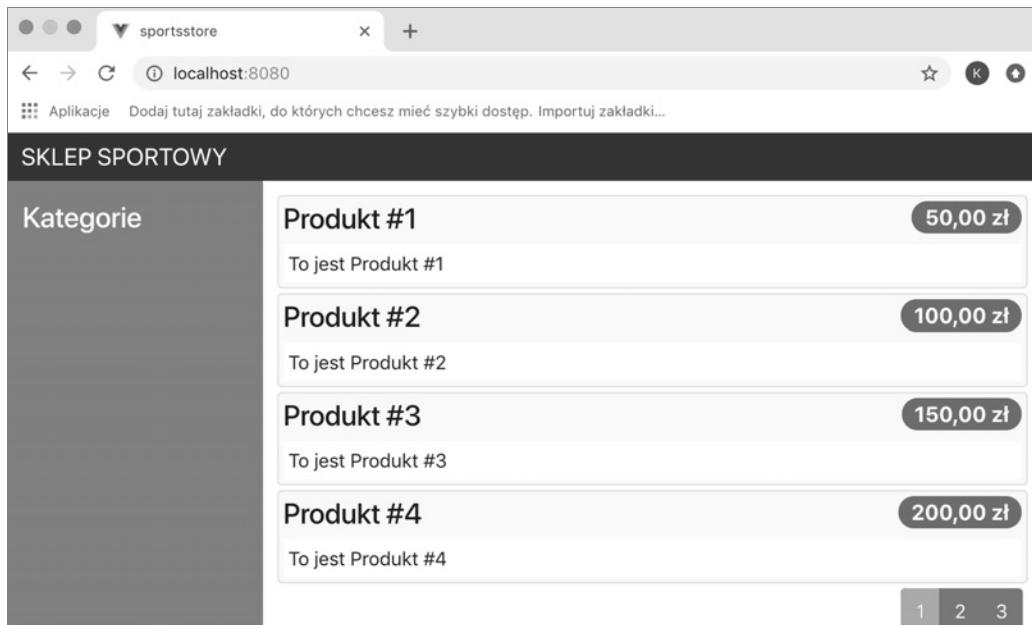
Zmieniłem także źródło danych wyświetlanych w komponentie ProductList na getter processedProducts. Oznacza to, że na liście będą wyświetlane tylko produkty z aktualnie wybranej strony. Zastosowanie funkcji mapGetters pozwala na powiązanie gettera processedProducts z nazwą products, dzięki czemu nie muszę zmieniać tej nazwy we wszystkich dyrektywach v-for w szablonie. Po zapisaniu zmian przeglądarka odświeży aplikację, a przyciski paginacji powinny pojawić się tak jak na rysunku 5.7.

Zmiana treści na liście produktów

Aby umożliwić użytkownikowi zmianę strony w obrębie listy produktów, muszę zareagować na kliknięcie jednego z przycisków. W listingu 5.19 korzystam z dyrektywy v-on, która pozwala zareagować na zajście zdarzenia takiego jak kliknięcie. W tej sytuacji wywołam mutację setCurrentPage magazynu danych.

Listing 5.19. Reagowanie na kliknięcie przycisków w pliku src/components/PageControls.vue

```
<template>
  <div class="text-right">
    <div class="btn-group mx-2">
      <button v-for="i in pageNumbers" v-bind:key="i"
        class="btn"
        v-bind:class="{ 'btn-primary': i == currentPage, 'btn-secondary':
          &gt;i != currentPage }"
        v-on:click="setCurrentPage(i)">
        {{ i }}
      </button>
    </div>
  </div>
```



Rysunek 5.7. Dodawanie przycisków stronicowania

```

</div>
</template>
<script>
  import { mapState, mapGetters, mapMutations } from "vuex";
  export default {
    computed: {
      ...mapState(["currentPage"]),
      ...mapGetters(["pageCount"]),
      pageNumbers() {
        return [...Array(this.pageCount + 1).keys()].slice(1);
      }
    },
    methods: {
      ...mapMutations(["setCurrentPage"])
    }
  }
</script>

```

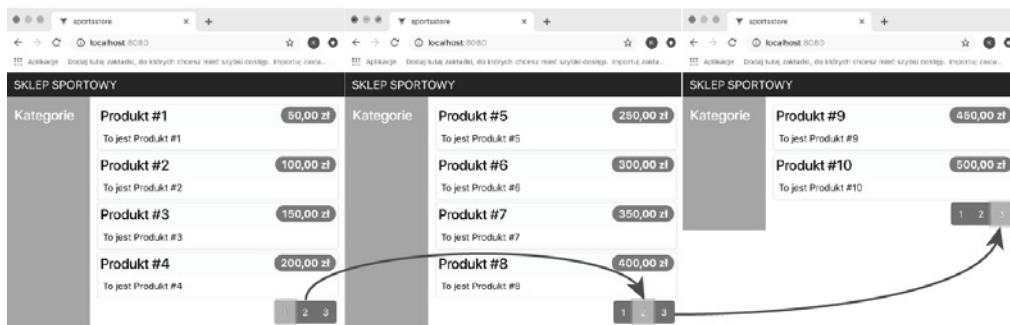
Funkcja `mapMutations` łączy funkcję `setCurrentPage` z metodą komponentu, wywoływaną przez dyrektywę `v-on` w momencie zajścia zdarzenia `click`.

```

...
<button v-for="i in pageNumbers" v-bind:key="i" class="btn" v-bind:class="{ 'btn-primary': i == currentPage, 'btn-secondary': i != currentPage }" v-on:click="setCurrentPage(i)">
...

```

Rodzaj zdarzenia jest określany jako argument dyrektywy. Jest on oddzielany od nazwy dyrektywy dwukropkiem. Wyrażenie nakazuje wywołanie metody `setCurrentPage` z przekazaniem wartości tymczasowej zmiennej i określającej numer strony, na którą chce przejść użytkownik. Funkcja `setCurrentPage` jest powiązana z mutacją magazynu danych o tej samej nazwie, co oznacza, że po kliknięciu jednego z przycisków paginacji dochodzi do zmiany zakresu wybranych produktów (rysunek 5.8).



Rysunek 5.8. Zmiana strony na liście produktów

Zmiana rozmiaru pojedynczej strony

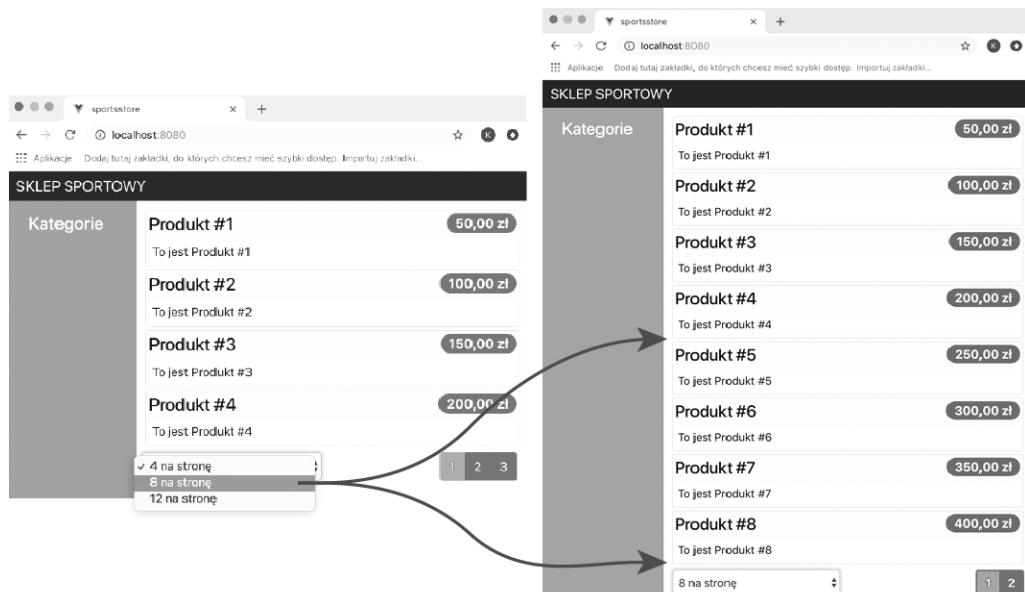
Aby zakończyć tworzenie funkcji paginacji, chciałbym pozwolić użytkownikowi na zmianę liczby produktów aktualnie wyświetlanych na stronie. Listing 5.20 wprowadza element typu select do szablonu komponentu. Dzięki niemu jesteśmy w stanie wywołać mutację pageSize magazynu danych po zmianie wartości pola select przez użytkownika.

Listing 5.20. Zmiana rozmiaru strony w pliku src/components/PageControls.vue

```
<template>
  <div class="row mt-2">
    <div class="col form-group">
      <select class="form-control" v-on:change="changePageSize">
        <option value="4">4 na stronę</option>
        <option value="8">8 na stronę</option>
        <option value="12">12 na stronę</option>
      </select>
    </div>
    <div class="text-right col">
      <div class="btn-group mx-2">
        <button v-for="i in pageNumbers" v-bind:key="i"
          class="btn btn-secondary"
          v-bind:class="{ 'btn-primary': i == currentPage, 'btn-secondary': i != currentPage }"
          v-on:click="setCurrentPage(i)">
          {{ i }}
        </button>
      </div>
    </div>
  </div>
</template>
<script>
  import { mapState, mapGetters, mapMutations } from "vuex";
  export default {
    computed: {
      ...mapState(["currentPage"]),
      ...mapGetters(["pageCount"]),
      pageNumbers() {
        return [...Array(this.pageCount + 1).keys()].slice(1);
      }
    },
    methods: {
      ...mapMutations(["setCurrentPage", "setPageSize"]),
    }
  }
</script>
```

```
        changePageSize($event) {
            this.setPageSize(Number($event.target.value));
        }
    }
</script>
```

Nowe elementy HTML dodają niezbędny kod do szablonu komponentu, dzięki czemu obok przycisków stronicowania pojawi się element select. Element ten wyświetla możliwe do wyboru opcje rozmiaru strony. Dyrektywa v-on nasłuchuje zdarzenia change, które jest wywoływanie po wyborze wartości przez użytkownika. Jeśli określisz tylko nazwę metody w dyrektywie v-on, metoda ta otrzyma obiekt zdarzenia, z którego możemy pobrać wszelkie informacje na temat wywołania zdarzenia. W tym przypadku korzystam z obiektu w celu pobrania wybranego rozmiaru strony i przekazania go do mutacji setPageSize w magazynie danych. Niezbędne powiązanie jest wykonywane przez funkcję mapMutations. W rezultacie rozmiar strony może być zmieniony poprzez zmianę aktualnie zaznaczonej wartości z listy select (rysunek 5.9).



Rysunek 5.9. Zmiana rozmiaru strony

- **Wskazówka** Zwróć uwagę, że wywołanie mutacji jest jedną czynnością niezbędną do zmiany stanu aplikacji. Vuex i Vue.js współpracują ze sobą, aby cała reszta pracy niezbędna do wyrenderowania odświeżonego widoku użytkownikowi została wykonana automatycznie.

Obsługa wyboru kategorii

Lista produktów powoli nabiera sensownego kształtu, dlatego możemy teraz zająć się dodaniem mechanizmu przeglądania listy według kategorii. Będę podążał poznaną już ścieżką — najpierw rozszerzę magazyn danych, następnie utworzę nowy komponent, aż wreszcie zintegruje komponent z resztą aplikacji.

W miarę upływu czasu przyzwyczaisz się do tego sposobu pracy. W listingu 5.21 dodaję getter do pobierania listy kategorii, z której użytkownik będzie mógł wybierać.

Listing 5.21. Dodawanie listy kategorii w pliku *src/store/index.js*

```

import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
const testData = [];
for (let i = 1; i <= 10; i++) {
  testData.push({
    id: i, name: `Produkt ${i}`, category: `Kategoria ${i % 3}`,
    description: `To jest Produkt ${i}`, price: i * 50
  })
}
export default new Vuex.Store({
  strict: true,
  state: {
    products: testData, productsTotal: testData.length, currentPage: 1,
    pageSize: 4,
    currentCategory: "Wszystkie"
  },
  getters: {
    productsFilteredByCategory: state => state.products
      .filter(p => state.currentCategory == "Wszystkie"
        || p.category == state.currentCategory),
    processedProducts: (state, getters) => {
      let index = (state.currentPage - 1) * state.pageSize;
      return getters.productsFilteredByCategory
        .slice(index, index + state.pageSize);
    },
    pageCount: (state, getters) =>
      Math.ceil(getters.productsFilteredByCategory.length / state.pageSize),
    categories: state => ["Wszystkie",
      ...new Set(state.products.map(p => p.category).sort())]
  },
  mutations: {
    setCurrentPage(state, page) {
      state.currentPage = page;
    },
    setPageSize(state, size) {
      state.pageSize = size;
      state.currentPage = 1;
    },
    setCurrentCategory(state, category) {
      state.currentCategory = category;
      state.currentPage = 1;
    }
  }
})

```

Właściwość stanu `currentCategory` reprezentuje kategorię aktualnie wybraną przez użytkownika.

Domyślnie jest to wartość `Wszystkie`, dzięki której aplikacja będzie w stanie wyświetlać wszystkie produkty niezależnie od kategorii.

Getterzy potrafią uzyskać dostęp do wyników innych getterów z magazynu danych przez wprowadzenie drugiego argumentu. W ten sposób mogę utworzyć getter `productsFilteredByCategory` i skorzystać z niego w getterach `processedProducts` i `pageCount`, aby uwzględnić wybór kategorii w ich wynikach.

Getter `categories` pozwala nam na zaprezentowanie użytkownikowi listy dostępnych kategorii. Getter przetwarza tablicę stanu `products`, aby uzyskać wartości właściwości `category` poszczególnych produktów. Następnie tworzymy z nich zbiór (Set), czyli kolekcję, która może zawierać tylko elementy unikatowe. W ten sposób likwidujemy duplikaty. Zbiór jest następnie rozwijany do postaci tablicy. Na zakończenie sortujemy go, dzięki czemu uzyskujemy posortowaną kolekcję unikatowych nazw kategorii.

Mutacja `setCurrentCategory` zmienia wartość właściwości stanu `currentCategory`, dzięki czemu będziemy w stanie przechowywać wybór użytkownika, a także odświeżyć zaznaczoną stronę.

Mechanizm zarządzania wyborem kategorii znajduje się w pliku `src/components/CategoryControls.vue` (listing 5.22).

Listing 5.22. Zawartość pliku `src/components/CategoryControls.vue`

```
<template>
<div class="container-fluid">
    <div class="row my-2" v-for="c in categories" v-bind:key="c">
        <button class="btn btn-block"
            v-on:click="setCurrentCategory(c)"
            v-bind:class="c == currentCategory
                ? 'btn-primary' : 'btn-secondary'"
            >{{ c }}</button>
    </div>
</div>
</template>
<script>
    import { mapState, mapGetters, mapMutations } from "vuex";
    export default {
        computed: {
            ...mapState(["currentCategory"]),
            ...mapGetters(["categories"])
        },
        methods: {
            ...mapMutations(["setCurrentCategory"])
        }
    }
</script>
```

Komponent przedstawia użytkownikowi listę przycisków. Jest ona generowana za pomocą dyrektywy `v-for` z wykorzystaniem wartości z właściwości `categories`. Właściwość ta jest mapowana na getter o tej samej nazwie z magazynu danych. Dyrektywa `v-bind` jest używana do określenia wartości atrybutu `class` przycisków. Jeśli przycisk reprezentuje aktualnie wybraną kategorię, otrzymuje on klasę `btn-primary`, a wszystkie pozostałe przyciski otrzymują klasę `btn-secondary`. Dzięki temu użytkownik od razu wie, która kategoria jest aktualnie wybrana.

Dyrektyna `v-on` nasłuchuje zdarzenia `click` i wywołuje mutację `setCurrentCategory`, która pozwala na przechodzenie pomiędzy kategoriami. Odświeżany na bieżąco model oznacza, że zmiana zostanie natychmiast odzwierciedlona w liście produktów przedstawionych użytkownikowi.

W listingu 5.23 importujemy nowy komponent i dodajemy go do właściwości `components` rodzica, dzięki czemu możemy wyświetlać nowe funkcje za pomocą własnego elementu HTML.

Listing 5.23. Dodawanie wyboru kategorii w pliku `src/components/Store.vue`

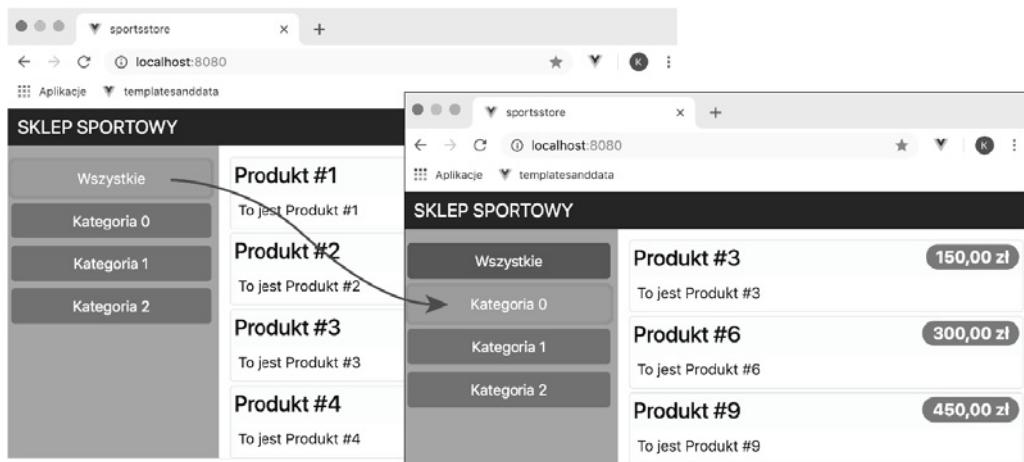
```
<template>
<div class="container-fluid">
    <div class="row">
        <div class="col bg-dark text-white">
            <a class="navbar-brand">SKLEP SPORTOWY</a>
        </div>
    </div>
    <div class="row">
        <div class="col-3 bg-info p-2">
            <CategoryControls />
        </div>
        <div class="col-9 p-2">
```

```

        <ProductList />
    </div>
</div>
</template>
<script>
    import ProductList from "./ProductList";
    import CategoryControls from "./CategoryControls";
    export default {
        components: { ProductList, CategoryControls }
    }
</script>

```

W efekcie użytkownik widzi listę przycisków, które mogą być wykorzystywane do filtrowania listy produktów według kategorii (rysunek 5.10).



Rysunek 5.10. Dodawanie obsługi filtrowania według kategorii

Zastosowanie REST-owej usługi sieciowej

Na początku tworzenia projektu lubię korzystać z danych testowych, ponieważ dzięki nim mogę zaprojektować pierwsze funkcje moich aplikacji bez konieczności obsługi żądań sieciowych. Skoro jednak pierwsze funkcje mamy już na swoim miejscu, musimy zamienić dane testowe na te pobrane z REST-owej usługi sieciowej. Pakiet json-server mamy już zainstalowany i uruchomiony. W tabeli 5.2 znajduje się zestawienie adresów URL, z których skorzystamy w naszej aplikacji.

Tabela 5.2. Lista adresów URL do pobrania danych w naszej aplikacji

URL	Opis
http://localhost:3500/products	Ten URL udostępnia listę produktów.
http://localhost:3500/categories	Ten URL udostępnia listę kategorii.

Vue.js nie zawiera wbudowanej obsługi żądań HTTP. Typowym wyborem w tym zakresie jest pakiet Axios. Nie jest on ograniczony jedynie do Vue.js — można z niego korzystać również w innych rozwiązaniach. Jest on dobrze zaprojektowany i łatwy w obsłudze.

Żądania HTTP są wykonywane asynchronicznie. Chcę wykonać żądania HTTP w magazynie danych. Vuex wspiera zadania asynchroniczne za pomocą tzw. **akcji** (ang. *actions*). W listingu 5.24 dodaję akcję do pobrania danych kategorii i produktów z serwera, a także modyfikuję właściwości stanu, na których opiera się działanie reszty aplikacji.

Listing 5.24. Pobieranie danych z serwera w pliku *src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
    strict: true,
    state: {
        products: [],
        categoriesData: [],
        productsTotal: 0,
        currentPage: 1,
        pageSize: 4,
        currentCategory: "Wszystkie"
    },
    getters: {
        productsFilteredByCategory: state => state.products
            .filter(p => state.currentCategory === "Wszystkie"
                || p.category === state.currentCategory),
        processedProducts: (state, getters) => {
            let index = (state.currentPage - 1) * state.pageSize;
            return getters.productsFilteredByCategory.slice(index,
                index + state.pageSize);
        },
        pageCount: (state, getters) =>
            Math.ceil(getters.productsFilteredByCategory.length / state.pageSize),
        categories: state => ["Wszystkie", ...state.categoriesData]
    },
    mutations: {
        setCurrentPage(state, page) {
            state.currentPage = page;
        },
        setPageSize(state, size) {
            state.pageSize = size;
            state.currentPage = 1;
        },
        setCurrentCategory(state, category) {
            state.currentCategory = category;
            state.currentPage = 1;
        },
        setData(state, data) {
            state.products = data pdata;
            state.productsTotal = data pdata.length;
            state.categoriesData = data cdata.sort();
        }
    },
    actions: {
        async getData(context) {
            let pdata = (await Axios.get(productsUrl)).data;
```

```

        let cdata = (await Axios.get(categoriesUrl)).data;
        context.commit("setData", { pdata, cdata });
    }
}
})

```

Pakiet Axios dostarcza metodę get w celu wykonywania żądań HTTP GET. Pobieram dane z obu adresów URL, korzystając ze słów kluczowych `async` i `await`, aby poczekać na otrzymanie danych. Metoda `get` zwraca obiekt, w którego właściwości `data` znajduje się obiekt powstały w wyniku przeparsowania odpowiedzi w formacie JSON, otrzymanej z usługi sieciowej.

Akcje Vuex to funkcje, które otrzymują obiekt kontekstu. Obiekt ten udostępnia możliwości magazynu danych. Akcja `getData` korzysta z kontekstu w celu wykonania mutacji `setData`. Nie mogę skorzystać z funkcji `mapMutation` wewnętrz magazynu danych, dlatego stosuję alternatywny mechanizm — wywołuję metodę `commit` i określam nazwę mutacji w argumencie.

Teraz muszę wykonać akcję magazynu danych w momencie inicjalizacji aplikacji. Komponenty Vue.js mają dobrze zdefiniowany cykl życia, opisany przeze mnie w rozdziale 17. Dla każdej części cyklu życia komponent może zadeklarować metody, które zostaną wówczas wywoływane. W listingu 5.25 dodaję metodę `created`, która jest wykonywana w momencie utworzenia komponentu. Korzystam z tej metody w celu wywołania akcji `getData`. Akcja jest powiązana z metodą dzięki funkcji `mapActions`.

Listing 5.25. Pobieranie danych w pliku src/App.vue

```

<template>
  <store />
</template>
<script>
  import Store from "./components/Store";
  import { mapActions } from "vuex";
  export default {
    name: 'app',
    components: { Store },
    methods: {
      ...mapActions(["getData"])
    },
    created() {
      this.getData();
    }
  }
</script>

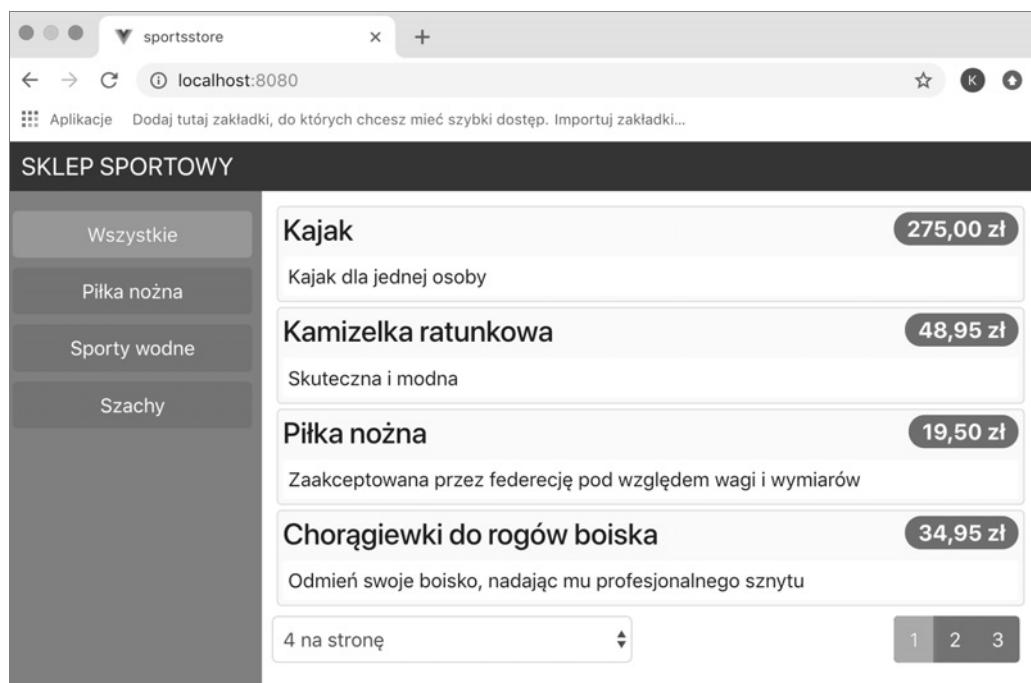
```

W efekcie dane testowe zostały zastąpione danymi z usługi sieciowej (rysunek 5.11).

■ **Uwaga** W tym przypadku, aby zobaczyć zmianę, konieczne może być ręczne odświeżenie przeglądarki. Jeśli dalej nie widzisz zmian, zrestartuj całą aplikację za pomocą narzędzi deweloperskich.

Podsumowanie

W tym rozdziale rozpoczęliśmy tworzenie projektu *Sklep sportowy*. Zaczeliśmy od utworzenia źródła danych, które daje współdzielony dostęp do danych w całej aplikacji. Oprócz tego opracowaliśmy mechanizm wyświetlania produktów użytkownikowi, ze wsparciem dla paginacji i filtrowania według kategorii. Na zakończenie dodaliśmy obsługę pakietu Axios w celu pobrania danych z REST-owej usługi sieciowej, dzięki czemu mogliśmy pozbyć się danych testowych. W kolejnym rozdziale będziemy kontynuować rozwój aplikacji, dodając obsługę koszyka zakupowego, procedury rozliczenia i tworzenia zamówień.



Rysunek 5.11. Zastosowanie danych z usługi sieciowej

ROZDZIAŁ 6.



Sklep sportowy: rozliczenie i zamówienia

W tym rozdziale kontynuuję dodawanie funkcji do sklepu sportowego, którego tworzenie rozpoczęłem w rozdziale 5. Dodamy obsługę koszyka, a także rozliczenia (ang. *checkout*), co pozwoli użytkownikowi zgłosić zamówienie do usługi sieciowej.

-
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.
-

Przygotowania do rozdziału

W tym rozdziale korzystamy z projektu *Sklep sportowy (SportsStore)* z rozdziału 5. W odniesieniu do stanu projektu z końca rozdziału 5. nie są wymagane żadne zmiany. Aby uruchomić REST-ową usługę sieciową, otwórz wiersz poleceń i wykonaj poniższe polecenie w katalogu *sportsstore*:

```
npm run json
```

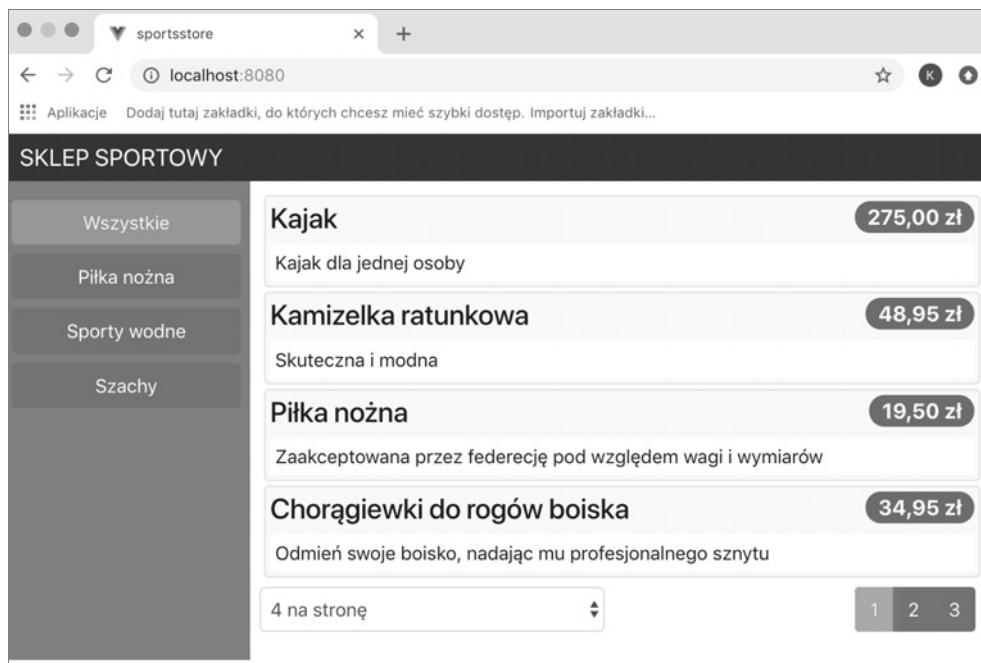
Otwórz drugą instancję wiersza poleceń i wykonaj poniższe polecenie w katalogu *sportsstore*, aby uruchomić narzędzia deweloperskie i serwer HTTP:

```
npm run serve
```

Po zakończeniu procesu budowania otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, a uzyskasz efekt jak na rysunku 6.1.

Tworzenie zastępczej treści dla koszyka

Kolejną funkcją, którą dodamy do naszego sklepu, jest koszyk zakupowy, który pozwoli użytkownikowi na przechowywanie wybranych przez niego produktów. Najpierw dodamy do aplikacji komponent zawierający treść zastępczą, a następnie obsłużymy mechanizm wyświetlania koszyka. Na sam koniec dokończymy obsługę działania koszyka. Zaczniemy od utworzenia pliku *ShoppingCart.vue* w katalogu *src/components* (listing 6.1).



Rysunek 6.1. Aplikacja Sklep sportowy po uruchomieniu

Listing 6.1. Zawartość pliku src/components/ShoppingCart.vue

```
<template>
  <h4 class="bg-primary text-white text-center p-2">
    Treść zastępcza koszyka
  </h4>
</template>
```

Teraz komponent nie zawiera żadnych funkcji, ale będziemy mieć czytelną informację, że w danym miejscu będzie się znajdował koszyk.

Konfiguracja trasowania adresów URL

Proste aplikacje wyświetlają ten sam rodzaj treści przez cały czas. W miarę dodawania nowych funkcji nadchodzi moment, w którym użytkownik musi mieć możliwość korzystania z różnych komponentów. W naszym przypadku istnieje konieczność łatwego przejścia pomiędzy listą produktów a koszykiem. Vue.js wspiera tzw. **dynamiczne komponenty**, które pozwalają na zmianę aktualnie oglądanej treści. Funkcja ta została utworzona na podstawie pakietu Vue Router. Dzięki niej to adres URL określa rodzaj treści wyświetlanej użytkownikowi — mechanizm ten nosi nazwę **trasowania adresów URL** (ang. *URL routing*). Aby skonfigurować pod tym kątem naszą aplikację, utworzyłem katalog *src/router* i dodałem do niego plik *index.js* (listing 6.2).

Listing 6.2. Zawartość pliku src/router/index.js

```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
Vue.use(VueRouter);
```

```
export default new VueRouter({
  mode: "history",
  routes: [
    { path: "/", component: Store },
    { path: "/cart", component: ShoppingCart },
    { path: "**", redirect: "/" }
  ]
})
```

- **Uwaga** Trasowanie URL jest szczegółowo omówione w rozdziałach 23. – 25., a dynamiczne komponenty — w rozdziale 21.

Pakiet Vue Router musi zostać zarejestrowany za pomocą metody `Vue.use` w ten sam sposób co pakiet Vuex w rozdziale 5. Plik `index.js` eksportuje nowy obiekt typu `VueRouter`, który otrzymuje obiekt konfiguracji wiążący adres URL z odpowiednimi komponentami.

- **Uwaga** W listingu 6.2 wartość właściwości `mode` ustawiam na `history`. Oznacza to, że Vue Router skorzysta z jednego z najnowszych API przeglądarek do obsługi adresów URL. W ten sposób otrzymamy bardziej użyteczny mechanizm, niemniej trzeba pamiętać, że nie jest on obsługiwany przez starsze przeglądarki (więcej na ten temat w rozdziale 22.).

Właściwość `routes` zawiera zestaw obiektów, które wiążą adresy URL z komponentami. Domyślny adres URL aplikacji spowoduje wyświetlenie komponentu `Store`, a adres `/cart` — wyświetlenie koszyka. Trzeci obiekt w tablicy `routes` przechwytuje wszystkie inne adresy i przekierowuje je do adresu domyslnego `(/)`, czyli do strony głównej sklepu.

Do pliku `main.js` dodałem instrukcje z listingu 6.3, które odpowiadają za właściwą inicjalizację trasowania i jego dostępność w całej aplikacji.

Listing 6.3. Włączenie trasowania URL (src/main.js)

```
import Vue from 'vue'
import App from './App.vue'
Vue.config.productionTip = false
import "bootstrap/dist/css/bootstrap.min.css";
import "font-awesome/css/font-awesome.min.css"
import store from "./store";
import router from "./router";
new Vue({
  render: h => h(App),
  store,
  router
}).$mount('#app')
```

Pierwsza z nowych instrukcji importuje moduł z listingu 6.3, a druga dodaje go do obiektu konfiguracji, przekazywanego do nowo tworzonego obiektu `Vue`. Bez tej operacji funkcje trasowania nie byłyby dostępne.

Wyświetlanie trasowanego komponentu

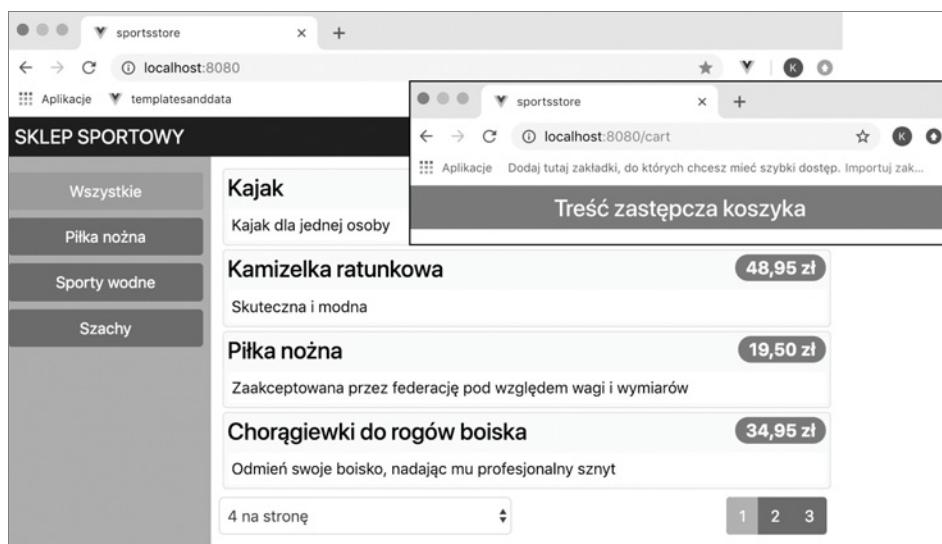
Skoro dodałem obsługę trasowania do aplikacji, mogę skorzystać z niego w celu wyświetlania komponentów w aplikacji. W listingu 6.4 usunąłem własny element HTML, który wyświetla komponent `Store`, i zastąpiłem go tym, który dynamicznie wstawi treść na podstawie bieżącego adresu URL.

Listing 6.4. Dodawanie widoku trasowanego w pliku src/App.vue

```
<template>
  <router-view />
</template>
<script>
  // import Store from "./components/Store";
  import { mapActions } from "vuex";
  export default {
    name: 'app',
    // components: { Store },
    methods: {
      ...mapActions(["getData"])
    },
    created() {
      this.getData();
    }
  }
</script>
```

Element router-view wyświetla komponent, bazując na konfiguracji zdefiniowanej w listingu 6.4. Oznaczyłem jako komentarze instrukcje odpowiedzialne za dodanie komponentu Store — nie są one już nam potrzebne, ponieważ od teraz do systemu trasowania odpowiada za zarządzanie treścią wyświetlana przez komponent App.

Jeśli przejdziesz pod adres <http://localhost:8080>, zobaczyś treść wyświetlzoną przez komponent Store. Pod adresem <http://localhost:8080/cart> ujrzyś zatem zastępczą treść koszyka (rysunek 6.2).



Rysunek 6.2. Zastosowanie trasowania adresów URL

Implementacja funkcji koszyka

Skoro aplikacja umie wyświetlać komponent koszyka, możemy uzupełnić go o niezbędne funkcje. W kolejnych punktach rozszerzymy magazyn danych, dodamy funkcje do komponentu Cart, aby wyświetlić produkty wybrane przez użytkownika, a także dodamy funkcje nawigacyjne, dzięki czemu użytkownik będzie w stanie wybrać produkty i zapoznać się z podsumowaniem swoich produktów w koszyku.

Dodatkowy moduł w magazynie danych

Dalszą pracę zaczniemy od rozszerzenia magazynu danych. W tym celu dodamy moduł JavaScript związanego ściśle z koszykiem, dzięki czemu będziemy w stanie oddzielić dane koszyka od istniejących elementów magazynu danych. Listing 6.5 przedstawia treść pliku *src/store/cart.js*.

Listing 6.5. Zawartość pliku *src/store/cart.js*

```
export default {
  namespaced: true,
  state: {
    lines: []
  },
  getters: {
    itemCount: state => state.lines.reduce((total, line) =>
      total + line.quantity, 0),
    totalPrice: state => state.lines.reduce((total, line) =>
      total + (line.quantity * line.product.price), 0),
  },
  mutations: {
    addProduct(state, product) {
      let line = state.lines.find(line => line.product.id == product.id);
      if (line != null) {
        line.quantity++;
      } else {
        state.lines.push({ product: product, quantity:1 });
      }
    },
    changeQuantity(state, update) {
      update.line.quantity = update.quantity;
    },
    removeProduct(state, lineToRemove) {
      let index = state.lines.findIndex(line => line == lineToRemove);
      if (index > -1) {
        state.lines.splice(index, 1);
      }
    }
  }
}
```

Aby rozszerzyć magazyn danych o nowy moduł, utworzyłem domyślną funkcję dla eksportu, która zwraca obiekt zawierający właściwości *state*, *getters* i *mutations*. Podążyłem tą samą drogą co w rozdziale 5., gdy dodałem magazyn danych do projektu. Właściwość *namespaced* otrzymuje wartość *true*, aby oddzielić listę produktów od koszyka w magazynie danych. Oznacza to, że będą one dostępne z wykorzystaniem prefiksu. Bez tego ustalenia mechanizmy zdefiniowane w pliku *cart.js* zostałyby złączone w jeden duży magazyn danych, co mogłoby prowadzić do niejasności, o ile nie zadbałbyś o nazewnictwo właściwości i funkcji.

Aby dołączyć moduł do magazynu danych, dodaję instrukcje z listingu 6.6 do pliku *index.js*.

Listing 6.6. Dodawanie modułu do pliku *src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
```

```
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
    strict: true,
    modules: { cart: CartModule },
    state: {
        products: [],
        categoriesData: [],
        productsTotal: 0,
        currentPage: 1,
        pageSize: 4,
        currentCategory: "Wszystkie"
    },
    //...pozostałe mechanizmy magazynu danych zostały pominięte...
})
```

Skorzystałem z instrukcji `import`, aby zadeklarować zależność od modułu koszyka pod nazwą `CartModule`. Dołączenie modułu do magazynu danych wymaga dodania właściwości `modules`, w której umieściłem specjalny obiekt. W tym obiekcie znalazła się właściwość, której nazwa będzie jednocześnie prefiksem stosowanym przez nas w celu uzyskania dostępu do danych z modułu. W związku z tym wszelkie elementy nowego modułu magazynu danych będą poprzedzane prefiksem `cart`.

Obsługa mechanizmu wyboru produktów

Kolejnym krokiem jest dołączenie funkcji, która pozwoli użytkownikowi na dodanie produktów do koszyka. W listingu 6.7 dodaję przycisk, który pojawi się obok każdego produktu na liście. Po jego kliknięciu produkt zostanie dodany do koszyka.

Listing 6.7. Dodawanie wyboru produktów w pliku src/components/ProductList.vue

```
<template>
    <div>
        <div v-for="p in products" v-bind:key="p.id" class="card m-1 p-1 bg-light">
            <h4>
                {{p.name}}
                <span class="badge badge-pill badge-primary float-right">
                    {{ p.price | currency }}
                </span>
            </h4>
            <div class="card-text bg-white p-1">
                {{ p.description }}
                <button class="btn btn-success btn-sm float-right"
                    v-on:click="handleProductAdd(p)">
                    Dodaj do koszyka
                </button>
            </div>
        </div>
        <page-controls />
    </div>
</template>
<script>
import { mapGetters, mapMutations } from "vuex";
import PageControls from "./PageControls";
export default {
    components: { PageControls },
    computed: {
        ...mapGetters({ products: "processedProducts" })
    },
    methods: {
        handleProductAdd(product) {
            this.$store.commit("addProduct", product);
        }
    }
}
```

```

filters: {
  currency(value) {
    return new Intl.NumberFormat("pl-PL", { style: "currency", currency:
      "PLN" }).format(value);
  }
},
methods: {
  ...mapMutations({ addProduct: "cart/addProduct" }),
  handleProductAdd(product) {
    this.addProduct(product);
    this.$router.push("/cart");
  }
}
</script>

```

Dodałem element button do szablonu i skorzystałem z dyrektywy v-on, aby zareagować na zdarzenie click, wywołując metodę handleProductAdd. Metoda ta wywołuje mutację cart/addProduct w magazynie danych. Jest to pierwszy przykład użycia prefiksu zdefiniowanego w listingu 6.7. Przypominam, że z prefiksów mogę korzystać, ponieważ właściwość namespaced jest ustawiona na true.

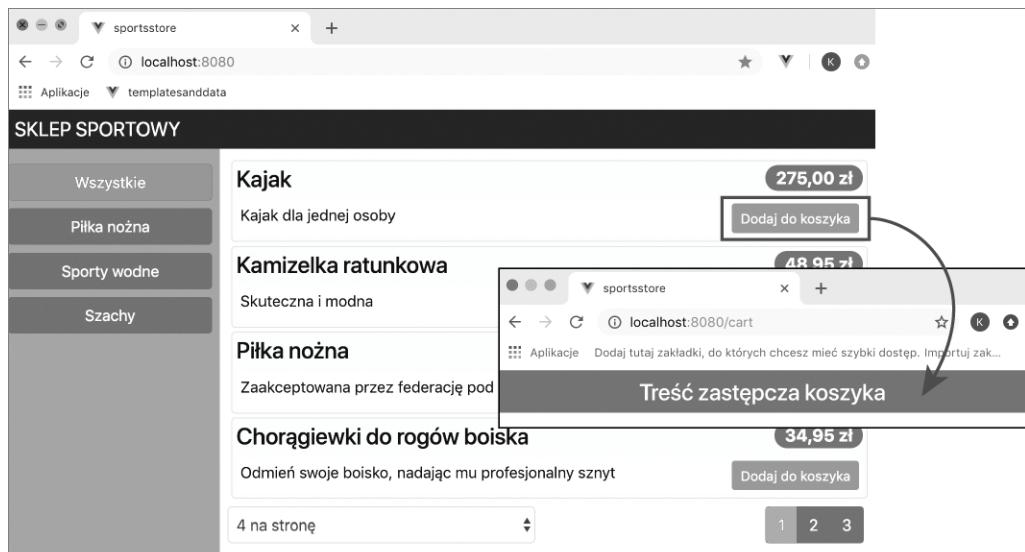
Po wywołaniu metody mutacji addProduct metoda handleProductAdd korzysta z trasowania, aby przejść pod adres /cart:

```

...
this.$router.push("/cart");
...

```

Mechanika oferowana przez pakiet Vue Router jest dostarczana do komponentów za pomocą właściwości \$router (dostęp do wszystkich właściwości i metod w komponencie wymaga zastosowania słowa this). Metoda push nakazuje routerowi zmienić adres URL przeglądarki, co w rezultacie prowadzi do wyświetlenia innego komponentu. W efekcie po kliknięciu przycisku *Dodaj do koszyka* zostaje wyświetlony komponent koszyka (rysunek 6.3).



Rysunek 6.3. Zastosowanie nawigacji URL

Wyświetlanie zawartości koszyka

Najwyższy czas wyświetlić wybrane przez użytkownika produkty z koszyka zamiast dodanej uprzednio treści zastępczej. Aby nieco uprościć cały mechanizm, wyodrębnim z niego osobny komponent do wyświetlania pojedynczego, wybranego produktu. Plik *ShoppingCartLine.vue* jest przedstawiony w listingu 6.8.

Listing 6.8. Zawartość pliku src/components/ShoppingCartLine.vue

```
<template>
  <tr>
    <td>
      <input type="number" class="form-control-sm"
             style="width:5em"
             v-bind:value="qvalue"
             v-on:input="sendChangeEvent"/>
    </td>
    <td>{{ line.product.name }}</td>
    <td class="text-right">
      {{ line.product.price | currency }}
    </td>
    <td class="text-right">
      {{ (line.quantity * line.product.price) | currency }}
    </td>
    <td class="text-center">
      <button class="btn btn-sm btn-danger"
             v-on:click="sendRemoveEvent">
        Usuń
      </button>
    </td>
  </tr>
</template>
<script>
  export default {
    props: ["line"],
    data: function() {
      return {
        qvalue: this.line.quantity
      }
    },
    methods: {
      sendChangeEvent($event) {
        if ($event.target.value > 0) {
          this.$emit("quantity", Number($event.target.value));
          this.qvalue = $event.target.value;
        } else {
          this.$emit("quantity", 1);
          this.qvalue = 1;
          $event.target.value = this.qvalue;
        }
      },
      sendRemoveEvent() {
        this.$emit("remove", this.line);
      }
    }
  }
</script>
```

Ten komponent korzysta z elementu `props`, który pozwala na przekazanie obiektów danych z komponentu rodzica do dziecka. W tym przypadku przekazujemy obiekt o nazwie `line`. W obiekcie tym rodzic przekaże z koszyka pojedynczy rekord, który zostanie wyświetlony w ramach tej instancji komponentu. Komponent potrafi także wysłać własne zdarzenie, za pomocą którego będzie komunikować się ze swoim rodzicem. Gdy użytkownik zmieni wartość elementu `input`, reprezentującego liczbę danego produktu, lub kliknie przycisk `Usuń`, komponent wywoła metodę `this.$emit`, przesyłając zdarzenie do rodzica. Tego rodzaju mechanizmy pozwalają na lokalną komunikację pomiędzy komponentami bez potrzeby korzystania z globalnych systemów, takich jak magazyn danych.

Aby wyświetlić zawartość koszyka, zamieniam treść zastępczą w komponencie `ShoppingCart` na kod HTML i JavaScript z listingu 6.9.

Listing 6.9. Wyświetlanie produktów w pliku `src/components/ShoppingCart.vue`

```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col bg-dark text-white">
        <a class="navbar-brand">SKLEP SPORTOWY</a>
      </div>
    </div>
    <div class="row">
      <div class="col mt-2">
        <h2 class="text-center">Twój koszyk</h2>
        <table class="table table-bordered table-striped p-2">
          <thead>
            <tr>
              <th>Ilość</th><th>Produkt</th>
              <th class="text-right">Cena</th>
              <th class="text-right">Podsum&gt;</th>
            </tr>
          </thead>
          <tbody>
            <tr v-if="lines.length == 0">
              <td colspan="4" class="text-center">
                Twój koszyk jest pusty
              </td>
            </tr>
            <cart-line v-for="line in lines" v-bind:key="line.product.id"
                       v-bind:line="line"
                       v-on:quantity="handleQuantityChange(line, $event)"
                       v-on:remove="remove" />
          </tbody>
          <tfoot v-if="lines.length > 0">
            <tr>
              <td colspan="3" class="text-right">Razem:</td>
              <td class="text-right">
                {{ totalPrice | currency }}
              </td>
            </tr>
          </tfoot>
        </table>
      </div>
    </div>
    <div class="row">
      <div class="col">
        <div class="text-center">
          <router-link to="/" class="btn btn-secondary m-1">
            Kontynuuj zakupy
          </router-link>
        </div>
      </div>
    </div>
  </div>
```

```

<router-link to="/checkout" class="btn btn-primary m-1"
             v-bind:disabled="lines.length == 0">
    Do kasę
</router-link>
</div>
</div>
</div>
</template>
<script>
import { mapState, mapMutations, mapGetters } from "vuex";
import CartLine from "./ShoppingCartLine";
export default {
  components: { CartLine },
  computed: {
    ...mapState({ lines: state => state.cart.lines }),
    ...mapGetters({ totalPrice : "cart/totalPrice" })
  },
  methods: {
    ...mapMutations({
      change: "cart/changeQuantity",
      remove: "cart/removeProduct"
    }),
    handleQuantityChange(line, $event) {
      this.change({ line, quantity: $event});
    }
  }
}
</script>

```

Większość przedstawionego kodu korzysta z funkcji, które już znasz, ale warto odnotować kilka ciekawych fragmentów. Najpierw przyjrzymy się metodzie konfiguracji komponentów ShoppingCartLine, będących dziećmi koszyka:

```

...
<cart-line v-for="line in lines" v-bind:key="line.product.id"
v-bind:line="line"
v-on:quantity="handleQuantityChange(line, $event)"
v-on:remove="remove" />
...

```

Element `cart-line` jest używany do zastosowania zimportowanej dyrektywy `CartLine`. Widać też lokalne funkcje komunikacyjne, które korzystają z możliwości elementu koszyka, zadeklarowanych we wcześniejszym listingu. Dyrektywa `v-bind` ustawia wartość właściwości `line`, za pomocą której dyrektywa `CartLine` otrzymuje obiekt do wyświetlenia. Dyrektywa `v-on` pozwala na obsługę specjalnych zdarzeń generowanych przez dyrektywę `CartLine`.

Niektóre fragmenty szablonu z listingu 6.9 będą wyświetlane tylko, jeśli koszyk będzie zawierać jakieś produkty. Elementy mogą być dodawane lub usuwane na podstawie wyrażenia języka JavaScript z wykorzystaniem dyrektywy `v-if` (więcej na jej temat znajdziesz w rozdziale 12.).

Szablon z listingu 6.9 zawiera nowy element, którego jeszcze nie widziałeś.

```

...
<router-link to="/" class="btn btn-primary m-1">Kontynuuj zakupy</router-link>
...

```

Element `router-link` jest dostarczany przez pakiet Vue Router, aby wygenerować element nawigacyjny. W momencie przetworzenia szablonu komponentu element `router-link` jest zamieniany na element kotwicy (znacznik `a`), który pozwoli na przejście pod adres URL określony za pomocą atrybutu `to`.

Element `router-link` stanowi alternatywę dla nawigacji realizowanej z poziomu kodu. Można skonfigurować go tak, aby efektem działania były inne elementy. Można też wykorzystać klasy, które będą stosowane wobec elementu, gdy adres danego elementu będzie identyczny z aktualnym adresem URL aplikacji (opisuję to szczegółowo w rozdziale 23.). W listingu 6.9 korzystam z elementu `router-link`, aby umożliwić użytkownikowi powrót do listy produktów i przejście do adresu (trasy) `/checkout`, który dodam do aplikacji w dalszej części rozdziału w celu obsłużenia procesu rozliczenia.

-
- **Wskazówka** Framework Bootstrap pozwala na stylowanie elementów w taki sposób, aby wyglądały jak przyciski. Klasy, które dołączylem do elementów `router-link` w listingu 6.9, odnoszą się również do elementów a. Są one przedstawiane w formie przycisków *Kontynuuj zakupy* i *Do kasę* (rysunek 6.4).
-

Na zakończenie omawiania listingu 6.9 chciałbym przybliżyć zastosowanie właściwości stanu magazynu danych, który zdefiniowałem w listingu 6.5. Postanowiłem oddzielić informacje zdefiniowane w module magazynu danych od reszty magazynu, co oznacza, że muszę skorzystać z prefiksu. W momencie wiązania getterów, mutacji i akcji prefiks musi być dołączony do nazwy. Nieco inne rozwiązanie przyjmujemy przy uzyskiwaniu dostępu do właściwości stanu.

```
...
...mapState({ lines: state => state.cart.lines }),  
...
```

Właściwości stanu są związane za pomocą funkcji, która otrzymuje stan obiektu i wybiera wymaganą przez nas właściwość. W tym przypadku wymaganą właściwością jest `lines`, dostępna za pośrednictwem prefiksu `cart`.

Tworzenie globalnego filtru

Gdy pierwszy raz wprowadziłem filtr `currency` (w rozdziale 5.), zadeklarowałem go w jednym komponencie. Teraz, gdy w naszym projekcie jest już więcej funkcji, jednocześnie pojawiło się więcej wartości, które powinny być formatowane jako kwoty pieniędzy. Zamiast duplikować filtr, zarejestruję go globalnie, dzięki czemu będzie dostępny dla wszystkich komponentów (listing 6.10).

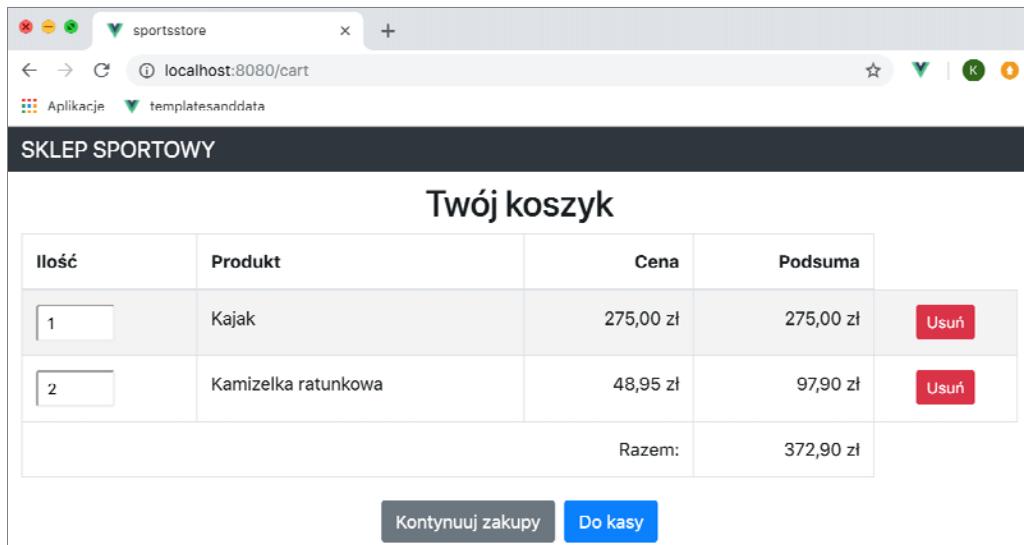
Listing 6.10. Tworzenie globalnego filtru w pliku src/main.js

```
import Vue from 'vue'  
import App from './App.vue'  
Vue.config.productionTip = false  
import "bootstrap/dist/css/bootstrap.min.css";  
import "font-awesome/css/font-awesome.min.css"  
import store from "./store";  
import router from "./router";  
Vue.filter("currency", (value) => new Intl.NumberFormat("pl-PL", { style: "currency", currency:  
  "PLN" }).format(value));  
new Vue({  
  render: h => h(App),  
  store,  
  router  
}).$mount('#app')
```

Jak przekonasz się w kolejnych rozdziałach, wiele fragmentów komponentów można zdefiniować globalnie, aby uniknąć duplikacji kodu. Filtry globalne definiuje się za pomocą wywołań metody `Vue.filter`, które muszą mieć miejsce przed utworzeniem obiektu Vue, jak w listingu 6.10. Więcej na temat filtrów znajdziesz w rozdziale 11.

Testowanie podstawowych funkcji koszyka

Aby przetestować koszyk, przejdź na stronę <http://localhost:8080> i kliknij przycisk *Dodaj do koszyka* przy produktach, które chcesz do niego dodać. Za każdym razem, gdy klikniesz przycisk, magazyn danych zostanie zaktualizowany, a przeglądarka przejdzie do trasy /cart, dzięki czemu zobaczysz podsumowanie swoich dotychczasowych wyborów (rysunek 6.4). Możesz zwiększać i zmniejszać liczbę produktów lub je usuwać. Możesz też kliknąć przycisk *Kontynuuj zakupy*, aby powrócić do sklepu. Kliknięcie przycisku *Do kasy* spowoduje w tym momencie powrót do strony głównej, ponieważ nie skonfigurowałem jeszcze trasy /checkout. W związku z tym zadziała trasa uniwersalna, którą zdefiniowałem w listingu 6.10. Trasa ta przekieruje przeglądarkę na domyślny adres, czyli do strony głównej.



Ilość	Produkt	Cena	Podsumma
1	Kajak	275,00 zł	275,00 zł
2	Kamizelka ratunkowa	48,95 zł	97,90 zł
	Razem:		372,90 zł

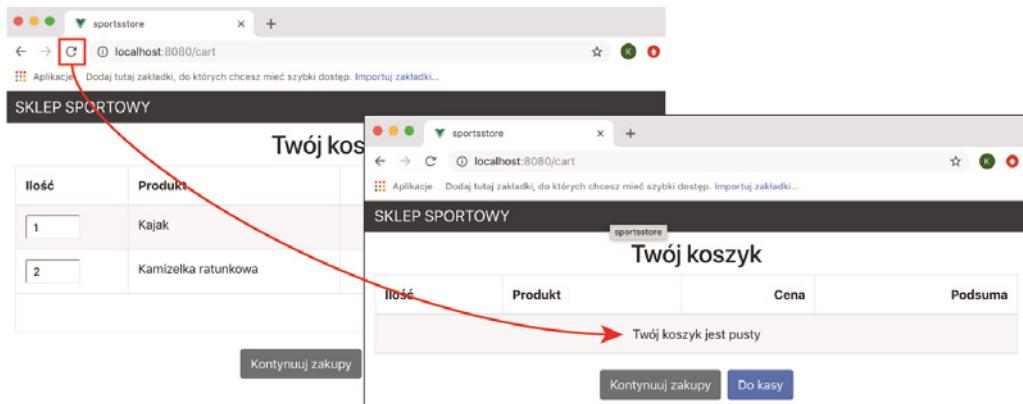
Rysunek 6.4. Podsumowanie koszyka

Utrwalanie koszyka

Jeśli przejdziesz pod adres <http://localhost:8080/cart> ręcznie lub odświeżysz przeglądarkę, utracisz wszystkie produkty, które uprzednio wybrałeś, przez co Twoim oczom ukaże się pusty koszyk, jak na rysunku 6.5.

Zmiany w adresie URL wykonane z poziomu systemu trasowania są obsługiwane inaczej niż te pochodzące od użytkownika. Jeżeli to aplikacja wykonuje przejście, zmiana jest interpretowana jako ruch w ramach aplikacji — np. przejście z listy produktów do magazynu. Jeżeli jednak to użytkownik nawiguje samodzielnie, zmiana jest interpretowana jako odświeżenie bądź zmiana strony. To, że aplikacja wie, jak obsługiwać zmianę adresu URL, jest nieistotne — tego rodzaju zmiana przerywa działanie aplikacji i powoduje wykonanie nowych, „świeżych” żądań HTTP, przez co tworzona jest nowa instancja aplikacji. Cały stan aplikacji zostaje usunięty, przez co widzisz pusty koszyk.

Takie zachowanie przeglądarki jest nieuniknione. Jedynym rozwiążaniem tego problemu jest utrwalenie koszyka, dzięki czemu będzie on dostępny zaraz po utworzeniu nowej instancji aplikacji. Istnieje wiele metod utrwalenia danych koszyka — jedną z najpopularniejszych jest utrwalenie go na serwerze przy każdej zmianie wywołanej przez użytkownika. W tej książce chcę się jednak skupić na Vue.js, a nie na zapisywaniu danych po stronie serwera. W związku z tym skorzystam z tego samego podejścia co w rozdziale 1. — zapiszę dane w lokalnej pamięci przeglądarki. W listingu 6.11 przedstawiam zmodyfikowany magazyn danych, w którym wybór produktów jest utrwalany każdorazowo po wprowadzeniu zmiany.



Rysunek 6.5. Pusty koszyk

Listing 6.11. Utrwalanie danych w pliku src/store/cart.js

```
export default {
  namespaced: true,
  state: {
    lines: []
  },
  getters: {
    itemCount: state => state.lines.reduce((total, line) =>
      total + line.quantity, 0),
    totalPrice: state => state.lines.reduce((total, line) =>
      total + (line.quantity * line.product.price), 0),
  },
  mutations: {
    addProduct(state, product) {
      let line = state.lines.find(line => line.product.id == product.id);
      if (line != null) {
        line.quantity++;
      } else {
        state.lines.push({ product: product, quantity:1 });
      }
    },
    changeQuantity(state, update) {
      update.line.quantity = update.quantity;
    },
    removeProduct(state, lineToRemove) {
      let index = state.lines.findIndex(line => line == lineToRemove);
      if (index > -1) {
        state.lines.splice(index, 1);
      }
    },
    setCartData(state, data) {
      state.lines = data;
    }
  },
  actions: {
    loadCartData(context) {
      let data = localStorage.getItem("cart");
      if (data != null) {
        context.commit("setCartData", JSON.parse(data));
      }
    }
  }
};
```

```
    },
    storeCartData(context) {
      localStorage.setItem("cart", JSON.stringify(context.state.lines));
    },
    clearCartData(context) {
      context.commit("setCartData", []);
    },
    initializeCart(context, store) {
      context.dispatch("loadCartData");
      store.watch(state => state.cart.lines, () => context.dispatch("storeCartData"),
        { deep: true});
    }
}
```

Do modułu magazynu danych dodałem akcje ładowania, zapisywania i czyszczenia danych koszyka przy użyciu API lokalnej pamięci. Akcje nie mogą modyfikować danych stanu bezpośrednio w magazynie, dlatego dodałem mutację, która ustawia właściwość `lines`. Najważniejszym dodatkiem jest akcja `initializeCart`, która odpowiada za obsługę koszyka w momencie startu aplikacji. W chwili wywołania akcji pierwsza instrukcja wykonuje metodę `dispatch`, która pozwala na wywołanie akcji z poziomu kodu. Aby móc obserwować magazyn danych pod kątem zmian właściwości stanu `lines`, korzystam z metody `watch`:

```
...  
store.watch(state => state.cart.lines, () => context.dispatch("storeCartData"), { deep: true});  
...
```

Argumenty metody `watch` to funkcja, która wybiera właściwość stanu, i funkcja wywoływaną w momencie wykrycia zmiany. Ta instrukcja wybiera właściwość `lines` i korzysta z metody `dispatch`, aby wykonać akcję `storeCartData`, gdy dochodzi do zmiany. Oprócz tego w wywołaniu jest obecny obiekt konfiguracji z właściwością `deep` ustawioną na wartość `true`. Wartość ta informuje Vuex, że chcemy otrzymywać powiadomienia o wszelkich zmianach właściwości w tablicy `lines` (nie dzieje się tak domyślnie). Jak wyjaśniam w rozdziale 13., bez wyboru tej opcji otrzymywaliśmy powiadomienia jedynie o dodawaniu elementu do koszyka bądź usuwaniu go z niego, ale już np. o zmianie liczebności produktu w koszyku — nie.

Vuex nie ma możliwości wywołania żadnej funkcji w momencieinicjalizacji magazynu danych, dlatego do komponentu App, obokistniejącej instrukcji pobierania danych z serwera, dodaję instrukcję, która wywołuje akcję initializeCart (listing 6.12).

Listing 6.12. Inicjalizacja koszyka w pliku src/App.vue

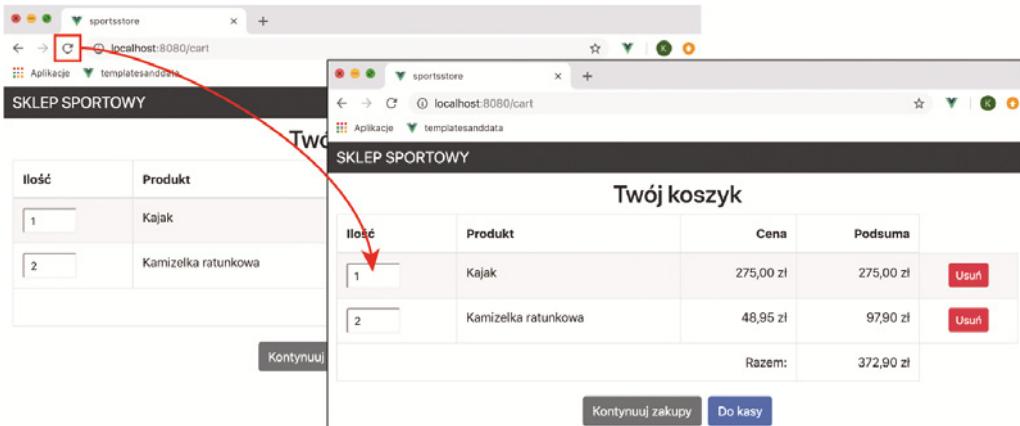
```
<template>
  <router-view />
</template>
<script>
  import { mapActions } from "vuex";
  export default {
    name: 'app',
    methods: {
      ...mapActions({
        getData: "getData",
        initializeCart: "cart/initializeCart"
      })
    },
    created() {
      this.getData();
      this.initializeCart(this.$store);
    }
  }
</script>
```

```

        }
    }
</script>

```

Efektem wprowadzonych zmian jest utrwalenie zmian dokonywanych w koszyku. Użytkownik nie straci zawartości koszyka po ręcznym przejęciu pod adres /cart ani po przeładowaniu przeglądarki (rysunek 6.6).



Rysunek 6.6. Przechowywanie danych koszyka

Dodawanie widżetu podsumowania koszyka

Ostatnim elementem koszyka pozostałym do zrobienia jest podsumowanie, wyświetlane na górze listy produktów, dzięki któremu użytkownik może zobaczyć podsumowanie swoich produktów i przejść wprost pod adres /cart bez konieczności dodawania produktu do koszyka. Plik *src/components/CartSummary.vue* z listingu 6.13 zawiera nowy komponent.

Listing 6.13. Zawartość pliku src/components/CartSummary.vue

```

<template>
  <div class="float-right">
    <small>
      Twój koszyk:
      <span v-if="itemCount > 0">
        {{ itemCount }} elementy(ów) {{ totalPrice | currency }}
      </span>
      <span v-else>
        (pusty)
      </span>
    </small>
    <router-link to="/cart" class="btn btn-sm bg-dark text-white"
      v-bind:disabled="itemCount == 0">
      <i class="fa fa-shopping-cart"></i>
    </router-link>
  </div>
</template>
<script>
  import { mapGetters } from 'vuex';
  export default {

```

```

        computed: {
            ...mapGetters({
                itemCount: "cart/itemCount",
                totalPrice: "cart/totalPrice"
            })
        }
    }
</script>

```

Komponent korzysta z dyrektywy `v-else`, która jest użytecznym uzupełnieniem dyrektywy `v-if`. Dyrektywa `v-else` skorzysta z elementu, jeśli wyrażenie w dyrektywie `v-if` przyjmie wartość `false` (więcej na ten temat w rozdziale 12.). Dzięki temu mogę wyświetlić podsumowanie, jeśli koszyk zawiera produkty, a w przeciwnym razie pokazuję jedynie komunikat.

- **Wskazówka** Element `router-link` z listingu 6.13 zawiera element `i`, który jest stylowany za pomocą stylów z pakietu *Font Awesome*, dodanego w rozdziale 5. Pakiet ten zapewnia doskonałe wsparcie w zakresie ikon dla aplikacji webowych, uwzględniając w tym również ikonę wózka sklepowego, niezbędną w naszej aplikacji. Więcej na temat tego pakietu znajdziesz na stronie <http://fontawesome.io/>.

Dodłczanie nowego komponentu do aplikacji wymaga zmian w komponencie `Store` (listing 6.14).

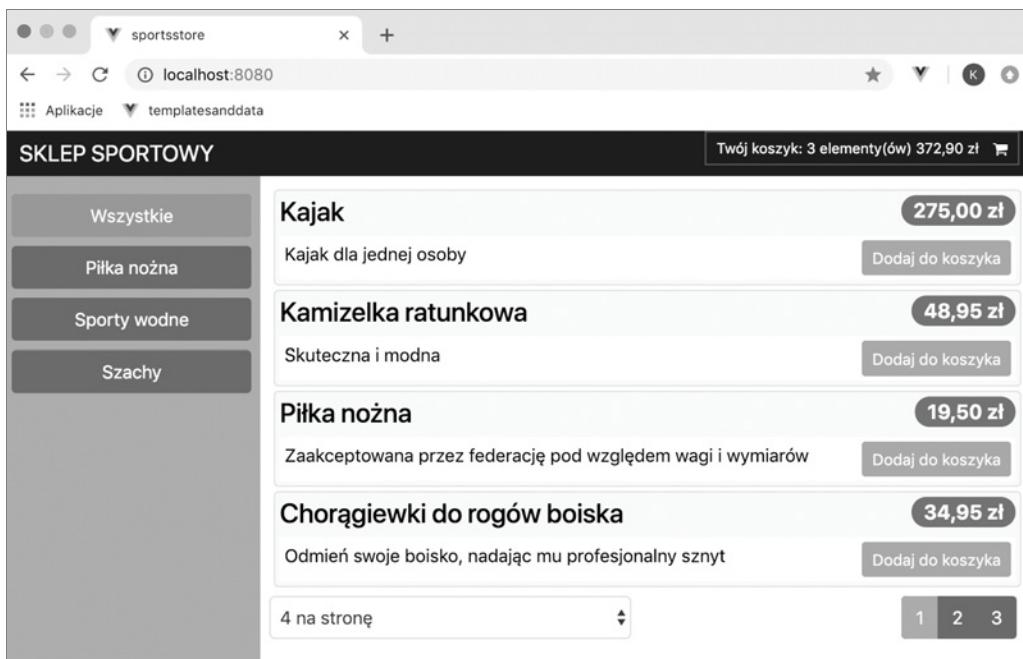
Listing 6.14. Dołączanie komponentu do pliku src/components/Store.vue

```

<template>
    <div class="container-fluid">
        <div class="row">
            <div class="col bg-dark text-white">
                <a class="navbar-brand">SKLEP SPORTOWY</a>
                <cart-summary />
            </div>
        </div>
        <div class="row">
            <div class="col-3 bg-info p-2">
                <CategoryControls />
            </div>
            <div class="col-9 p-2">
                <ProductList />
            </div>
        </div>
    </div>
</template>
<script>
    import ProductList from "./ProductList";
    import CategoryControls from "./CategoryControls";
    import CartSummary from "./CartSummary";
    export default {
        components: { ProductList, CategoryControls, CartSummary }
    }
</script>

```

W rezultacie użytkownik otrzymuje zwięzłe podsumowanie koszyka (rysunek 6.7), a także możliwość przejścia do całego koszyka przez kliknięcie ikony wózka.



Rysunek 6.7. Widżet podsumowania koszyka

Obsługa rozliczenia i dodawania zamówień

Kolejnym krokiem w tworzeniu naszej aplikacji jest możliwość rozliczenia zamówienia (przejścia do sklepowej kasę) i jego wygenerowania. W tym celu do katalogu `src/store` dodaję plik `orders.js` (listing 6.15).

Listing 6.15. Zawartość pliku src/store/orders.js

```
import Axios from "axios";
const ORDERS_URL = "http://localhost:3500/orders";
export default {
  actions: {
    async storeOrder(context, order) {
      order.cartLines = context.rootState.cart.lines;
      return (await Axios.post(ORDERS_URL, order)).data.id;
    }
  }
}
```

Nowy moduł magazynu danych zawiera tylko akcję do zapisywania zamówienia — kolejne mechanizmy pojawią się w trakcie implementowania funkcji administracyjnych. Akcja `storeOrder` korzysta z pakietu Axios w celu wysłania żądania HTTP POST do usługi sieciowej, co doprowadzi do zapisania zamówienia w bazie danych.

Pobranie danych z innego modułu wymaga zastosowania właściwości `rootState` kontekstu akcji. W ten sposób mogę skorzystać z właściwości `lines` koszyka, dzięki czemu produkty wybrane przez użytkownika są przesyłane do usługi sieciowej wraz z informacjami podanymi przez użytkownika w procesie rozliczenia „przy kasie”.

Pakiet json-server, z którego skorzystałem do utworzenia REST-owej usługi sieciowej, odpowiada na żądania POST obiektem w formacie JSON, zawierającym właściwość `id`. Właściwość ta jest przypisywana

automatycznie i zawiera unikatowy identyfikator obiektu w bazie danych. W listingu 6.15 korzystam ze słów kluczowych `async` i `await`, aby zaczekać na odpowiedź na żądanie POST i zwrócić wartość właściwości `id`, dostarczoną przez serwer.

Dla zachowania różnorodności postanowiłem w tym przypadku nie korzystać z funkcji przestrzeni nazw. Oznacza to, że gettery i mutacje tego modułu zostaną złączone z tymi zadeklarowanymi w pliku `index.js` i nie będą dostępne za pomocą prefiksu (choć, jak tłumacz w rozdziale 20., właściwości `state` są zawsze prefiksowane, nawet gdy nie korzystasz jawnie z przestrzeni nazw — przykład tego zobaczysz w rozdziale 7.). W listingu 6.16 importuję moduł `orders` do głównego pliku magazynu danych i dodaję go do właściwości `modules`, tak jak wcześniej zrobiłem z modułem `cart`.

Listing 6.16. Importowanie modułu w pliku src/store/index.js

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
    strict: true,
    modules: { cart: CartModule, orders: OrdersModule },
    // ...dane pominięte dla skrócenia listingu...
})
```

Tworzenie i rejestracja komponentów rozliczenia

Aby pomóc użytkownikowi w rozliczeniu zakupów, dodaję plik `Checkout.vue` do katalogu `src/components` (listing 6.17).

Listing 6.17. Zawartość pliku src/components/Checkout.vue

```
<template>
<div>
    <div class="container-fluid">
        <div class="row">
            <div class="col bg-dark text-white">
                <a class="navbar-brand">SKLEP SPORTOWY</a>
            </div>
        </div>
    </div>
    <div class="m-2">
        <div class="form-group m-2">
            <label>Imię</label>
            <input v-model="name" class="form-control" />
        </div>
    </div>
    <div class="text-center">
        <router-link to="/cart" class="btn btn-secondary m-1">
            Powrót
        </router-link>
        <button class="btn btn-primary m-1" v-on:click="submitOrder">
            Złóż zamówienie
        </button>
    </div>
</div>
```

```

        </div>
    </div>
</template>
<script>
export default {
    data: function() {
        return {
            name: null
        }
    },
    methods: {
        submitOrder() {
            // todo: zapisz zamówienie
        }
    }
}
</script>

```

Zacząłem od dodania jednego elementu formularza, który pozwala użytkownikowi na wprowadzenie imienia. Pozostałe elementy dodam niebawem, ale chciałem zacząć od wąskiego zakresu informacji, aby ustawić prawidłowo cały proces bez konieczności powtarzania tego samego kodu w wielu listingach.

Ten komponent korzysta z dwóch mechanizmów, które nie pojawiły się wcześniej, ponieważ nasza aplikacja jest zbudowana na bazie magazynu danych Vuex. Pierwszy z nich to przykład posiadania przez komponent danych lokalnych, które nie są dostępne dla żadnego innego komponentu. W tym celu w dość nietypowy sposób definiujemy właściwość data w elemencie script:

```

...
data: function() {
    return {
        name: null
    }
},
...

```

Ten fragment kodu definiuje pojedynczą właściwość danych o nazwie name zainicjalizowaną wartością null. Z pewnością przyzwyczaisz się do tego wyrażenia w miarę postępów w nauce Vue.js. W rozdziale 11. objaśniam, że brak właściwego ustawienia właściwości data skutkuje otrzymaniem ostrzeżenia. W drugiej i trzeciej części tej książki korzystam intensywnie z właściwości danych, aby zademonstrować różne funkcje bez konieczności dodawania magazynu danych.

Drugą funkcją zaprezentowaną w listingu 6.17 jest dyrektywa v-model elementu input, która tworzy dwukierunkowe wiązanie z właściwością name. Wiązanie oznacza, że wartość właściwości name i zawartość elementu input są ze sobą zsynchronizowane. Jest to niezwykle wygodne w pracy z elementami formularza (więcej na ten temat w rozdziale 15.). Nie korzystałem z tej możliwości we wcześniej omawianych komponentach sklepu, ponieważ nie może być ona stosowana z magazynem danych. Wynika to z faktu, że Vuex wymaga wprowadzenia zmian za pomocą mutacji, a tego z kolei nie obsługuje dyrektywa v-model (por. rozdział 20.).

Do wyświetlenia komunikatu o wysłaniu zamówienia utworzę odrębny komponent. Do pliku *src/components/OrderThanks.vue* dodaję zawartość listingu 6.18.

Listing 6.18. Zawartość pliku *src/components/OrderThanks.vue*

```

<template>
    <div class="m-2 text-center">
        <h2>Dziękujemy!</h2>
        <p>Dziękujemy za złożenie zamówienia o identyfikatorze {{orderId}}.</p>
        <p>Dostarczymy wybrane produkty tak szybko, jak to możliwe.</p>
        <routert-link to="/" class="btn btn-primary">Powrót do sklepu</routert-link>
    </div>

```

```
</template>
<script>
    export default {
        computed: {
            orderId() {
                return this.$route.params.id;
            }
        }
    }
</script>
```

Komponent wyświetla wartość pozyskaną z adresu URL aktualnej trasy za pomocą właściwości `this.$route`. Jest ona dostępna we wszystkich komponentach, gdy włączony jest pakiet Vue Router. W tym przypadku z trasy uzyskuję parametr reprezentujący numer zamówienia użytkownika. Numer ten zostanie uzyskany z żądania HTTP POST z listingu 6.15.

Dodatek komponentów do aplikacji wymaga zadeklarowania nowych tras (listing 6.19).

Listing 6.19. Dodawanie tras do pliku `src/router/index.js`

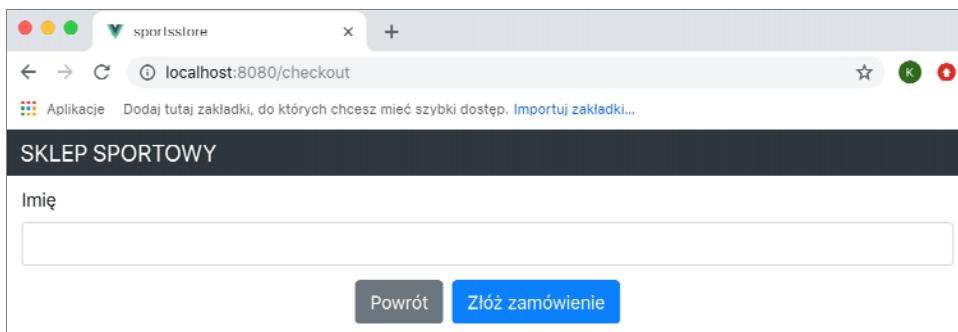
```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
Vue.use(VueRouter);
export default new VueRouter({
    mode: "history",
    routes: [
        { path: "/", component: Store },
        { path: "/cart", component: ShoppingCart },
        { path: "/checkout", component: Checkout },
        { path: "/thanks/:id", component: OrderThanks }, { path: "*", redirect: "/" }
    ]
})
```

Adres URL `/checkout` odwołuje się do komponentu z listingu 6.17. Odnosi się on bezpośrednio do adresu URL, który zastosowałem w elemencie `router-link` wyświetlonym w koszyku prezentowanym użytkownikowi w formie przycisku *Do kasy*.

Adres URL `/thanks/:id` wyświetla wiadomość z podziękowaniem, korzystając z komponentu z listingu 6.18. Dwukropki z segmentu `:id` oznaczają, że system trasowania powinien dopasować do tej trasy dowolny adres URL składający się z dwóch segmentów, gdzie pierwszym segmentem jest `thanks`. Zawartość drugiego segmentu adresu zostanie przypisana do zmiennej `id`, która następnie zostanie zaprezentowana w komponencie `OrderThanks` (listing 6.18). Więcej na temat tworzenia złożonych tras znajdziesz w rozdziałach 23. – 25.

Aby przetestować komponent rozliczenia, przejdź pod adres `http://localhost:8080/checkout` lub wejdź na stronę główną, wybierz produkt, a następnie kliknij przycisk *Do kasy*. Niezależnie od metody dostępu zobaczyesz efekt jak na rysunku 6.8.

- **Wskazówka** Skierowanie użytkownika wprost do kasy jest spotykane niezwykle rzadko. Metody ograniczenia nawigacji opisuję w rozdziale 7.



Rysunek 6.8. Początkowy stan komponentu rozliczenia

Dodawanie formularza walidacji

Walidacja danych przesyłanych przez użytkowników jest sprawą niezwykłej wagi. Aplikacja musi otrzymać dane w formacie, który jest w stanie przetworzyć. W rozdziale 15. omawiam, jak można utworzyć własny kod walidacji formularza, jednak w prawdziwym projekcie znacznie łatwiej skorzystać z pakietu, który zrobi to za nas — np. Vuelidate, dodanego do projektu w rozdziale 5. W listingu 6.20 dodaję instrukcję do pliku `main.js`, aby zadeklarować zależność od pakietu Vuelidate i dodać mechanizm do aplikacji.

Listing 6.20. Włączanie pakietu Vuelidate do pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
Vue.config.productionTip = false
import "bootstrap/dist/css/bootstrap.min.css";
import "font-awesome/css/font-awesome.min.css"
import store from "./store";
import router from "./router";
import Vuelidate from "vuelidate";
Vue.filter("currency", (value) => new Intl.NumberFormat("pl-PL",
    { style: "currency", currency: "PLN" }).format(value));
Vue.use(Vuelidate);
new Vue({
  render: h => h(App),
  store,
  router
}).$mount('#app')
```

Vuelidate jest rozpowszechniany w formie wtyczki Vue.js, która musi być zarejestrowana za pomocą metody `Vue.use` przed utworzeniem obiektu Vue, jak widać w listingu. Zasadę działania wtyczek i sposoby tworzenia własnych omawiam w rozdziale 26.

Kluczowym aspektem walidacji danych jest dostarczenie do użytkownika komunikatu, gdy wartość nie może zostać zaakceptowana lub nie została dostarczona. Trzeba się upewnić, że komunikat jest użyteczny dla użytkownika, a ponadto że nie zostanie wyświetlony do momentu wysłania danych, co mogłoby doprowadzić do zduplikowania kodu dla każdego z elementów `input`, które będą walidowane. Aby tego uniknąć, dodaję plik `src/components/ValidationError.vue`, z którego skorzystałem do utworzenia komponentu w listingu 6.21.

Listing 6.21. Zawartość pliku src/components/ValidationError.vue

```
<template>
<div v-if="show" class="text-danger">
  <div v-for="m in messages" v-bind:key="m">{{ m }}</div>
```

```

        </div>
</template>
<script>
export default {
    props: ["validation"],
    computed: {
        show() {
            return this.validation.$dirty && this.validation.$invalid
        },
        messages() {
            let messages = [];
            if (this.validation.$dirty) {
                if (this.hasValidationError("required")) {
                    messages.push("Proszę wprowadzić wartość")
                } else if (this.hasValidationError("email")) {
                    messages.push("Proszę wprowadzić prawidłowy adres e-mail");
                }
            }
            return messages;
        }
    },
    methods: {
        hasValidationError(type) {
            return this.validation.$params.hasOwnProperty(type) && !this.validation[type];
        }
    }
}
</script>

```

Pakiet Vuelidate dostarcza szczegółowych informacji na temat walidacji danych za pomocą obiektu, który zostanie przekazany przez prop komponentu o nazwie validation. W aplikacji *Sklep sportowy* obsługuję dwa walidatory — required, który sprawdza, czy użytkownik wprowadził jakąkolwiek wartość, i email, który sprawdza format podanej wartości pod kątem zgodności z formatem adresu e-mail.

- **Wskazówka** Vuelidate zawiera duży zestaw validatorów. Więcej szczegółów na stronie <https://github.com/vuelidate/vuelidate>.

Aby określić, które komunikaty powinien wyświetlić komponent ValidationError, korzystam z właściwości z tabeli 6.1, zdefiniowanych w obiekcie, który zostanie otrzymany za pośrednictwem propa.

Tabela 6.1. Właściwości obiektu walidacji

Nazwa	Opis
\$invalid	Jeśli ta właściwość ma wartość true, oznacza to, że zawartość elementów naruszyła minimum jedną z zastosowanych reguł walidacji.
\$dirty	Jeśli ta właściwość ma wartość true, oznacza to, że element został wyedytowany przez użytkownika.
required	Jeśli ta właściwość istnieje, validator required zostanie zastosowany do elementu. Jeśli właściwość ma wartość false, oznacza to, że element nie zawiera wartości.
email	Jeśli ta właściwość istnieje, validator email zostanie zastosowany do elementu. Jeśli właściwość ma wartość false, oznacza to, że element nie zawiera poprawnego adresu e-mail.

Komponent z listingu 6.21 korzysta z właściwości z tabeli 6.1, aby określić, czy obiekt otrzymany za pomocą propa zgłasza błędy walidacji, a jeśli tak — to które komunikaty powinny być wyświetlane dla użytkownika.

To podejście z pewnością stanie się bardziej zrozumiałe, gdy zobaczymy sposób, w jaki stosujemy komponent `Validation`. W listingu 6.22 dodaję walidację danych do komponentu `Checkout`, a wszystkie problemy są przekazywane do komponentu `Validation`.

Listing 6.22. Walidacja danych w pliku src/components/Checkout.vue

```
<template>
  <div>
    <div class="container-fluid">
      <div class="row">
        <div class="col bg-dark text-white">
          <a class="navbar-brand">SKLEP SPORTOWY</a>
        </div>
      </div>
    </div>
    <div class="m-2">
      <div class="form-group m-2">
        <label>Imię</label>
        <input v-model="$v.name.$model" class="form-control" />
        <validation-error v-bind:validation="$v.name" />
      </div>
    </div>
    <div class="text-center">
      <router-link to="/cart" class="btn btn-secondary m-1">
        Powrót
      </router-link>
      <button class="btn btn-primary m-1" v-on:click="submitOrder">
        Złóż zamówienie
      </button>
    </div>
  </div>
</template>
<script>
import { required } from "vuelidate/lib/validators";
import Validation from "./Validation";
export default {
  components: { Validation },
  data: function() {
    return {
      name: null
    }
  },
  validations: {
    name: {
      required
    }
  },
  methods: {
    submitOrder() {
      this.$v.$touch();
      // todo: zapisz zamówienie
    }
  }
}
</script>
```

Walidacja danych w Vuelidate jest wdrażana za pomocą właściwości `validations` w elemencie `script`. Właściwość ta sama zawiera właściwości odpowiadające nazwom wartości `data`, które będą walidowane. W listingu dodaję właściwość `name` i stosuję validator `required`, który musi zostać zainportowany z katalogu `vuelidate/lib/validators`.

Podłączenie funkcji walidacji do elementu `input` wymaga zmiany dyrektywy `v-model`, np. tak:

```
...
<input v-model="$v.name.$model" class="form-control" />
...
```

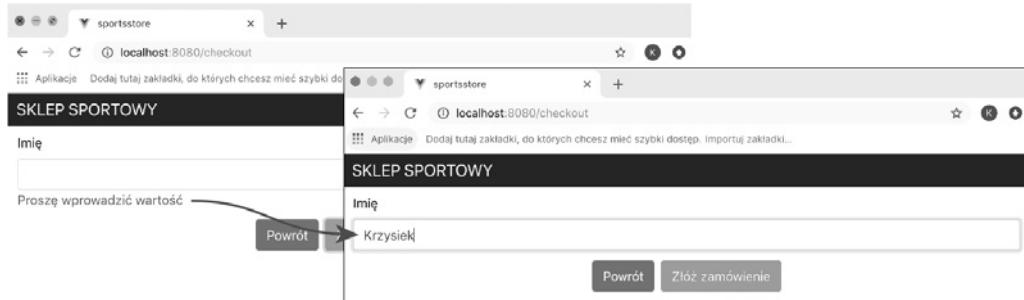
Mechanizmy walidacji są dostępne za pomocą właściwości `$v`, której właściwości odpowiadają ustawieniom konfiguracji walidacji. W tym przypadku mamy do czynienia z właściwością `name`, która odpowiada wartości danych `name`, a jej wartość jest dostępna za pomocą właściwości `$model`.

To właśnie obiekt zwrzony za pomocą właściwości `$v.name` przekazuję do komponentu `Validation`:

```
...
<validation-error v-bind:validation="$v.name" />
...
```

To podejście daje dostęp do właściwości opisanych w tabeli 6.1, używanych do wyświetlenia komunikatów o błędach walidacji bez potrzeby określania szczegółowych informacji na temat przetwarzanych wartości danych.

Obiekt `$v` definiuje metodę `$touch`, która oznacza wszystkie elementy jako `niesprawdzone`, tak jakby użytkownik zmienił je wszystkie. Ta użyteczna funkcja umożliwia wyzwolenie mechanizmu walidacji jako wyniku działania użytkownika przy jednoczesnym powstrzymaniu się od wyświetlania komunikatów walidacji do momentu interakcji z elementem `input` przez użytkownika. Aby sprawdzić efekt, przejdź na stronę `http://localhost:8080/checkout` i kliknij przycisk `Złóż zamówienie` bez wprowadzania jakichkolwiek danych. Zobaczysz komunikat z prośbą o wprowadzenie tekstu do pola, który to komunikat zniknie po wprowadzeniu tekstu, jak na rysunku 6.9. Walidacja jest wykonywana na żywo, tak więc zobaczysz błąd ponownie, jeśli usuniesz cały tekst z pola tekstowego.



Rysunek 6.9. Walidacja danych

Dodawanie pozostałych pól i walidacji

W listingu 6.23 dodaję pozostałe pola danych, niezbędne do złożenia zamówienia, wraz z ustawieniami walidacji, które są konieczne, a także kodem do wysłania zamówienia po sprawdzeniu formularza.

Listing 6.23. Uzupełnianie formularza w pliku src/components/Checkout.vue

```
<template>
<div>
    <div class="container-fluid">
        <div class="row">
```

```

        <div class="col bg-dark text-white">
            <a class="navbar-brand">SKLEP SPORTOWY</a>
        </div>
    </div>
<div class="m-2">
    <div class="form-group m-2">
        <label>Imię</label>
        <input v-model="$v.order.name.$model" class="form-control" />
        <validation-error v-bind:validation="$v.order.name" />
    </div>
</div>
<div class="m-2">
    <div class="form-group m-2">
        <label>E-mail</label>
        <input v-model="$v.order.email.$model" class="form-control" />
        <validation-error v-bind:validation="$v.order.email" />
    </div>
</div>
<div class="m-2">
    <div class="form-group m-2">
        <label>Adres</label>
        <input v-model="$v.order.address.$model" class="form-control" />
        <validation-error v-bind:validation="$v.order.address" />
    </div>
</div>
<div class="m-2">
    <div class="form-group m-2">
        <label>Miasto</label>
        <input v-model="$v.order.city.$model" class="form-control" />
        <validation-error v-bind:validation="$v.order.city" />
    </div>
</div>
<div class="m-2">
    <div class="form-group m-2">
        <label>Kod pocztowy</label>
        <input v-model="$v.order.zip.$model" class="form-control" />
        <validation-error v-bind:validation="$v.order.zip" />
    </div>
</div>
<div class="text-center">
    <router-link to="/cart" class="btn btn-secondary m-1">
        Powrót
    </router-link>
    <button class="btn btn-primary m-1" v-on:click="submitOrder">
        Złóż zamówienie
    </button>
</div>
</template>
<script>
import { required, email } from "vuelidate/lib/validators";
import ValidationError from "./ValidationError";
import { mapActions } from "vuex";
export default {
    components: { ValidationError },
    data: function() {
        return {
            order: {

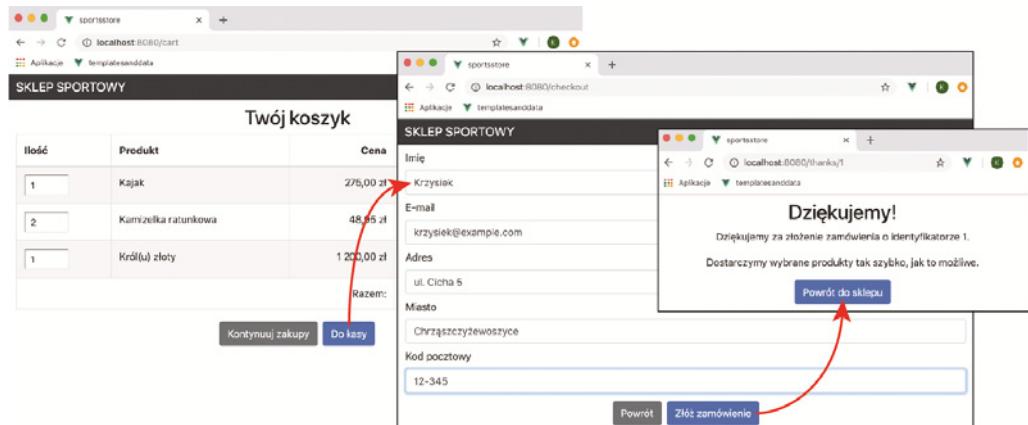
```

```

        name: null,
        email: null,
        address: null,
        city: null,
        zip: null
    }
},
validations: {
    order: {
        name: { required },
        email: { required, email },
        address: { required },
        city: { required },
        zip: { required }
    }
},
methods: {
    ...mapActions({
        "storeOrder": "storeOrder",
        "clearCart": "cart/clearCartData"
    }),
    async submitOrder() {
        this.$v.$touch();
        if (!this.$v.$invalid) {
            let order = await this.storeOrder(this.order);
            this.clearCart();
            this.$router.push(`~/thanks/${order}`);
        }
    }
}
</script>

```

Zgrupowałem wszystkie właściwości danych w ramach obiektu order, a także dodałem pola email, address, city i zip. W metodzie submitOrder sprawdzam właściwość `$v.$isValid`, która z kolei weryfikuje poprawność wszystkich validatorów. Jeśli wszystko jest w porządku, wywołuję akcję storeOrder, aby wysłać zamówienie do usługi sieciowej, wyczyścić koszyk zakupowy i przejść pod adres /thanks, gdzie wyświetlę komponent zdefiniowany w listingu 6.18. Rysunek 6.10 przedstawia sekwencję składania zamówienia.



Rysunek 6.10. Tworzenie zamówienia w procesie jego składania

Po zakończeniu procesu składania zamówienia możesz kliknąć przycisk *Powrót* i rozpocząć nowe zakupy. Nie zobaczyś teraz, że zamówienie zostało złożone, ponieważ dostęp do danych jest ograniczony. Niezbędne mechanizmy zostaną jednak dodane już w rozdziale 7.

Podsumowanie

W tym rozdziale dodałem obsługę trasowania adresów URL do naszej aplikacji i skorzystałem z niego, aby umożliwić przechodzenie pomiędzy różnymi obszarami aplikacji. Dodałem także koszyk zakupowy, który pozwala użytkownikom wybierać produkty w sklepie. Dzięki trwałości koszyka ręczne przejście pomiędzy adresami w przeglądarce lub jej odświeżenie nie spowoduje utraty danych. Stworzyłem także proces składania zamówienia, który weryfikuje dane dostarczone przez użytkownika i przesyła zamówienie do usługi sieciowej w celu jego przechowania. W kolejnym rozdziale zwiększę ilość danych, z którymi musi radzić sobie aplikacja, a także rozpocznę pracę nad funkcjami administracyjnymi.

ROZDZIAŁ 7.



Sklep sportowy: skalowanie i administracja

W tym rozdziale kontynuujemy dodawanie funkcji do sklepu sportowego, którego tworzenie rozpoczęliśmy w rozdziale 5. Dodamy obsługę dużych ilości danych, a także zaczniemy tworzyć funkcje administracyjne.

-
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.
-

Przygotowania do rozdziału

Aby przygotować się do tego rozdziału, zwiększę ilość danych, z którymi pracujemy w aplikacji. Skorzystam z pakietu Faker, dodanego do projektu w rozdziale 5., aby wygenerować dużą liczbę obiektów produktów. W tym celu zmieniam zawartość pliku *data.js* (listing 7.1).

Listing 7.1. Generowanie danych w pliku sportsstore/data.js

```
var faker = require("faker");
var data = [];
var categories = ["Sporty wodne", "Piłka nożna", "Szachy", "Bieganie"];
faker.locale = "pl";
faker.seed(100);
for (let i = 1; i <= 500; i++) {
    var category = faker.helpers.randomize(categories);
    data.push({
        id: i,
        name: faker.commerce.productName(),
        category: category,
        description: `${category}: ${faker.lorem.sentence(3)}`,
        price: faker.commerce.price()
    })
}
module.exports = function () {
    return {
        products: data,
        categories: categories,
    }
}
```

```
    orders: []
}
}
```

Pakiet Faker stanowi znakomite narzędzie do generowania danych pseudolosowych podczas tworzenia aplikacji. Dzięki niemu można łatwo znaleźć rozsądnią granicę wydajności aplikacji bez potrzeby ręcznego tworzenia realistycznie wyglądających danych. Pakiet Faker jest opisany szczegółowo pod adresem <http://marak.github.io/faker.js>. Korzystam z niego do generowania losowych nazw, opisów i cen.

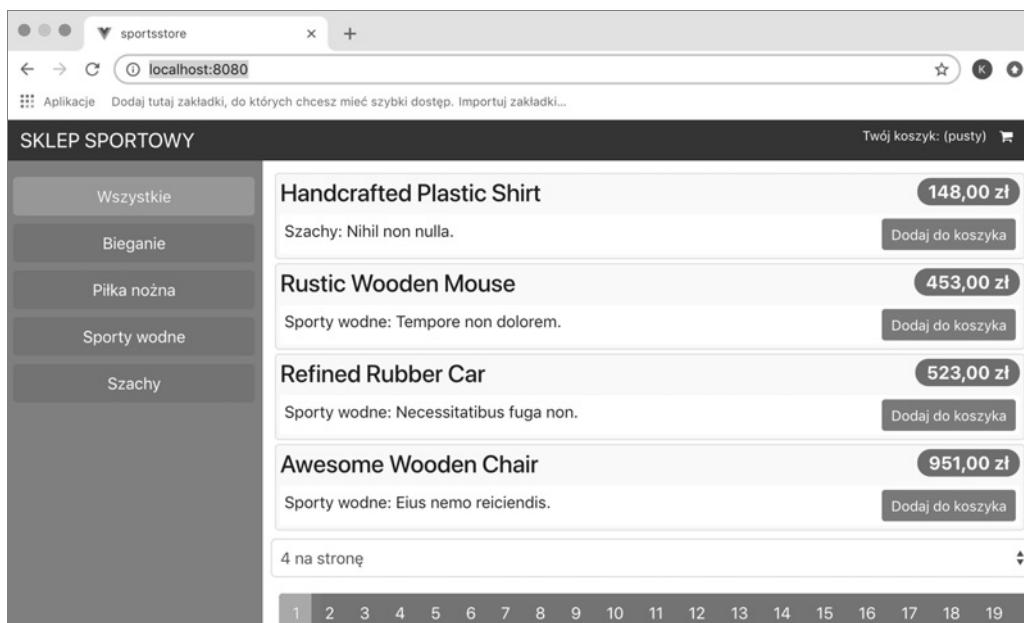
Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj następujące polecenie w katalogu *sportsstore*:

```
npm run json
```

Następnie otwórz drugie okno wiersza poleceń i wykonaj kolejne polecenie w tym samym katalogu, aby uruchomić narzędzia deweloperskie i serwer HTTP:

```
npm run serve
```

Po zakończeniu początkowego procesu budowania otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć treść jak na rysunku 7.1.



Rysunek 7.1. Uruchamianie aplikacji Sklep sportowy

Obsługa dużej ilości danych

Jak widać, po wprowadzeniu tak dużej ilości danych aplikacja potrzebuje pewnych zmian — wiersz przycisków paginacji jest tak długi, że stał się bezużyteczny. W kolejnych podrozdziałach usprawnimy naszą aplikację, aby prezentować dane w bardziej przyjazny sposób, jednocześnie redukując ilość danych pobieranych za jednym razem z serwera.

Usprawnienie stronicowania

Zacznę od rozwiązania najbardziej oczywistego problemu, czyli pokazywania użytkownikowi długiej listy numerów stron, co niezwykle utrudnia nawigację. Aby rozwiązać ten problem, ograniczę listę przycisków, co ułatwi nawigację przy jednoczesnym pozbawieniu użytkownika możliwości przejścia do dowolnej strony (listing 7.2).

Listing 7.2. Usprawnianie nawigacji w pliku src/components/PageControls.vue

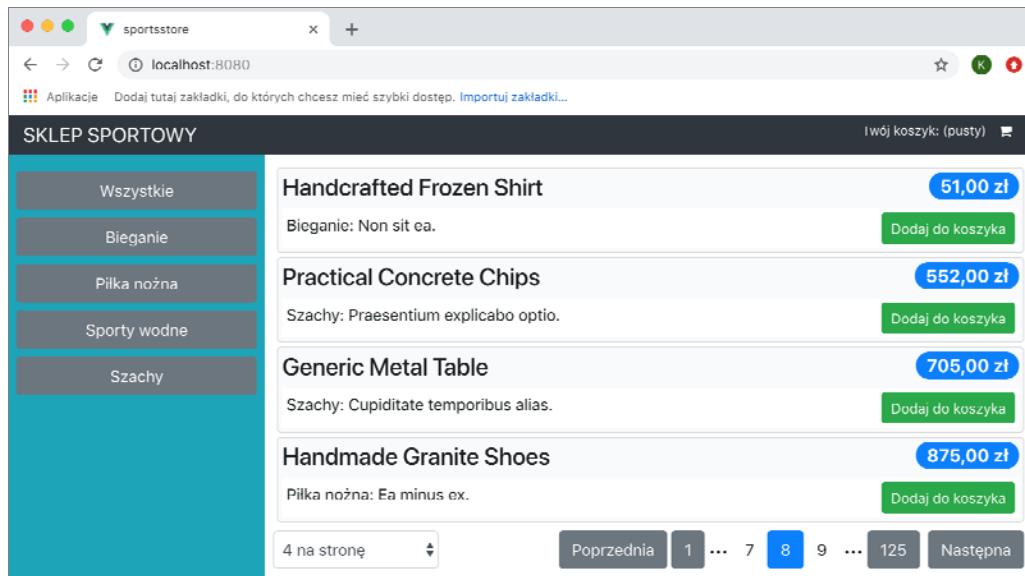
```
<template>
  <div class="row mt-2">
    <div class="col-3 form-group">
      <select class="form-control" v-on:change="changePageSize">
        <option value="4">4 na stronę</option>
        <option value="8">8 na stronę</option>
        <option value="12">12 na stronę</option>
      </select>
    </div>
    <div class="text-right col">
      <button v-bind:disabled="currentPage == 1"
              v-on:click="setCurrentPage(currentPage - 1)"
              class="btn btn-secondary mx-1">Poprzednia</button>
      <span v-if="currentPage > 4">
        <button v-on:click="setCurrentPage(1)"
                class="btn btn-secondary mx-1">1</button>
        <span class="h4"><...></span>
      </span>
      <span class="mx-1">
        <button v-for="i in pageNumbers" v-bind:key="i"
                class="btn btn-secondary"
                v-bind:class="{ 'btn-primary': i == currentPage }"
                v-on:click="setCurrentPage(i)">{ i }</button>
      </span>
      <span v-if="currentPage <= pageCount - 4">
        <span class="h4"><...></span>
        <button v-on:click="setCurrentPage(pageCount)"
                class="btn btn-secondary mx-1">{ pageCount }</button>
      </span>
      <button v-bind:disabled="currentPage == pageCount"
              v-on:click="setCurrentPage(currentPage + 1)"
              class="btn btn-secondary mx-1">Następna</button>
    </div>
  </div>
</template>
<script>
  import { mapState, mapGetters, mapMutations } from "vuex";
  export default {
    computed: {
      ...mapState(["currentPage"]),
      ...mapGetters(["pageCount"]),
      pageNumbers() {
        if (this.pageCount < 4) {
          return [...Array(this.pageCount + 1).keys()].slice(1);
        } else if (this.currentPage <= 4) {
          return [1, 2, 3, 4, 5];
        } else if (this.currentPage > this.pageCount - 4) {
          return [...Array(5).keys()].reverse()
            .map(v => this.pageCount - v);
        }
        const start = Math.max(1, this.currentPage - 4);
        const end = Math.min(this.pageCount, this.currentPage + 4);
        return [...Array(end - start + 1).keys()]
          .map(v => start + v);
      }
    }
  }
</script>
```

```

        } else {
            return [this.currentPage -1, this.currentPage,
                    this.currentPage + 1];
        }
    },
methods: {
    ...mapMutations(["setCurrentPage", "setPageSize"]),
    changePageSize($event) {
        this.setPageSize($event.target.value);
    }
}
}
</script>

```

Nie musimy korzystać z żadnych nowych funkcji Vue.js, aby widok działał zgodnie z naszymi oczekiwaniami. Teraz użytkownik widzi strony: aktualną, poprzednią, następną, a także pierwszą i ostatnią (rysunek 7.2).



Rysunek 7.2. Ograniczanie ustawień stronicowania

Przy opracowywaniu ustawień stronicowania wzorowałem się na Amazonie — tego rodzaju podejście jest jednak znane użytkownikom również z wielu innych stron. Należy przy tym pamiętać, że to podejście jest dostosowane do użytkowników, którzy koncentrują się na kilku pierwszych stronach wyników, bowiem daje do nich łatwiejszy dostęp.

Ograniczanie ilości danych pobieranych przez aplikację

Aplikacja wykonuje jedno żądanie w momencie startu, pobierając wszystkie dane dostępne w usłudze sieciowej. Takie zachowanie uniemożliwia skalowanie, zwłaszcza że większość pobranych danych nie zostanie nigdy pokazana użytkownikowi, ponieważ nigdy on do nich nie dotrze. Aby rozwiązać ten problem, będziemy pobierać dane na żądanie użytkownika, zakładając, że większość użytkowników skorzysta tylko z produktów z pierwszych stron (listing 7.3).

Listing 7.3. Pobieranie danych w pliku src/store/index.js

```

import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
  strict: true,
  modules: { cart: CartModule, orders: OrdersModule },
  state: {
    //products: [],
    categoriesData: [],
    //productsTotal: 0,
    currentPage: 1,
    pageSize: 4,
    currentCategory: "Wszystkie",
    pages: [],
    serverPageCount: 0
  },
  getters: {
    //productsFilteredByCategory: state => state.products
    //filter(p => state.currentCategory == "Wszystkie"
    //|| p.category == state.currentCategory),
    processedProducts: (state) => {
      return state.pages[state.currentPage];
    },
    pageCount: (state) => state.serverPageCount,
    categories: state => ["Wszystkie", ...state.categoriesData]
  },
  mutations: {
    _setCurrentPage(state, page) {
      state.currentPage = page;
    },
    _setPageSize(state, size) {
      state.pageSize = size;
      state.currentPage = 1;
    },
    _setCurrentCategory(state, category) {
      state.currentCategory = category;
      state.currentPage = 1;
    },
    //setData(state, data) {
    //  state.products = data pdata;
    //  state.productsTotal = data pdata.length;
    //  state.categoriesData = data cdata.sort();
    //}
    addPage(state, page) {
      for (let i = 0; i < page.pageCount; i++) {
        Vue.set(state.pages, page.number + i,
          page.data.slice(i * state.pageSize,
            (i * state.pageSize) + state.pageSize));
      }
    },
  },
});

```

```
        clearPages(state) {
            state.pages.splice(0, state.pages.length);
        },
        setCategories(state, categories) {
            state.categoriesData = categories;
        },
        setPageCount(state, count) {
            state.serverPageCount = Math.ceil(Number(count) / state.pageSize);
        },
    },
    actions: {
        async getData(context) {
            await context.dispatch("getPage", 2);
            context.commit("setCategories", (await Axios.get(categoriesUrl)).data);
        },
        async getPage(context, getPageCount = 1) {
            let url = `${productsUrl}?_page=${context.state.currentPage}` +
                `&_limit=${context.state.pageSize * getPageCount}`;
            if (context.state.currentCategory != "Wszystkie") {
                url += `&category=${context.state.currentCategory}`;
            }
            let response = await Axios.get(url);
            context.commit("setPageCount", response.headers["x-total-count"]);
            context.commit("addPage", { number: context.state.currentPage,
                data: response.data, pageCount: getPageCount});
        },
        setCurrentPage(context, page) {
            context.commit("_setCurrentPage", page);
            if (!context.state.pages[page]) {
                context.dispatch("getPage");
            }
        },
        setPageSize(context, size) {
            context.commit("clearPages");
            context.commit("_setPageSize", size);
            context.dispatch("getPage", 2);
        },
        setCurrentCategory(context, category) {
            context.commit("clearPages");
            context.commit("_setCurrentCategory", category);
            context.dispatch("getPage", 2);
        }
    }
})
```

Wprowadzone zmiany są prostsze, niż się może wydawać. Odzwierciedlają one ścisłe podejście do obsługi magazynu danych w Vuex. Tylko w akcjach możemy wykonywać zadania asynchroniczne, co oznacza, że wszelka aktywność, która zawiera żądanie HTTP, musi być wykonana w ramach akcji, a akcje zmieniają stan za pomocą mutacji. W efekcie tworzę akcję z nazwami, które były używane w mutacjach. Do nazw mutacji z kolei dodaję znaki podkreślenia, dzięki czemu jestem w stanie modyfikować dane w magazynie. Z początku może wydawać się to dziwne, ale przepływ sterowania z akcji poprzez mutacje do stanu to właściwe podejście. Zastosowanie takiego podejścia pomaga w debugowaniu (co objaśniam w rozdziale 20.).

Istotą zmian w listingu 7.3 jest zmiana adresu URL, który jest przesyłany do usługi sieciowej. Poprzednio adres do pobierania produktów był następujący:

```
...  
http://localhost:3500/products  
...
```

Pakiet json-server, za pomocą którego obsługujemy usługę sieciową, zawiera wsparcie dla stronicowania i filtrowania danych, dzięki czemu możemy zastosować adres URL jak poniżej:

```
...
http://localhost:3500/products?_page=3&_limit=4&category=Sporty%20wodne
...
```

Parametry `_page` i `_limit` są używane do pobierania stron danych, a parametr `category` jest używany do pobierania tylko tych obiektów, których właściwość `category` ma wartość `Sporty wodne`. To właśnie taki format adresu URL wprowadzamy do aplikacji *Sklep sportowy*.

Podejście, które zastosowałem w listingu, polega na pobraniu danych tylko ze stron, które zostaną wyświetlone użytkownikom. Aplikacja odnotowuje, które dane zostały pobrane, i wysyła jedynie żądania HTTP w przypadku stron, dla których jeszcze nie zostały pobrane dane. Dane pobrane z serwera zostaną usunięte, gdy użytkownik zmieni rozmiar strony lub kategorię. Oprócz tego w momencie uruchomienia aplikacji pobierane są dwie strony, aby umożliwić użytkownikowi przejście do kolejnej strony bez wykonywania żądania sieciowego.

-
- **Ostrzeżenie** Z pewnością nietrudno wpaść na pomysł zastosowania wyszukanych mechanizmów do zarządzania danymi przechowywanymi w pamięci przeglądarki, aby zminimalizować liczbę żądań HTTP wykonywanych przez aplikację. Wydajna obsługa pamięci podręcznej (ang. *caching*) jest jednak niezwykle trudna w implementacji, a złożoność, którą dodaje ona do projektu, często przewyższa korzyści. W związku z tym zachęcam do skorzystania z prostego podejścia (jak w listingu 7.3), a zarządzanie danymi w bardziej skomplikowany sposób radzę zastosować tylko wtedy, gdy jest się przekonanym, że to potrzebne.
-

Określanie liczby elementów

W przypadku obsługi danych przedstawianych z wykorzystaniem stronicowania trudno jest określić, jak dużo obiektów jest dostępnych. Ta informacja jest niezbędna do wyświetlenia przycisków paginacji. Aby poradzić sobie z tym problemem, pakiet json-server załącza nagłówek `X-Total-Count`, który zawiera liczbę obiektów dostępnych w kolekcji. Za każdym razem, gdy wykonujemy żądanie HTTP, otrzymujemy nagłówek odpowiedzi, dzięki któremu możemy zaktualizować liczbę stron, np.:

```
...
context.commit("setPageCount", response.headers["x-total-count"]);
...
```

Nie wszystkie usługi sieciowe oferują dokładnie taką opcję, ale zazwyczaj jest do dyspozycji analogiczny mechanizm, dzięki któremu wiemy, jak dużo obiektów jest dostępnych.

Dodawanie elementu do tablicy stron

Vue.js i Vuex dobrze radzą sobie ze śledzeniem zmian i zapewnianiem aktualności danych w aplikacji. W większości przypadków dzieje się to automatycznie. Niestety, z uwagi na pewne ograniczenia języka JavaScript, Vue.js wymaga pomocy w przypadku niektórych operacji, np. przy zamianie elementu tablicy. Taka operacja nie wywoła mechanizmu wykrycia zmiany. W związku z tym, gdy otrzymujemy stronę z serwera i dodajemy ją do tablicy, konieczne jest wykonanie metody, którą Vue.js udostępnia nam właśnie w tym celu:

```
...
Vue.set(state.pages, page.number + i, page.data.slice(i * state.pageSize, (i * state.pageSize) +
  ↵state.pageSize));
...
```

Metoda `Vue.set` przyjmuje trzy argumenty: obiekt lub tablicę do zmodyfikowania, właściwość lub indeks do przypisania, a także wartość, która zostanie użyta w przypisaniu. Zastosowanie metody `Vue.set` da nam gwarancję, że zmiana zostanie rozpoznana i obsłużona jako aktualizacja (więcej na ten temat w rozdziale 13.).

Modyfikacja komponentu w celu obsługi akcji

Zmiany, które wprowadziłem w listingu 7.3, wymagają wprowadzenia analogicznych modyfikacji w komponentach, które korzystają z mutacji zastąpionych przez nas akcjami. Listing 7.4 przedstawia niezbędne zmiany w komponencie `CategoryControls.vue`.

Listing 7.4. Zastosowanie akcji w pliku src/components/CategoryControls.vue

```
...
<script>
  import { mapState, mapGetters, mapActions } from "vuex";
  export default {
    computed: {
      ...mapState(["currentCategory"]),
      ...mapGetters(["categories"])
    },
    methods: {
      ...mapActions(["setCurrentCategory"])
    }
  }
</script>
...
```

Zmiana w komponencie wymaga zamiany wywołania funkcji `mapMutations` na `mapActions`. Obie funkcje dodają metody do komponentu, dzięki czemu nie są konieczne żadne zmiany. W listingu 7.5 aktualizuję komponent `PageControls.vue`.

Listing 7.5. Zastosowanie akcji w pliku src/components/PageControls.vue

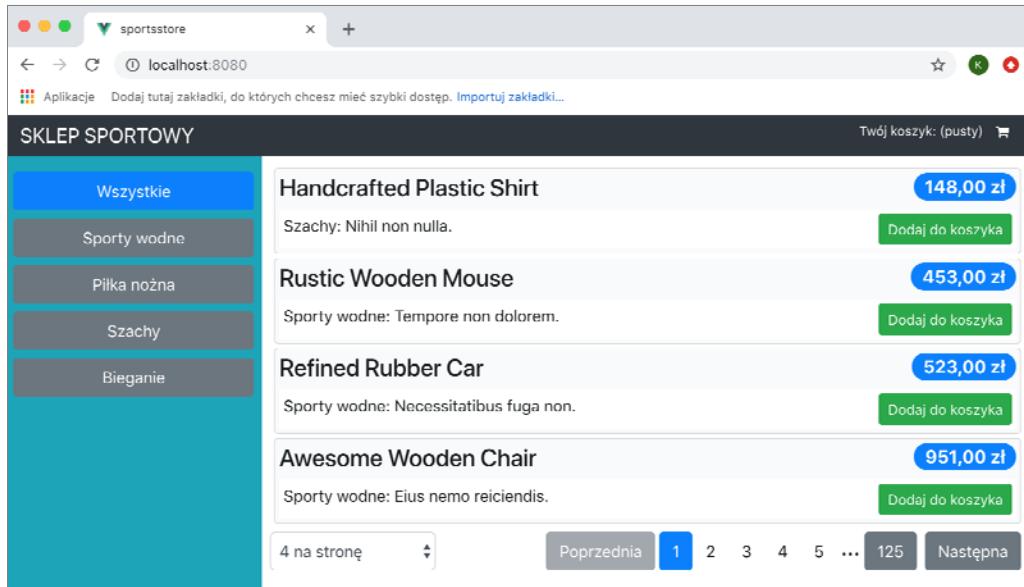
```
...
<script>
  import { mapState, mapGetters, mapActions } from "vuex";
  export default {
    computed: {
      ...mapState(["currentPage"]),
      ...mapGetters(["pageCount"]),
      pageNumbers() {
        if (this.pageCount < 4) {
          return [...Array(this.pageCount + 1).keys()].slice(1);
        } else if (this.currentPage <= 4) {
          return [1, 2, 3, 4, 5];
        } else if (this.currentPage > this.pageCount - 4) {
          return [...Array(5).keys()].reverse()
            .map(v => this.pageCount - v);
        } else {
          return [this.currentPage - 1, this.currentPage,
            this.currentPage + 1];
        }
      },
      methods: {
        ...mapActions(["setCurrentPage", "setPageSize"]),
        changePageSize($event) {
...
```

```

        this.setPageSize($event.target.value);
    }
}
</script>
...

```

Nie ma żadnych zmian w sposobie wyświetlania treści, ale jeśli przejdziesz do narzędzi deweloperskich przeglądarki, na zakładkę *Network*, to zauważysz żądania wysyłane do usługi (rysunek 7.3).



Rysunek 7.3. Pobieranie danych z podziałem na strony

Obsługa wyszukiwania

Kolejną funkcją dodaną do naszej aplikacji będzie mechanizm wyszukiwania, który zawsze przydaje się w przypadku dużych, trudnych do przejrzenia zbiorów danych. W listingu 7.6 zaktualizowałem magazyn danych, dzięki czemu tekst do wyszukania będzie dołączany do adresu URL używanego w żądaniu, z wykorzystaniem funkcji dostarczonej przez pakiet json-server, z którego korzystam do uruchamiania usługi sieciowej.

Listing 7.6. Dodawanie obsługi wyszukiwania w pliku src/store/index.js

```

import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";

Vue.use(Vuex);

const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;

```

```

export default new Vuex.Store({
    strict: true,
    modules: { cart: CartModule, orders: OrdersModule },
    state: {
        categoriesData: [],
        currentPage: 1,
        pageSize: 4,
        currentCategory: "Wszystkie",
        pages: [],
        serverPageCount: 0,
        searchTerm: "",
        showSearch: false
    },
    getters: {
        processedProducts: (state) => {
            return state.pages[state.currentPage];
        },
        pageCount: (state) => state.serverPageCount,
        categories: state => ["Wszystkie", ...state.categoriesData]
    },
    mutations: {
        _setCurrentPage(state, page) {
            state.currentPage = page;
        },
        _setPageSize(state, size) {
            state.pageSize = size;
            state.currentPage = 1;
        },
        _setCurrentCategory(state, category) {
            state.currentCategory = category;
            state.currentPage = 1;
        },
        addPage(state, page) {
            for (let i = 0; i < page.pageCount; i++) {
                Vue.set(state.pages, page.number + i,
                    page.data.slice(i * state.pageSize,
                        (i * state.pageSize) + state.pageSize));
            }
        },
        clearPages(state) {
            state.pages.splice(0, state.pages.length);
        },
        setCategories(state, categories) {
            state.categoriesData = categories;
        },
        setPageCount(state, count) {
            state.serverPageCount = Math.ceil(Number(count) / state.pageSize);
        },
        setShowSearch(state, show) {
            state.showSearch = show;
        },
        setSearchTerm(state, term) {
            state.searchTerm = term;
            state.currentPage = 1;
        },
    },
    actions: {
        async getData(context) {

```

```

        await context.dispatch("getPage", 2);
        context.commit("setCategories", (await Axios.get(categoriesUrl)).data);
    },
    async getPage(context, getPageCount = 1) {
        let url = `${productsUrl}?_page=${context.state.currentPage}`
            + `&_limit=${context.state.pageSize * getPageCount}`;
        if (context.state.currentCategory != "Wszystkie") {
            url += `&category=${context.state.currentCategory}`;
        }
        if (context.state.searchTerm != "") {
            url += `&q=${context.state.searchTerm}`;
        }
        let response = await Axios.get(url);
        context.commit("setPageCount", response.headers["x-total-count"]);
        context.commit("addPage", { number: context.state.currentPage,
            data: response.data, pageCount: getPageCount });
    },
    setCurrentPage(context, page) {
        context.commit("_setCurrentPage", page);
        if (!context.state.pages[page]) {
            context.dispatch("getPage");
        }
    },
    setPageSize(context, size) {
        context.commit("clearPages");
        context.commit("_setPageSize", size);
        context.dispatch("getPage", 2);
    },
    setCurrentCategory(context, category) {
        context.commit("clearPages");
        context.commit("_setCurrentCategory", category);
        context.dispatch("getPage", 2);
    },
    search(context, term) {
        context.commit("setSearchTerm", term);
        context.commit("clearPages");
        context.dispatch("getPage", 2);
    },
    clearSearchTerm(context) {
        context.commit("setSearchTerm", "");
        context.commit("clearPages");
        context.dispatch("getPage", 2);
    }
}
})

```

Do prawidłowego działania mechanizmu wyszukiwania konieczne są dwie właściwości stanu: `showSearch`, która określa, czy użytkownik widzi okienko wyszukiwania, i `searchTerm`, która definiuje zapytanie do wyszukania. Pakiet json-server wspiera przeszukiwanie za pomocą parametru `q`, co oznacza, że aplikacja będzie generować adresy URL w następującej postaci:

```
...
http://localhost:3500/products?_page=3&_limit=4&category=Pilka%20no%C5%82na&q=samoch%C3%B3d
...
```

Ten adres URL pobiera trzecią stronę z czterema elementami, ograniczoną do kategorii *Pilka nożna*, elementów zawierających słowo *samochód*. Aby udostępnić użytkownikowi mechanizm wyszukiwania, dodaję plik `src/components/Search.vue` (listing 7.7).

Listing 7.7. Zawartość pliku *src/components/Search.vue*

```
<template>
  <div v-if="showSearch" class="row my-2">
    <label class="col-2 col-form-label text-right">Szukaj:</label>
    <input class="col form-control"
      v-bind:value="searchTerm" v-on:input="doSearch"
      placeholder="Czego szukasz?" />
    <button class="col-1 btn btn-sm btn-secondary mx-4"
      v-on:click="handleClose">
      Zamknij
    </button>
  </div>
</template>
<script>
import { mapMutations, mapState, mapActions } from "vuex";
export default {
  computed: {
    ...mapState(["showSearch", "searchTerm"])
  },
  methods: {
    ...mapMutations(["setShowSearch"]),
    ...mapActions(["clearSearchTerm", "search"]),
    handleClose() {
      this.clearSearchTerm();
      this.setShowSearch(false);
    },
    doSearch($event) {
      this.search($event.target.value);
    }
  }
}
</script>
```

Komponent wyświetla element `input` użytkownikowi, dzięki czemu jest on w stanie wprowadzić zapytanie. Dyrektywa `v-on` reaguje na każdą zmianę zawartości, wywołując operację wyszukiwania. Aby włączyć ów mechanizm w aplikacji, muszę wprowadzić zmiany w komponencie `Store` (listing 7.8).

Listing 7.8. Włączanie wyszukiwania w pliku *src/components/Store.vue*

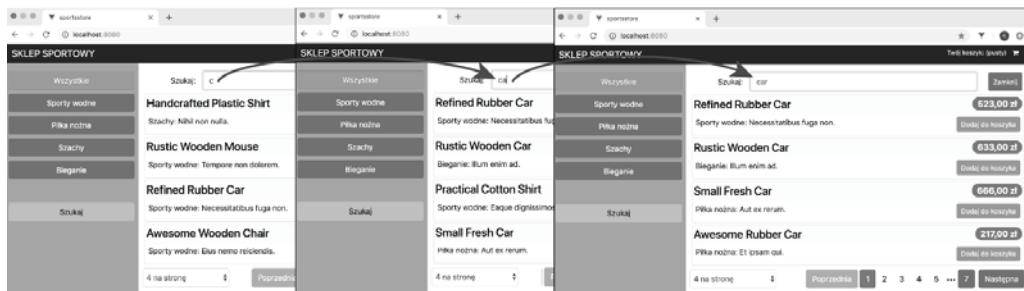
```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col bg-dark text-white">
        <a class="navbar-brand">SKLEP SPORTOWY</a>
        <cart-summary />
      </div>
    </div>
    <div class="row">
      <div class="col-3 bg-info p-2">
        <CategoryControls class="mb-5" />
        <button class="btn btn-block btn-warning mt-5"
          v-on:click="setShowSearch(true)">
          Szukaj
        </button>
      </div>
      <div class="col-9 p-2">
        <Search />
        <ProductList />
      </div>
    </div>
  </div>
```

```

        </div>
    </div>
</div>
</template>
<script>
    import ProductList from "./ProductList";
    import CategoryControls from "./CategoryControls"; import CartSummary from "./CartSummary";
    import { mapMutations } from "vuex";
    import Search from "./Search";
    export default {
        components: { ProductList, CategoryControls, CartSummary, Search },
        methods: {
            ...mapMutations(["setShowSearch"])
        }
    }
</script>

```

Przycisk do pokazania funkcji wyszukiwania został umieszczony obok listy kategorii, a komponent utworzony w listingu 7.7 został zarejestrowany. Dzięki temu użytkownik może kliknąć przycisk *Szukaj* i wprowadzić tekst do wyszukania, jak widać na rysunku 7.4. Wyniki są prezentowane w formie stron i przefiltrowane według kategorii.



Rysunek 7.4. Proces wyszukiwania

Praca nad funkcjami administracyjnymi

Każda aplikacja, która przedstawia treści użytkownikowi, wymaga pewnego zakresu prac administracyjnych. W przypadku naszego sklepu oznacza to konieczność zarządzania katalogiem produktów i zamówieniami, które złożyli klienci. W kolejnych punktach dodam kilka kluczowych funkcji administracyjnych — zacznę od uwierzytelniania, a później dodam kolejne istotne dla administratora mechanizmy.

Implementacja uwierzytelniania

Efektem uwierzytelnienia w REST-owej usłudze sieciowej jest token sieciowy JSON (JSON Web Token, JWT). Token ten jest zwracany przez serwer i musi być dołączony do wszystkich kolejnych żądań, aby udowodnić, że dany klient może wykonać chronione operacje. Więcej na temat specyfikacji JWT znajdziesz na stronie <https://tools.ietf.org/html/rfc7519>. Na potrzeby naszej aplikacji musisz wiedzieć, że uwierzytelnia ona użytkownika, wysyłając żądanie POST na adres /login i załączając obiekt sformatowany jako dokument JSON w treści żądania. Dokument zawiera nazwę użytkownika i hasło. W naszym kodzie uwierzytelniania w rozdziale 5. jest dostępny tylko jeden zestaw danych uwierzytelniających, które przedstawiam w tabeli 7.1.

Tabela 7.1. Dane uwierzytelniające obsługiwane przez REST-ową usługę sieciową.

Nazwa użytkownika	Hastō
admin	secret

Jak zaznaczyłem w rozdziale 5., nie powinieneś zaszywać w kodzie danych uwierzytelniających. W tym przypadku nie stanowi to jednak problemu.

Jeśli na adres /login zostaną przesłane prawidłowe dane uwierzytelniające, odpowiedź z usługi RESTowej będzie podobna do poniższej:

```
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRTaW4iLCJ1eHBpcmVz
  &SW4i0iIxaCISImIhdCI6MTQ30Dk1NjI1Mn0.1JaDDrSu-bHBtdWrz0312p_DG5tKypGv6cANg0yz1g8"
}
```

Właściwość success opisuje wynik operacji uwierzytelniania, a właściwość token zawiera JWT, który powinien być dołączony w kolejnych żądaniach w nagłówku Authorization, zgodnie z poniższym formatem:

```
Authorization: Bearer<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRTaW4iLCJ1eHBpcmVz
  &SW4i0iIxaCISImIhdCI6MTQ30Dk1NjI1Mn0.1JaDDrSu-bHBtdWrz0312p_DG5tKypGv6cANg0yz1g8>
```

Jeśli serwer otrzyma nieprawidłowe dane uwierzytelniające, obiekt JSON będzie zawierał jedynie właściwość success ustawioną na false:

```
{
  "success": false
}
```

Skonfigurowałem tokeny JWT zwrócone przez serwer tak, aby wygasły po upływie jednej godziny.

Rozszerzanie magazynu danych

Aby uwzględnić nagłówek uwierzytelniania w żądaniach HTTP, dodałem plik *auth.js* do katalogu *src/store* (listing 7.9).

Listing 7.9. Zawartość pliku *src/store/auth.js*

```
import Axios from "axios";
const loginUrl = "http://localhost:3500/login";
export default {
  state: {
    authenticated: false,
    jwt: null
  },
  getters: {
    authenticatedAxios(state) {
      return Axios.create({
        headers: {
          "Authorization": `Bearer<${state.jwt}>`
        }
      });
    }
  },
  mutations: {
```

```

        setAuthenticated(state, header) {
            state.jwt = header;
            state.authenticated = true;
        },
        clearAuthentication(state) {
            state.authenticated = false;
            state.jwt = null;
        }
    },
    actions: {
        async authenticate(context, credentials) {
            let response = await Axios.post(loginUrl, credentials);
            if (response.data.success == true) {
                context.commit("setAuthenticated", response.data.token);
            }
        }
    }
}
}

```

Akcja uwierzytelniania korzysta z biblioteki Axios w celu wysłania żądania HTTP POST pod adres /login, a także ustawia właściwości stanu authenticated i jwt, jeśli żądanie zakończyło się sukcesem. Metoda create biblioteki Axios powoduje skonfigurowanie obiektu, który jest używany do wykonywania żądań. Korzystam z tej funkcji w getterze authenticatedAxios, aby dostarczyć obiekt Axios z właściwie ustawionym nagłówkiem Authorization we wszystkich żądaniach. W listingu 7.10 dodaję nowy moduł do magazynu danych.

Listing 7.10. Dodawanie modułu w pliku src/store/index.js

```

import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";
import AuthModule from "./auth";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
    strict: true,
    modules: { cart: CartModule, orders: OrdersModule, auth: AuthModule },
    //...funkcje magazynu danych zostały pominięte...
})

```

Dodawanie komponentów administracyjnych

Aby skorzystać z uwierzytelniania, muszę dodać dwa komponenty. Jeden z nich poprosi użytkownika o dane uwierzytelniające, a drugi będzie wyświetlany po skutecznym uwierzytelnianiu i będzie przedstawiać funkcje administracyjne. Aby oddzielić te funkcje od reszty aplikacji, utworzę katalog src/components/admin, do którego dodam plik Admin.vue (listing 7.11).

Listing 7.11. Zawartość pliku src/components/admin/Admin.vue

```

<template>
    <div class="bg-danger text-white text-center h4 p-2">Panel administracyjny</div>
</template>

```

Przedstawiony listing zawiera jedynie treść zastępczą, aby można było pokazać cokolwiek w czasie, gdy będę konfigurował resztę aplikacji. Niebawem ta treść zostanie zastąpiona przez prawdziwe funkcje administracyjne. Aby poprosić użytkownika o dane uwierzytelniające, dodaję plik `src/components/admin/Authentication.vue` (listing 7.12).

Listing 7.12. Zawartość pliku `src/components/admin/Authentication.vue`

```
<template>
<div class="m-2">
    <h4 class="bg-primary text-white text-center p-2">
        Panel administracyjny sklepu sportowego
    </h4>
    <h4 v-if="showFailureMessage" class="bg-danger text-white text-center p-2 my-2">
        Uwierzytelnianie nie powiodło się. Spróbuj ponownie.
    </h4>
    <div class="form-group">
        <label>Nazwa użytkownika</label>
        <input class="form-control" v-model="$v.username.$model">
        <validation-error v-bind:validation="$v.username" />
    </div>
    <div class="form-group">
        <label>Hasło</label>
        <input type="password" class="form-control" v-model="$v.password.$model">
        <validation-error v-bind:validation="$v.password" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" v-on:click="handleAuth">Zaloguj</button>
    </div>
</div>
</template>
<script>
import { required } from "vuelidate/lib/validators";
import { mapActions, mapState } from "vuex";
import ValidationError from "../ValidationError";
export default {
    components: { ValidationError },
    data: function() {
        return {
            username: "admin",
            password: "secret",
            showFailureMessage: false,
        }
    },
    computed: {
        ...mapState({authenticated: state => state.auth.authenticated })
    },
    validations: {
        username: { required },
        password: { required }
    },
    methods: {
        ...mapActions(["authenticate"]),
        async handleAuth() {
            this.$v.$touch();
            if (!this.$v.$invalid) {
                await this.authenticate({ name: this.username,

```

```

        password: this.password });
    if (this.authenticated) {
      this.$router.push("/admin");
    } else {
      this.showFailureMessage = true;
    }
  }
}
</script>
```

- **Uwaga** Aby uprościć proces tworzenia oprogramowania, w listingu 7.12 ustawiam nazwę użytkownika i hasło na dane uwierzytelniające z tabeli 7.1. W miarę tworzenia dalszej treści aplikacji będziesz często ponownie się uwierzytelniać i w związku z tym ponowne wprowadzanie danych będzie niezwykle frustrujące. Z tego względu dane są zaszyte w kodzie. W rozdziale 8. usunę te wartości z komponentu w ramach przygotowań do wdrożenia.

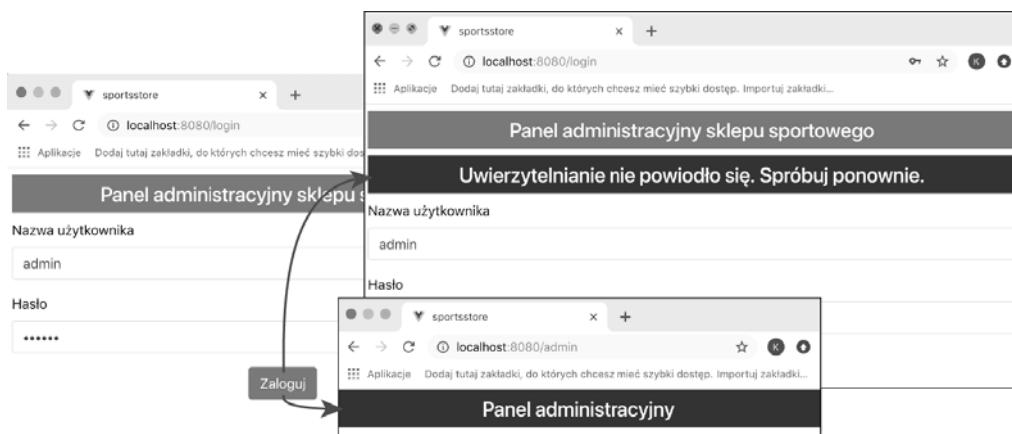
Komponent przedstawia dwa elementy typu `input` i przycisk, który wykonuje uwierzytelnianie za pomocą funkcji magazynu danych z listingu 7.9. Mechanizmy validacji danych zapewniają, że użytkownik wprowadzi wartości w odpowiednie pola. Jeśli uwierzytelnianie nie powiedzie się, zostanie wyświetlona wiadomość z ostrzeżeniem. W przeciwnym razie system trasowania przeniesie użytkownika do strony `/admin`. Aby dodać obsługę nowych komponentów do konfiguracji systemu trasowania, musimy dodać trasy z listingu 7.13.

Listing 7.13. Dodawanie tras do pliku src/router/index.js

```

import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
import Authentication from "../components/admin/Authentication";
import Admin from "../components/admin/Admin";
Vue.use(VueRouter);
export default new VueRouter({
  mode: "history",
  routes: [
    { path: "/", component: Store },
    { path: "/cart", component: ShoppingCart },
    { path: "/checkout", component: Checkout },
    { path: "/thanks/:id", component: OrderThanks },
    { path: "/login", component: Authentication },
    { path: "/admin", component: Admin },
    { path: "**", redirect: "/" }
  ]
})
```

Aby sprawdzić proces uwierzytelniania, przejdź pod adres `http://localhost:8080/login`, wprowadź nazwę użytkownika i hasło, a następnie kliknij *Zaloguj*. Jeśli wprowadzisz dane z tabeli 7.1, uwierzytelnianie powiedzie się i zobaczysz treść zastępczą panelu administracyjnego. Jeśli skorzystasz z innych danych, uwierzytelnianie się nie powiedzie i zobaczysz błąd. Oba wyniki pokazano na rysunku 7.5.



Rysunek 7.5. Uwierzytelnianie użytkownika

Dodawanie strażnika trasy

Obecnie nic nie powstrzyma użytkownika przed przejściem pod adres /admin i tym samym — ominięciem procesu uwierzytelniania. Aby się przed tym uchronić, musimy dodać **strażnika trasy** (ang. *route guard*). Jest to funkcja wywoływana w momencie, gdy następuje zmiana trasy. Funkcja ta potrafi powstrzymać przed zmianą lub zezwolić na jej zajście. Pakiet Vue Router dostarcza szeroki zakres opcji konfiguracyjnych strażników tras, opisanych w rozdziale 24. W tym rozdziale potrzebuję jedynie strażnika, który ochroni trasę /admin i tym samym powstrzyma przejście pod ten adres (listing 7.14).

Listing 7.14. Dodawanie strażnika trasy w pliku src/router/index.js

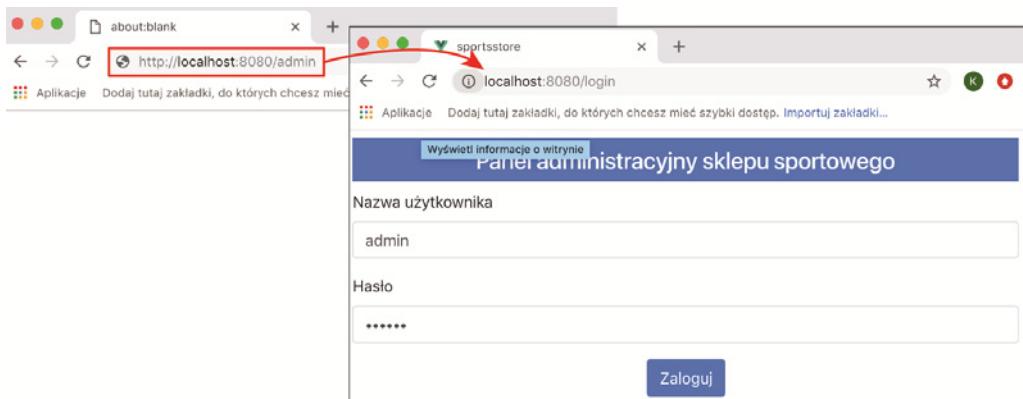
```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
import Authentication from "../components/admin/Authentication";
import Admin from "../components/admin/Admin";
import dataStore from "../store";
Vue.use(VueRouter);
export default new VueRouter({
    mode: "history",
    routes: [
        { path: "/", component: Store },
        { path: "/cart", component: ShoppingCart },
        { path: "/checkout", component: Checkout},
        { path: "/thanks/:id", component: OrderThanks},
        { path: "/login", component: Authentication },
        { path: "/admin", component: Admin,
            beforeEnter(to, from, next) {
                if (dataStore.state.auth.authenticated) {
                    next();
                } else {
                    next("/login");
                }
            }
        },
    ],
},
```

```

        { path: "**", redirect: "/" }
    ]
})

```

Funkcja beforeEnter jest wywoływana w momencie przejścia pod adres /admin. Funkcja ta sprawdza, czy użytkownik jest zalogowany. Strażnicy nawigacji działają dzięki zastosowaniu funkcji next, przekazywanej jako parametr. Próba zmiany adresu jest akceptowana, jeśli funkcja next zostanie wywołana bez argumentów. W przeciwnym razie (gdy przekażemy do tej funkcji argument), zmiana zostanie powstrzymana — użytkownik trafi pod adres /login z prośbą o podanie danych uwierzytelniających. Aby się o tym przekonać, przejdź pod adres <http://localhost:8080/admin>. Zobaczysz, że strażnik trasy przekieruje przeglądarkę pod adres /login (rysunek 7.6).



Rysunek 7.6. Efekt zastosowania strażnika trasy

Zwróc uwagę, że skorzystałem z instrukcji import, aby uzyskać dostęp do magazynu danych. Funkcja pomocnicza mapState, z której korzystałem w poprzednich przykładach, a także właściwość \$store, która może być używana w celu bezpośredniego dostępu do magazynu danych, działają jedynie w komponentach. Nie są dostępne w pozostałych elementach aplikacji, takich jak konfiguracja trasowania. Instrukcja import daje dostęp do magazynu danych, za pomocą którego mogę odczytać wartość właściwości state, wymaganej przez strażnika trasy do określenia, czy użytkownik jest zalogowany.

Dodawanie struktury komponentu administracyjnego

Po dodaniu funkcji uwierzytelniających możemy zająć się głównymi funkcjami panelu administratora. Administrator musi móc zarządzać produktami, które ogląda użytkownik, a także być w stanie zarządzać zamówieniami i oznaczać je jako wysłane. Rozpoczniemy od utworzenia komponentów z treścią zastępczą dla każdej z tych funkcji. Utworzymy w ten sposób strukturę, z którą użytkownik będzie w stanie się zapoznać, a następnie zaimplementujemy wszystkie funkcje w praktyce. Najpierw dodaję plik *ProductAdmin.vue* do katalogu *src/components/admin* (listing 7.15).

Listing 7.15. Zawartość pliku src/components/admin/ProductAdmin.vue

```

<template>
  <div class="bg-danger text-white text-center h4 p-2">
    Zarządzanie produktami
  </div>
</template>

```

Następnie dodaję plik *OrderAdmin.vue* (listing 7.16).

Listing 7.16. Zawartość pliku *src/components/admin/OrderAdmin.vue*

```
<template>
  <div class="bg-danger text-white text-center h4 p-2">
    Zarządzanie zamówieniami
  </div>
</template>
```

Aby administrator mógł zapoznać się z tymi komponentami, zamieniam treść komponentu Admin na tę przedstawioną w listingu 7.17.

Listing 7.17. Wyświetlanie nowych komponentów w pliku *src/components/admin/Admin.vue*

```
<template>
  <div class="container-fluid">
    <div class="row">
      <div class="col bg-secondary text-white">
        <a class="navbar-brand">Panel administracyjny</a>
      </div>
    </div>
    <div class="row">
      <div class="col-3 bg-secondary p-2">
        <router-link to="/admin/products" class="btn btn-block btn-primary"
          &gt;active-class="active">
          Produkty
        </router-link>
        <router-link to="/admin/orders" class="btn btn-block btn-primary"
          &gt;active-class="active">
          Zamówienia
        </router-link>
      </div>
      <div class="col-9 p-2">
        <router-view />
      </div>
    </div>
  </div>
</template>
```

Aplikacje mogą zawierać wiele elementów router-view. W tym miejscu skorzystałem z jednego, aby móc przełączać się pomiędzy komponentami zarządzania produktami a zamówieniami za pomocą tras, które skonfiguruje niebawem. Aby wybrać komponent, korzystam z elementów router-link, które są formatowane jako przyciski. Aby wskazać przycisk, który reprezentuje aktualnie przedstawianą stronę, korzystam z atrybutu active-class. Atrybut ten określa klasę, która zostanie dodana do elementu, gdy trasa określona za pomocą atrybutu będzie aktywna (rozdział 23.).

Aby skonfigurować element router-view, dodaję trasy z listingu 7.18 do konfiguracji trasowania aplikacji.

Listing 7.18. Dodawanie tras do pliku *src/router/index.js*

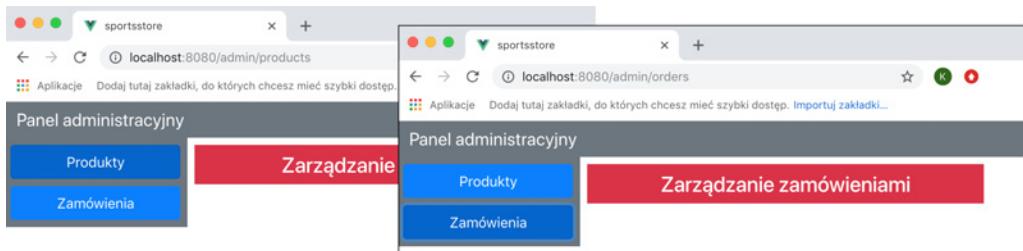
```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
import Authentication from "../components/admin/Admin";
import Admin from "../components/admin/Admin";
import ProductAdmin from "../components/admin/ProductAdmin";
import OrderAdmin from "../components/admin/OrderAdmin";
```

```

import dataStore from "../store";
Vue.use(VueRouter);
export default new VueRouter({
    mode: "history",
    routes: [
        { path: "/", component: Store },
        { path: "/cart", component: ShoppingCart },
        { path: "/checkout", component: Checkout },
        { path: "/thanks/:id", component: OrderThanks },
        { path: "/login", component: Authentication },
        { path: "/admin", component: Admin,
            beforeEnter(to, from, next) {
                if (dataStore.state.auth.authenticated) {
                    next();
                } else {
                    next("/login");
                }
            },
            children: [
                { path: "products", component: ProductAdmin },
                { path: "orders", component: OrderAdmin },
                { path: "", redirect: "/admin/products" }
            ]
        },
        { path: "**", redirect: "/" }
    ]
})

```

Nowe fragmenty w kodzie korzystają z funkcji **tras-dziecka** (ang. *child route*), opisywanej przeze mnie w rozdziale 23. Funkcja ta pozwala na zagnieżdżanie elementów router-view. W rezultacie przejście pod adres /admin/products spowoduje wyświetlenie komponentów Admin i ProductAdmin, zaś przejście pod adres /admin/orders zamiast drugiego wyświetli komponent OrderAdmin (rysunek 7.7). Dodatkowo korzystamy z tras „ratunkowej”, która przekieruje z adresu /admin na /admin/products.



Rysunek 7.7. Zastosowanie tras-dzieci i zagnieżdżonych elementów router-view

Implementacja zarządzania zamówieniami

Aby umożliwić zarządzanie zamówieniami, muszę przedstawić listę obiektów otrzymanych z serwera i umożliwić oznaczanie ich jako wysłane. Na początku muszę rozszerzyć magazyn danych (listing 7.19).

Listing 7.19. Dodawanie funkcji w pliku src/store/orders.js

```

import Axios from "axios";
import Vue from "vue";
const ORDERS_URL = "http://localhost:3500/orders";

```

```

export default {
  state: {
    orders: []
  },
  mutations: {
    setOrders(state, data) {
      state.orders = data;
    },
    changeOrderShipped(state, order) {
      Vue.set(order, "shipped",
        order.shipped == null || !order.shipped ? true : false);
    }
  },
  actions: {
    async storeOrder(context, order) {
      order.cartLines = context.rootState.cart.lines;
      return (await Axios.post(ORDERS_URL, order)).data.id;
    },
    async getOrders(context) {
      context.commit("setOrders",
        (await context.rootGetters.authenticatedAxios.get(ORDERS_URL)).data);
    },
    async updateOrder(context, order) {
      context.commit("changeOrderShipped", order);
      await context.rootGetters.authenticatedAxios.put(`/${ORDERS_URL}/${order.id}`,
order);
    }
  }
}

```

Wiesz już, jak korzystać z Vue.js w celu wyświetlania danych w formie stronicowanej, pobieranych na żądanie. W związku z tym zarządzanie zamówieniami odbywa się niezwykle prosto: pobieram wszystkie dane za pomocą akcji getOrders, a modyfikacja zamówienia odbywa się za pomocą akcji updateOrder.

Aby wyświetlić listę zamówień i umożliwić oznaczanie ich jako wysłane, zamienięm treść komponentu OrderAdmin na tę z listingu 7.20.

Listing 7.20. Zarządzanie zamówieniami w pliku src/components/admin/OrderAdmin.vue

```

<template>
  <div>
    <h4 class="bg-info text-white text-center p-2">Zamówienia</h4>
    <div class="form-group text-center">
      <input class="form-check-input" type="checkbox" v-model="showShipped" />
      <label class="form-check-label">Pokaż wysłane zamówienia</label>
    </div>
    <table class="table table-sm table-bordered">
      <thead>
        <tr>
          <th>ID</th><th>Imię</th><th>Miasto, kod pocztowy</th>
          <th class="text-right">Suma łączna</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        <tr v-if="displayOrders.length == 0">
          <td colspan="5">Nie ma zamówień</td>
        </tr>
        <tr v-for="o in displayOrders" v-bind:key="o.id">

```

```

        <td>{{ o.id }}</td>
        <td>{{ o.name }}</td>
        <td>{{ `${o.city}, ${o.zip}` }}</td>
        <td class="text-right">{{ getTotal(o) | currency }}</td>
        <td class="text-center">
            <button class="btn btn-sm btn-danger"
                v-on:click="shipOrder(o)">
                {{ o.shipped ? 'Nie wysłano' : 'Wysłano' }}
            </button>
        </td>
    </tr>
</tbody>
</table>
</div>
</template>
<script>
import { mapState, mapActions, mapMutations } from "vuex";
export default {
    data: function() {
        return {
            showShipped: false
        }
    },
    computed: {
        ...mapState({ orders: state => state.orders.orders}),
        displayOrders() {
            return this.showShipped ? this.orders
                : this.orders.filter(o => o.shipped != true);
        }
    },
    methods: {
        ...mapMutations(["changeOrderShipped"]),
        ...mapActions(["getOrders", "updateOrder"]),
        getTotal(order) {
            if (order.cartLines != null && order.cartLines.length > 0) {
                return order.cartLines.reduce((total, line) =>
                    total + (line.quantity * line.product.price), 0)
            } else {
                return 0;
            }
        },
        shipOrder(order) {
            this.updateOrder(order);
        }
    },
    created() {
        this.getOrders();
    }
}
</script>

```

Komponent wyświetla tabelę zamówień, a każde z nich zawiera przycisk, który pozwala na zmianę stanu wysłania. Aby przekonać się o tym samemu, przejdź pod adres <http://localhost:8080/> i skorzystaj z funkcji sklepu, aby utworzyć jedno lub więcej zamówień. Następnie przejdź pod adres <http://localhost:8080/admin>, uwierzytelnij się i kliknij łącze *Zamówienia*, aby zapoznać się z listą jak na rysunku 7.8.

ID	Imię	Miasto, kod pocztowy	Suma łączna	
1	Jan	Warszawa, 12-345	148,00 zł	Wysłano
2	Janina	Warszawa, 23-456	951,00 zł	Wysłano

Rysunek 7.8. Zarządzanie zamówieniami

Podsumowanie

W tym rozdziale kontynuowałem tworzenie aplikacji *Sklep sportowy*. Pokazałem Ci, jak skorzystać z Vue.js i jego głównych bibliotek, aby obsługiwać dużą ilość danych. Umieściłem także pierwsze funkcje administracyjne, zaczynając od uwierzytelniania i zarządzania zamówieniami. W kolejnym rozdziale uzupełnię funkcje administracyjne i wdrożę gotową aplikację.

ROZDZIAŁ 8.

Sklep sportowy: administrowanie i wdrożenie

W tym rozdziale zakończę prace nad sklepem sportowym — dodam pozostałe funkcje administracyjne i pokażę przygotowania do wdrożenia. Jak sam się przekonasz, przejście od fazy tworzenia oprogramowania do wdrożenia jest całkiem proste.

Przygotowania do rozdziału

W tym rozdziale korzystam z projektu z rozdziału 7. bez żadnych zmian.

-
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.
-

Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj następujące polecenie w katalogu *sportsstore*:

```
npm run json
```

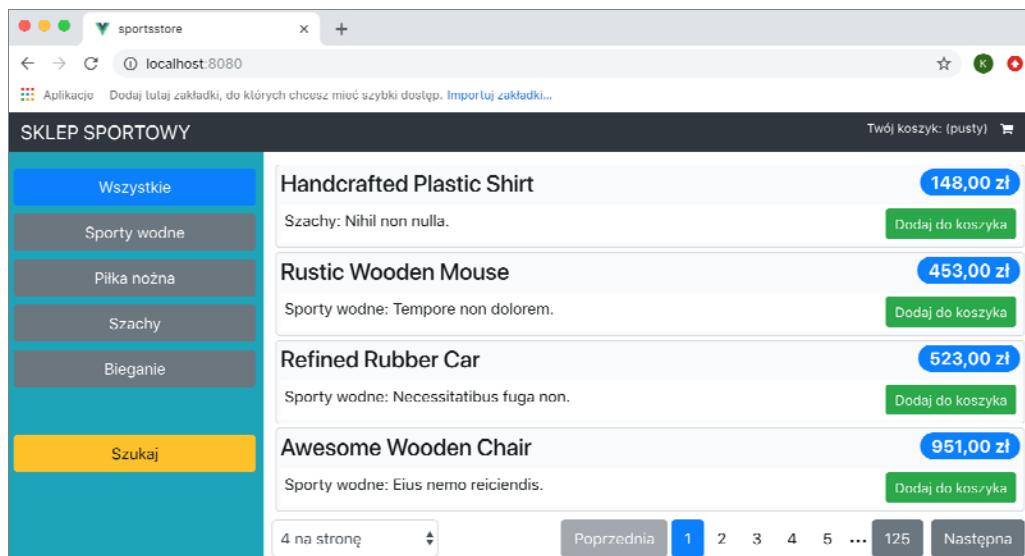
Następnie otwórz drugie okno wiersza poleceń i wykonaj kolejne polecenie w tym samym katalogu, aby uruchomić narzędzia deweloperskie i serwer HTTP:

```
npm run serve
```

Po zakończeniu początkowego procesu budowania otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć treść jak na rysunku 8.1.

Dodawanie funkcji administracyjnych

Aby zakończyć kwestię funkcji administracyjnych, muszę umożliwić w naszej aplikacji tworzenie, edycję i usuwanie produktów. Jak zawsze, zaczniemy od rozszerzenia magazynu danych, aby zadeklarować niezbędne akcje. W ramach akcji będziemy wysyłać żądania HTTP do usługi sieciowej i aktualizować dane produktów jak w listingu 8.1.



Rysunek 8.1. Uruchamianie aplikacji Sklep sportowy

Listing 8.1. Dodawanie funkcji administracyjnych do pliku src/store/index.js

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";
import AuthModule from "./auth";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
    strict: true,
    modules: { cart: CartModule, orders: OrdersModule, auth: AuthModule },
    state: {
        //...właściwości stanu zostały pominięte...
    },
    getters: {
        processedProducts: (state) => {
            return state.pages[state.currentPage];
        },
        pageCount: (state) => state.serverPageCount,
        categories: state => ["Wszystkie", ...state.categoriesData],
        productById:(state) => (id) => {
            return state.pages[state.currentPage].find(p => p.id == id);
        }
    },
    mutations: {
        _setCurrentPage(state, page) {
            state.currentPage = page;
        },
        //...pozostałe mutacje zostały pominięte...
        setSearchTerm(state, term) {

```

```
        state.searchTerm = term;
        state.currentPage = 1;
    },
    _addProduct(state, product) {
        state.pages[state.currentPage].unshift(product);
    },
    _updateProduct(state, product) {
        let page = state.pages[state.currentPage];
        let index = page.findIndex(p => p.id == product.id);
        Vue.set(page, index, product);
    }
},
actions: {
    async getData(context) {
        await context.dispatch("getPage", 2);
        context.commit("setCategories", (await Axios.get(categoriesUrl)).data);
    },
    // ...pozostale akcje zostaly pominięte...
    async addProduct(context, product) {
        let data = (await context.getters.authenticatedAxios.post(productsUrl,
            product)).data;
        product.id = data.id;
        this.commit("_addProduct", product);
    },
    async removeProduct(context, product) {
        await context.getters.authenticatedAxios
            .delete(`${productsUrl}/${product.id}`);
        context.commit("clearPages");
        context.dispatch("getPage", 1);
    },
    async updateProduct(context, product) {
        await context.getters.authenticatedAxios
            .put(`${productsUrl}/${product.id}`, product);
        this.commit("_updateProduct", product);
    }
}
})
```

Akcje dodania, zmiany i usunięcia będą wykonywane przez komponenty. Wiąże się to z wysyłaniem żądań HTTP i wprowadzaniem niezbędnych zmian w danych lokalnych. W momencie zmiany produktu znajduję istniejący obiekt i zamieniam go. W przypadku innych operacji wybieram drogę na skróty. Gdy chcę dodać produkt, wstawiam go na początek aktualnej strony produktów, mimo że w ten sposób rozmiar strony będzie nieprawidłowy. Dzięki temu nie muszę jednak się zastanawiać, na której stronie obiekt zostanie wyświetlony. W przypadku usunięcia obiektu odświeżam dane, dzięki czemu nie muszę ręcznie przetwarzać zawartości strony. Te „skróty” są realizowane w mutacjach `_addProduct` i `_updateProduct`. Mutacje są poprzedzone podkreśleniem, co oznacza, że nie są one przeznaczone do szerszego zastosowania. Dodaję także getter w listingu 8.1, dzięki czemu jestem w stanie znaleźć produkt na bieżącej stronie wyników za pomocą `id`. Tego rodzaju getter przyjmuje parametr i funkcjonuje jak metoda. Oznacza to, że mogę zdefiniować operację, która wyszukuje produkt w magazynie danych, zamiast pobierać wszystkie obiekty ze strony i wykonywać wyszukiwanie w komponencie.

Przedstawianie listy produktów

Aby wyświetlić listę produktów i niezbędne narzędzia do ich tworzenia, edycji i usuwania, zamieniam treść zastępczą na zawartość listingu 8.2.

Listing 8.2. Dodawanie funkcji w pliku *src/components/admin/ProductAdmin.vue*

```

<template>
  <div>
    <router-link to="/admin/products/create" class="btn btn-primary my-2">
      Utwórz produkt
    </router-link>
    <table class="table table-sm table-bordered">
      <thead>
        <th>ID</th><th>Nazwa</th><th>Kategoria</th>
        <th class="text-right">Cena</th><th></th>
      </thead>
      <tbody>
        <tr v-for="p in products" v-bind:key="p.id">
          <td>{{ p.id }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.category }}</td>
          <td class="text-right">{{ p.price | currency }}</td>
          <td class="text-center">
            <button class="btn btn-sm btn-danger mx-1"
                   v-on:click="removeProduct(p)">Usuń</button>
            <button class="btn btn-sm btn-warning mx-1"
                   v-on:click="handleEdit(p)">Edytuj</button>
          </td>
        </tr>
      </tbody>
    </table>
    <page-controls />
  </div>
</template>
<script>
import PageControls from "../PageControls";
import { mapGetters, mapActions } from "vuex";
export default {
  components: { PageControls },
  computed: {
    ...mapGetters({
      products: "processedProducts"
    })
  },
  methods: {
    ...mapActions(["removeProduct"]),
    handleEdit(product) {
      this.$router.push(`/admin/products/edit/${product.id}`);
    }
  }
}
</script>

```

Ten komponent przedstawia tabelę produktów, wypełnioną danymi z modelu za pomocą dyrektywy `v-for`. Każdy wiersz zawiera przyciski *Edytuj* i *Usuń*. Kliknięcie przycisku *Usuń* spowoduje wywołanie akcji `removeProduct`, dodanej do magazynu danych w listingu 8.1. Kliknięcie przycisku *Edytuj* lub *Utwórz produkt* przekieruje przeglądarkę pod adres URL, z którego skorzystam do wyświetlenia edytora produktów.

-
- **Wskazówka** Zwróć uwagę, że aby obsługiwać stronicowanie produktów, korzystam z komponentu `PageControls` z wcześniejszych rozdziałów. Funkcje administracyjne operują na tych samych funkcjach magazynu danych co w przypadku zwykłego użytkownika. Dzięki temu wspólne mechanizmy, takie jak paginacja, mogą być bez problemu użyte ponownie.
-

Dodawanie treści zastępczej edytora i tras URL

Podążając znanim tokiem pracy, zaczniemy od utworzenia treści zastępczej dla edytora, do której powrócimy, gdy zostanie ona zintegrowana z resztą aplikacji. Dodałem plik `src/components/admin/ProductEditor.vue` o zawartości pokazanej w listingu 8.3.

Listing 8.3. Zawartość pliku `src/components/admin/ProductEditor.vue`

```
<template>
  <div class="bg-info text-white text-center h4 p-2">
    Edytor produktów
  </div>
</template>
```

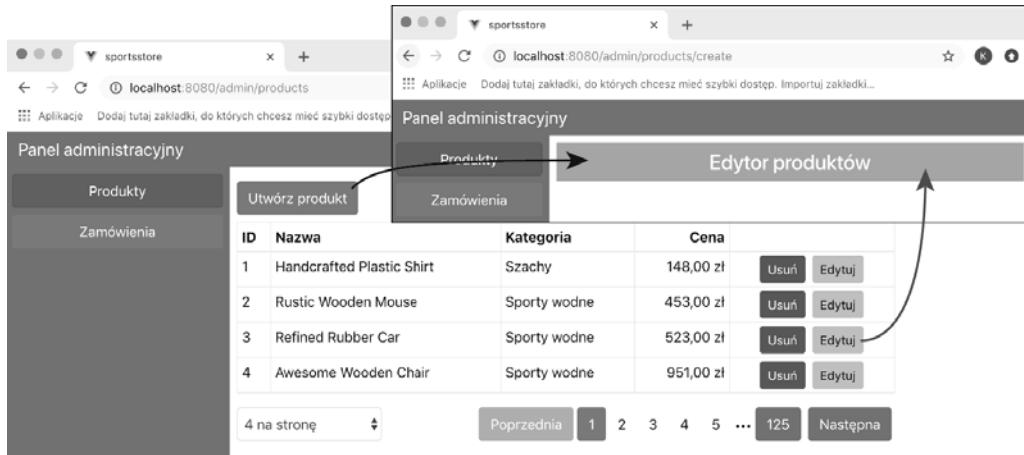
Aby zintegrować komponent z resztą aplikacji, dodaję trasę pokazaną w listingu 8.4, która ustawi adresy URL zastosowane w komponencie `ProductAdmin` w przypadku przycisków *Edytuj* lub *Utwórz produkt*.

Listing 8.4. Dodawanie trasy w pliku `src/router/index.js`

```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
import Authentication from "../components/admin/Authentication";
import Admin from "../components/admin/Admin";
import ProductAdmin from "../components/admin/ProductAdmin";
import OrderAdmin from "../components/admin/OrderAdmin";
import ProductEditor from "../components/admin/ProductEditor";
import dataStore from "../store";
Vue.use(VueRouter);
export default new VueRouter({
  mode: "history",
  routes: [
    { path: "/", component: Store },
    { path: "/cart", component: ShoppingCart },
    { path: "/checkout", component: Checkout },
    { path: "/thanks/:id", component: OrderThanks },
    { path: "/login", component: Authentication },
    { path: "/admin", component: Admin,
      beforeEnter(to, from, next) {
        if (dataStore.state.auth.authenticated) {
          next();
        } else {
          next("/login");
        }
      }
    },
    children: [
      { path: "products/:op(create|edit)/:id(\d+)?",
        component: ProductEditor },
      { path: "products", component: ProductAdmin },
      { path: "orders", component: OrderAdmin },
      { path: "", redirect: "/admin/products" }
    ]
  ],
  { path: "*", redirect: "/" }
})
```

Jak objaśniam w rozdziale 22., pakiet Vue Router może obsługiwać złożone wzorce adresów URL, w tym te zawierające wyrażenia regularne. Trasa dodana w listingu 8.4 dopasuje adres URL `/admin/products/create` (przeznaczony do tworzenia nowych produktów) i adres `/admin/products/edit/id` (gdzie ostatni fragment określa wartość liczbową reprezentującą ID produktu, który użytkownik chce edytować).

Aby zapoznać się z funkcjami administracyjnymi, przejdź pod adres `http://localhost:8080/login` i zaloguj się. Następnie zobaczysz listę produktów jak na rysunku 8.2. Jeśli klikniesz jeden z przycisków *Usuń*, wybrany produkt zostanie usunięty. Jeśli klikniesz przycisk *Utwórz produkt* lub jeden z przycisków *Edytuj*, zobaczysz treść zastępczą edytora.



Rysunek 8.2. Panel zarządzania produktami

■ **Wskazówka** Pamiętaj, że zawsze możesz odtworzyć komplet danych testowych, restartując proces json-server za pomocą polecenia przedstawionego na początku rozdziału.

Implementacja edytora produktów

Komponent edytora pozwoli na tworzenie i edycję produktów, a także określi na podstawie adresu URL, co dokładnie chce zrobić użytkownik. W listingu 8.5 usuwam treść zastępczą, a także dodaję treść i kod niezbędne do tworzenia i modyfikacji produktów.

Listing 8.5. Dodawanie funkcji do pliku src/components/admin/ProductEditor.vue

```
<template>
<div>
    <h4 class="text-center text-white p-2" v-bind:class="themeClass">
        {{ editMode ? "Edytuj" : "Utwórz produkt" }}
    </h4>
    <h4 v-if="$v.$invalid && $v.$dirty"
        class="bg-danger text-white text-center p-2">
        Należy podać wartości dla wszystkich pól
    </h4>
    <div class="form-group" v-if="editMode">
        <label>ID (Tylko do odczytu)</label>
        <input class="form-control" disabled v-model="product.id" />
    </div>
    <div class="form-group">
```

```

        <label>Nazwa</label>
        <input class="form-control" v-model="product.name" />
    </div>
    <div class="form-group">
        <label>Opis</label>
        <input class="form-control" v-model="product.description" />
    </div>
    <div class="form-group">
        <label>Kategoria</label>
        <select v-model="product.category" class="form-control">
            <option v-for="c in categories" v-bind:key="c">
                {{ c }}
            </option>
        </select>
    </div>
    <div class="form-group">
        <label>Cena</label>
        <input class="form-control" v-model="product.price" />
    </div>
    <div class="text-center">
        <router-link to="/admin/products"
            class="btn btn-secondary m-1">Anuluj
        </router-link>
        <button class="btn m-1" v-bind:class="themeClassButton"
            v-on:click="handleSave">
            {{ editMode ? "Zapisz zmiany" : "Zapisz produkt" }}</button>
    </div>
</div>
</template>
<script>
import { mapState, mapActions } from "vuex";
import { required } from "vuelidate/lib/validators";
export default {
    data: function() {
        return {
            product: {}
        }
    },
    computed: {
        ...mapState({
            pages: state => state.pages,
            currentPage: state => state.currentPage,
            categories: state => state.categoriesData
        }),
        editMode() {
            return this.$route.params["op"] == "edit";
        },
        themeClass() {
            return this.editMode ? "bg-info" : "bg-primary";
        },
        themeClassButton() {
            return this.editMode ? "btn-info" : "btn-primary";
        }
    },
    validations: {
        product: {
            name: { required },

```

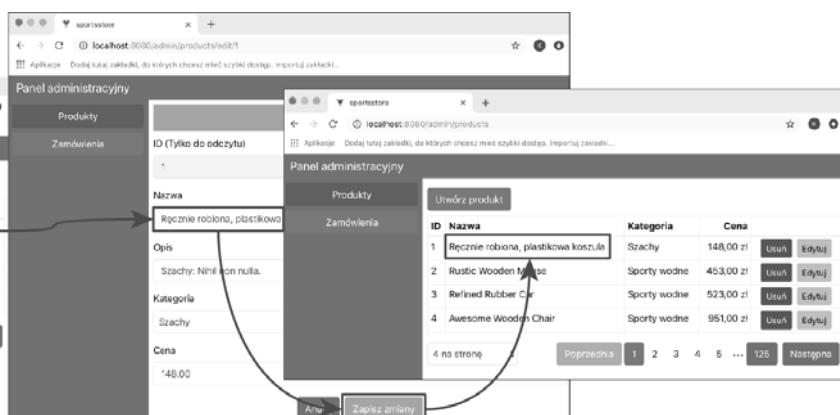
```

        description: { required },
        category: { required },
        price: { required }
    }
},
methods: {
    ...mapActions(["addProduct", "updateProduct"]),
    async handleSave() {
        this.$v.$touch();
        if (!this.$v.$invalid) {
            if (this.editMode) {
                await this.updateProduct(this.product);
            } else {
                await this.addProduct(this.product);
            }
            this.$router.push("/admin/products");
        }
    }
},
created() {
    if (this.editMode) {
        Object.assign(this.product,
            this.$store.getters.productById(this.$route.params["id"]))
    }
}
}
</script>

```

Komponent przedstawia formularz HTML zawierający pola niezbędne do utworzenia lub zmiany produktu. Cel powstania komponentu (tworzenie lub edycja produktu) jest określony za pomocą aktywnej trasy. Jeśli mamy do czynienia z edycją, magazyn danych zostaje odpytany o szczegóły produktu (dzieje się to w metodzie created opisanej w rozdziale 17.). Do skopiowania właściwości z obiektu z magazynu danych wykorzystuję metodę `Object.assign`. Dzięki temu zmiany mogą być wprowadzone bez konieczności zmian w magazynie danych. Użytkownik może kliknąć przycisk *Anuluj* i tym samym anulować zmiany. Formularz zawiera podstawowy mechanizm walidacji, a także wywołuje akcje w magazynie danych w celu wprowadzenia nowego produktu bądź zmiany już istniejącego w magazynie danych.

Aby zmienić produkt, przejdź na stronę <http://localhost:8080/admin>, zaloguj się i kliknij jeden z przycisków *Edytuj*. Skorzystaj z formularza, aby wprowadzić zmiany, i kliknij przycisk *Zapisz zmiany*, aby sprawdzić wynik operacji (rysunek 8.3).



Rysunek 8.3. Edycja produktu

Aby utworzyć produkt, kliknij przycisk *Utwórz produkt*, wypełnij formularz i kliknij przycisk *Zapisz produkt*. Nowy produkt zostanie wyświetlony na górze strony (rysunek 8.4).

ID	Nazwa	Kategoria	Cena
501	Buty do biegania	Bieganie	1,29 zł
1	Ręcznie robiona, plastikowa szachula	Szachy	148,00 zł
2	Rustic Wooden Mouse	Sporty wodne	453,00 zł
3	Refined Rubber Car	Sporty wodne	523,00 zł
4	Awesome Wooden Chair	Sporty wodne	951,00 zł

Rysunek 8.4. Tworzenie produktu

Wdrażanie sklepu sportowego

W kolejnych podrozdziałach przechodzę przez proces wdrażania aplikacji *Sklep sportowy*. Zaczniemy od wprowadzenia zmian w konfiguracji, które są niezbędne niezależnie od platformy, na której aplikacja zostanie wdrożona. Następnie skorzystam z Dockera do utworzenia kontenera, który będzie zawierał naszą aplikację i wszystkie usługi niezbędne do zastąpienia narzędzi deweloperskich stosowanych do tej pory.

Alternatywa dla Dockera

W tym rozdziale korzystam z Dockera, ponieważ jest on prosty w użyciu i spójny. Możesz z niego skorzystać na dowolnym, w miarę nowoczesnym komputerze bez potrzeby posiadania odrębnej maszyny produkcyjnej. Istnieje wiele alternatyw dla Dockera, a Vue.js może być wdrażane na liczne sposoby. Nie musisz używać Dockera, ale bez względu na wybór nie zapomnij wprowadzić zmian przedstawionych w dalszej części rozdziału.

Przygotowanie aplikacji do wdrożenia

Aby przygotować aplikację do wdrożenia, musimy wprowadzić kilka małych zmian. Nie są one związane z działaniem aplikacji — chodzi głównie o wyłączenie funkcji przydatnych dla programistów, które mogą mieć istotny wpływ na produkcyjne działanie aplikacji.

Przygotowanie magazynu danych

Na początku musimy wyłączyć tryb ścisły (ang. *strict mode*) w magazynie danych Vuex (listing 8.6). Jest to niezwykle użyteczna funkcja podczas tworzenia oprogramowania, ponieważ informuje nas o bezpośrednich modyfikacjach właściwości stanu (dokonywanych bez użycia mutacji). Nie jest ona przydatna w działaniu produkcyjnym, a dodatkowo obniża wydajność, zwłaszcza w złożonych aplikacjach.

Listing 8.6. Przygotowanie do wdrożenia w pliku *src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import CartModule from "./cart";
import OrdersModule from "./orders";
import AuthModule from "./auth";
Vue.use(Vuex);
const baseUrl = "/api";
const productsUrl = `${baseUrl}/products`;
const categoriesUrl = `${baseUrl}/categories`;
export default new Vuex.Store({
  strict: false,
  modules: { cart: CartModule, orders: OrdersModule, auth: AuthModule },
  // ...funkcje magazynu danych zostały pominięte...
})
```

Podczas fazy tworzenia korzystałem z odrębnego procesu do obsługi żądań usługi sieciowej. W gotowej aplikacji zamierzam połączyć obsługę żądań HTTP z resztą aplikacji w jeden serwer, co wymaga wprowadzania zmian w adresach URL umieszczonych w magazynie danych (przeznaczonych do wykonywania żądań) — stąd zmiana stałej `baseUrl`. W listingu 8.7 zmieniam adres URL wykorzystywany w module uwierzytelniania.

Listing 8.7. Zmiana adresu URL w pliku *src/store/auth.js*

```
import Axios from "axios";
const loginUrl = "/api/login";
export default {
  state: {
    authenticated: false,
    jwt: null
  },
  // ...funkcje magazynu danych zostały pominięte...
}
```

Adres URL do zarządzania zamówieniami również musi ulec zmianie (listing 8.8).

Listing 8.8. Zmiana adresu URL w pliku *src/store/orders.js*

```
import Axios from "axios";
import Vue from "vue";
const ORDERS_URL = "/api/orders";
export default {
  state: {
    orders: []
  },
  // ...funkcje magazynu danych zostały pominięte...
}
```

Przygotowanie komponentu uwierzytelniania

Kolejna zmiana polega na usunięciu danych uwierzytelniających do panelu administracyjnego z komponentu (listing 8.9) — było to niezwykle przydatne w czasie tworzenia aplikacji, ale bezwzględnie nie powinno się znaleźć w wersji produkcyjnej.

Listing 8.9. Usuwanie danych uwierzytelniających z pliku *src/components/admin/Authentication.vue*

```
...
<script>
import { required } from "vuelidate/lib/validators";
import { mapActions, mapState } from "vuex";
import ValidationError from "../ValidationError";
export default {
  components: { ValidationError },
  data: function() {
    return {
      username: null,
      password: null,
      showFailureMessage: false,
    }
  },
  computed: {
    ...mapState({authenticated: state => state.auth.authenticated })
  },
  validations: {
    username: { required },
    password: { required }
  },
  methods: {
    ...mapActions(["authenticate"]),
    async handleAuth() {
      this.$v.$touch();
      if (!this.$v.$invalid) {
        await this.authenticate({ name: this.username,
          password: this.password });
        if (this.authenticated) {
          this.$router.push("/admin");
        } else {
          this.showFailureMessage = true;
        }
      }
    }
  }
}
</script>
...
```

Ładowanie funkcji administracyjnych na żądanie

Każdy z mechanizmów dostępnych w naszej aplikacji jest dołączony do paczki języka JavaScript tworzonej przez narzędzia budowania Vue.js, mimo że funkcje administracyjne będą stosowane tylko przez niewielką część użytkowników. Vue.js pozwala na łatwy podział aplikacji na odrębne paczki, które mogą być ładowane tylko wtedy, kiedy są rzeczywiście potrzebne. Aby wyodrębnić funkcje administracyjne do własnej paczki, dodaję plik *index.js* do katalogu *src/components/admin* (listing 8.10).

Listing 8.10. Zawartość pliku *src/components/admin/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import Store from "../components/Store";
import ShoppingCart from "../components/ShoppingCart";
import Checkout from "../components/Checkout";
import OrderThanks from "../components/OrderThanks";
const Authentication = () =>
  import(/* webpackChunkName: "admin" */ "../components/admin/Authentication");
```

```

const Admin = () =>
  import(/* webpackChunkName: "admin" */ "../components/admin/Admin");
const ProductAdmin = () =>
  import(/* webpackChunkName: "admin" */ "../components/admin/ProductAdmin");
const OrderAdmin = () =>
  import(/* webpackChunkName: "admin" */ "../components/admin/OrderAdmin");
const ProductEditor = () =>
  import(/* webpackChunkName: "admin" */ "../components/admin/ProductEditor");
import dataStore from "../store";
Vue.use(VueRouter);
export default new VueRouter({
  mode: "history",
  routes: [
    { path: "/", component: Store },
    { path: "/cart", component: ShoppingCart },
    { path: "/checkout", component: Checkout},
    { path: "/thanks/:id", component: OrderThanks},
    { path: "/login", component: Authentication },
    { path: "/admin", component: Admin,
      beforeEnter(to, from, next) {
        if (dataStore.state.auth.authenticated) {
          next();
        } else {
          next("/login");
        }
      },
      children: [
        { path: "products/:op(create|edit)/:id(\\d+)?",
          component: ProductEditor },
        { path: "products", component: ProductAdmin },
        { path: "orders", component: OrderAdmin },
        { path: "", redirect: "/admin/products" }
      ]
    },
    { path: "**", redirect: "/" }
  ]
})

```

Zwykłe instrukcje `import` tworzą zależności statyczne od modułów. Oznacza to, że zaimportowanie komponentu zapewni jego obecność w pliku JavaScript przesłanym do przeglądarki. Jak objaśniam w rozdziale 21., rodzaj instrukcji `import`, z której korzystam w listingu 8.10, jest dynamiczny. Oznacza to, że komponenty niezbędne do administrowania zostaną umieszczone w odrębnym pliku JavaScript, który zostanie załadowany przy pierwszej próbie skorzystania z funkcji administracyjnych. W ten sposób upewniamy się, że funkcje administracyjne będą dostępne tylko dla nielicznych uprawnionych użytkowników.

- **Uwaga** Możesz zauważyc, że odrębny moduł, utworzony w listingu 8.10, zostanie załadowany nawet wtedy, gdy nie korzystasz z funkcji administracyjnych. Wynika to z faktu, że projekty Vue.js są skonfigurowane tak, aby dostarczyć przeglądarce specjalnych wskazówek dotyczących treści, która może być potrzebna w przyszłości. Przeglądarka może zignorować te wskazówki, ale może też pobrać wskazany moduł. W rozdziale 21. pokaże, jak skonfigurować projekt w taki sposób, aby wskazówki nie były przesyłane do przeglądarki.

Nietypowe komentarze, które umieściłem w instrukcjach `import`, zapewniają, że komponenty zostaną spakowane w jeden, odrębny plik. Bez tych komentarzy każdy komponent zostałby umieszczony we własnym pliku, który byłby pobierany przy pierwszej okazji. Skoro komponenty dostarczają powiązane

funkcje, postanowiłem zgrupować je razem. Jest to mechanizm dostępny w narzędziu używanym przez Vue.js do tworzenia paczek o nazwie *webpack* z kodem JavaScript. Nie będzie ono działać, jeśli nie korzystałeś w swoim projekcie z narzędzi wiersza poleceń omówionych w rozdziale 5.

Tworzenie pliku danych

Do tej pory pracowałem z testowymi danymi wygenerowanymi z poziomu kodu w momencie startu REST-owej usługi sieciowej. Było to podejście użyteczne w czasie tworzenia aplikacji, ponieważ oryginalne dane były odtwarzane przy każdym starcie aplikacji. W przypadku wdrożenia musimy utrważyć dane, gwarantując, że wszystkie zmiany są wprowadzane faktycznie na trwałe. W związku z tym dodaję do katalogu *sportsstore* plik *data.json* o treści jak w listingu 8.11.

Listing 8.11. Zawartość pliku *sportsstore/data.json*

```
{
  "products": [
    { "id": 1, "name": "Kajak", "category": "Sporty wodne",
      "description": "Kajak dla jednej osoby", "price": 275 },
    { "id": 2, "name": "Kamizelka ratunkowa", "category": "Sporty wodne",
      "description": "Skuteczna i modna", "price": 48.95 },
    { "id": 3, "name": "Piłka nożna", "category": "Piłka nożna",
      "description": "Zaakceptowana przez federację pod względem wagi i wymiarów",
      "price": 19.50 },
    { "id": 4, "name": "Choragiewki do rogów boiska", "category": "Piłka nożna",
      "description": "Odmień swoje boisko, nadając mu profesjonalny sznyt",
      "price": 34.95 },
    { "id": 5, "name": "Stadion", "category": "Piłka nożna",
      "description": "Stadion na 35 000 miejsc", "price": 79500 },
    { "id": 6, "name": "Myśląca czapeczka", "category": "Szachy",
      "description": "Zwiększa wydajność Twojego mózgu o 75%", "price": 16 },
    { "id": 7, "name": "Chwiejne krzesło", "category": "Szachy",
      "description": "Potajemnie doprowadź przeciwnika do irytacji",
      "price": 29.95 },
    { "id": 8, "name": "Szachownica", "category": "Chess",
      "description": "Zabawa dla całej rodziny", "price": 75 },
    { "id": 9, "name": "Król(u) złoty", "category": "Chess",
      "description": "Pozłacana, zdobiona diamentami figura króla", "price": 1200 }
  ],
  "categories": ["Sporty wodne", "Piłka nożna", "Szachy"],
  "orders": []
}
```

Budowanie aplikacji do wdrożenia

Aby przygotować aplikację *Sklep sportowy* w formie gotowej do wdrożenia, wykonaj polecenie z listingu 8.12 w katalogu *sportsstore*.

Listing 8.12. Budowanie projektu

```
npm run build
```

W procesie budowania generowane są pliki JavaScript zoptymalizowane pod kątem przesyłania ich do przeglądarki, a także wyłączane są pewne funkcje deweloperskie, takie jak automatyczne odświeżanie przeglądarki w momencie wykrycia zmiany w jakimkolwiek pliku. Proces budowania może zająć chwilę, a po jego zakończeniu powinieneś zobaczyć komunikat podobny do poniższego:

```

WARNING Compiled with 2 warnings
warning
asset size limit: The following asset(s) exceed the recommended size limit (244 KiB).
This can impact web performance.
Assets:
  img/fontawesome-webfont.912ec66d.svg (434 KiB)
warning
entrypoint size limit: The following entrypoint(s) combined asset size exceeds the
recommended limit (244 KiB). This can impact web performance.
Entrypoints:
  app (346 KiB)
    css/chunk-vendors.291cf91.css
    js/chunk-vendors.56adf36a.js
    js/app.846b07bf.js
File
dist\js\chunk-vendors.56adf36a.js
dist\js\app.846b07bf.js
dist\js\admin.b43c91ef.js
dist\css\chunk-vendors.291cf91.css
Size
160.98 kb
25.08 kb
11.62 kb
159.97 kb
Gzipped
54.10 kb
6.57 kb
3.13 kb
26.60 kb
Images and other types of assets omitted.
DONE Build complete. The dist directory is ready to be deployed.

```

Ostrzeżenia o rozmiarze plików mogą zostać zignorowane. Efektem procesu budowania jest zbiór plików JavaScript i CSS, które zawierają treść, kod i style wymagane przez aplikację. Wszystkie pliki znajdują się w katalogu *dist*. Nie przejmuj się treścią ostrzeżeń — może ona różnić się od powyższego wydruku z uwagi na częste aktualizacje narzędzi deweloperskich.

Testowanie aplikacji gotowej do wdrożenia

Przed utworzeniem paczki wdrożeniowej warto wykonać szybki test, aby przekonać się, że wszystko działa, jak należy. Podczas procesu tworzenia aplikacji żądania HTTP dotyczące plików HTML, JavaScript i CSS były obsługiwane przez narzędzia deweloperskie Vue.js, które nie mogą być używane w środowisku produkcyjnym. Alternatywnie można skorzystać z popularnego pakietu *Express*, który jest szeroko stosowanym serwerem webowym używającym technologii Node.js. Wykonaj polecenia z listingu 8.13, aby zainstalować pakiet *Express* i dodatkowy pakiet niezbędny do obsługi trasowania adresów URL.

Listing 8.13. Dodawanie pakietów

```

npm install --save-dev express@4.16.3
npm install --save-dev connect-history-api-fallback@1.5.0

```

Do katalogu *sportsstore* dodałem plik *server.js* o treści przedstawionej w listingu 8.14. Jego zadaniem jest konfiguracja pakietów z listingu 8.13, dzięki czemu będą one używane w naszej aplikacji.

Listing 8.14. Zawartość pliku sportsstore/server.js

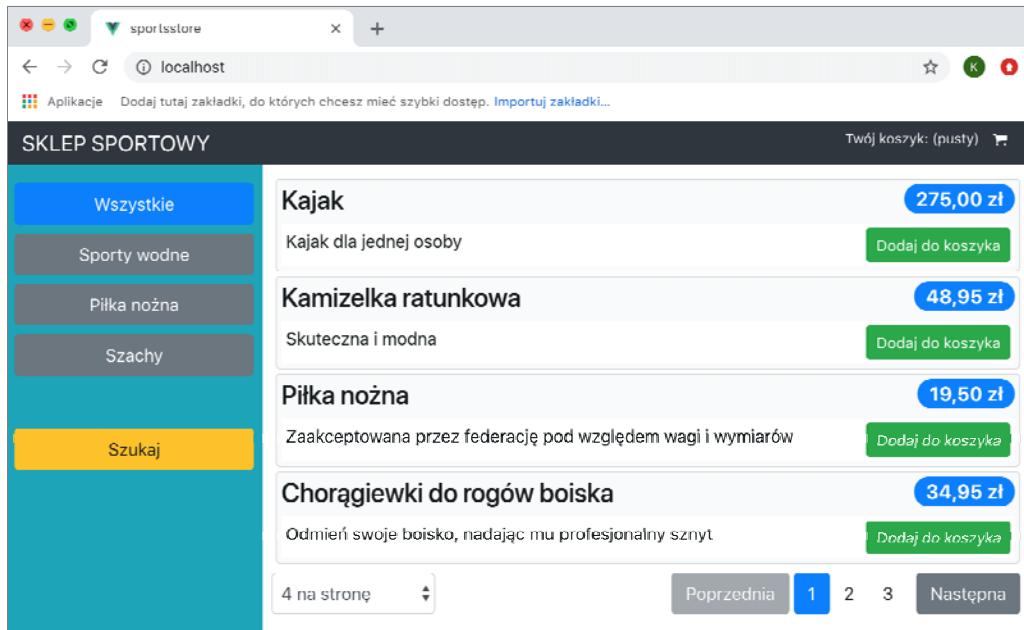
```
const express = require("express");
const history = require("connect-history-api-fallback");
const jsonServer = require("json-server");
const bodyParser = require('body-parser');
const auth = require("./authMiddleware");
const router = jsonServer.router("data.json");
const app = express();
app.use(bodyParser.json());
app.use(auth);
app.use("/api", router);
app.use(history());
app.use("/", express.static("./dist"));
app.listen(80, function () {
    console.log("Serwer HTTP działa na porcie 80");
});
```

Wykonaj polecenie z listingu 8.15 w katalogu *sportsstore*, aby uruchomić aplikację.

Listing 8.15. Testowanie wersji produkcyjnej naszej aplikacji

```
node server.js
```

To polecenie wykona instrukcje JavaScript z listingu 8.14, uruchamiając w ten sposób serwer webowy na porcie 80 i nasłuchując żądań. Aby przetestować aplikację, przejdź pod adres *http://localhost* i przekonaj się, że działa ona poprawnie (rysunek 8.5).



Rysunek 8.5. Testowanie aplikacji

Wdrożenie aplikacji

Aby wdrożyć aplikację, musisz zacząć od pobrania i zainstalowania narzędzi Dockera na Twojej maszynie deweloperskiej. W tym celu przejdź na stronę www.docker.com/products/docker. Znajdziesz tam wersje dostępne dla systemów macOS, Windows i Linux, a także specjalistyczne wydania dla platform chmurowych Amazona i Microsoftu. Darmowa wersja Community powinna wystarczyć do naszych celów.

-
- **Ostrzeżenie** Firma, która rozwija Dockera, jest znana z niezachowywania kompatybilności wstępnej. Oznacza to, że przykłady z tej książki mogą nie zadziałać z najnowszymi wersjami Dockera.
-

Tworzenie pliku paczki

Aby wdrożyć aplikację w Dockerze, musimy utworzyć plik *package.json*, który zainstaluje pakiety niezbędne do działania aplikacji. W związku z tym dodaję plik *deploy-package.json* do katalogu *sportsstore* (listing 8.16).

Listing 8.16. Zawartość pliku *sportsstore/deploy-package.json*

```
{
  "name": "sportsstore",
  "version": "1.0.0",
  "private": true,
  "dependencies": {
    "faker": "^4.1.0",
    "json-server": "^0.12.1",
    "jsonwebtoken": "^8.1.1",
    "express": "4.16.3",
    "connect-history-api-fallback": "1.5.0"
  }
}
```

Tworzenie kontenera Dockera

Aby utworzyć kontener, musimy dodać plik *Dockerfile* (bez rozszerzenia) do katalogu *sportsstore* i dodać treść z listingu 8.17.

Listing 8.17. Zawartość pliku *sportsstore/Dockerfile*

```
FROM node:8.11.2
RUN mkdir -p /usr/src/sportsstore
COPY dist /usr/src/sportsstore/dist
COPY authMiddleware.js /usr/src/sportsstore/
COPY data.json /usr/src/sportsstore/
COPY server.js /usr/src/sportsstore/server.js
COPY deploy-package.json /usr/src/sportsstore/package.json
WORKDIR /usr/src/sportsstore
RUN npm install
EXPOSE 80
CMD ["node", "server.js"]
```

Zawartość pliku *Dockerfile* korzysta z bazowego obrazu skonfigurowanego dla technologii Node.js i powoduje skopiowanie plików wymaganych do uruchomienia aplikacji, włączając w to plik paczki zawierający aplikację, a także plik *package.json* używany do zainstalowania pakietów niezbędnych do jej uruchomienia.

Uruchom polecenie z listingu 8.18 w katalogu *sportsstore*, aby utworzyć obraz, który będzie zawierał aplikację *Sklep sportowy* wraz z wszystkimi wymaganymi narzędziami i pakietami.

Listing 8.18. Budowanie obrazu Dockera

```
docker build . -t sportsstore -f Dockerfile
```

Obraz to nic innego jak szablon dla kontenerów. W miarę przetwarzania pliku *Dockerfile* przez Dockera pakiety NPM zostaną pobrane i zainstalowane, a pliki konfiguracyjne i kod aplikacji — skopiowane do obrazu.

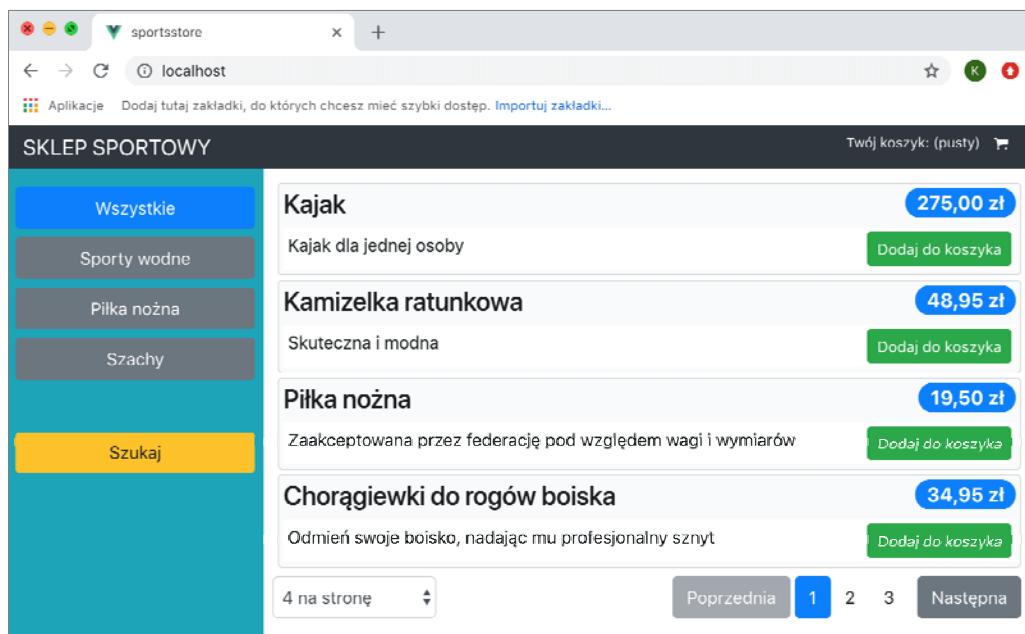
Uruchamianie aplikacji

Po utworzeniu obrazu utwórz i uruchom nowy kontener za pomocą polecenia z listingu 8.19.

Listing 8.19. Tworzenie kontenera Dockera

```
docker run -p 80:80 sportsstore
```

Przejdź pod adres *http://localhost* w przeglądarce i przekonaj się, że aplikacja uruchomiona w kontenerze działa (rysunek 8.6).



Rysunek 8.6. Uruchamianie skonteneryzowanej aplikacji Sklep sportowy

- **Wskazówka** Jeśli otrzymasz błąd z uwagi na brak dostępności portu 80, możesz spróbować skorzystać z innego portu, zmieniając pierwszą wartość argumentu *-p*. Na przykład jeśli w przeglądarce chcesz wejść na port 500, zmień wartość argumentu na *-p 500:80*.

Aby zatrzymać kontener, wykonaj polecenie z listingu 8.20.

Listing 8.20. Zatrzymywanie kontenera w Dockerze

```
docker ps
```

Zobaczysz listę uruchomionych kontenerów (pominąłem niektóre pola dla czytelności):

CONTAINER ID	IMAGE	COMMAND	CREATED
ecc84f7245d6	sportsstore	"node server.js"	33 seconds ago

Korzystając z wartości z kolumny *Container ID*, wykonaj polecenie z listingu 8.21.

Listing 8.21. Zatrzymywanie kontenera w Dockerze

```
docker stop ecc84f7245d6
```

Podsumowanie

W tym rozdziale zakończyłem prace nad aplikacją *Sklep sportowy*. Dodałem obsługę zarządzania katalogiem produktów, a także przygotowałem projekt do wdrożenia, pokazując, jak łatwo można przystosować projekt Vue.js do uruchomienia w środowisku produkcyjnym.

Na tym kończymy pierwszą część książki. W części drugiej zacznijemy zagłębiać się w szczegóły, poznamy bliżej funkcje, z których skorzystaliśmy podczas tworzenia naszej aplikacji.

CZĘŚĆ II



Vue.js pod lupa



ROZDZIAŁ 9.

Jak działa Vue.js?

Na początku swojej przygody z Vue.js możesz poczuć się przytłoczony dużą liczbą zagadnień do zrozumienia. Nie wszystko wydaje się mieć sens od samego początku. W tym rozdziale objaśniam, jak działa Vue.js, i pokazuję, że nie ma w nim żadnej magii — warto o tym pamiętać, gdy będę zagłębiał się w tajniki poszczególnych zagadnień Vue.js w kolejnych rozdziałach. Tabela 9.1 podsumowuje bieżący rozdział.

Tabela 9.1. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Utwórz prostą aplikację webową bez pomocy Vue.js.	Skorzystaj z API DOM (ang. <i>Document Object Model</i> — obiektowy model dokumentu).	9.6
Utwórz prostą aplikację webową z pomocą Vue.js.	Utwórz obiekt <code>Vue</code> i skonfiguruj go przy użyciu właściwości <code>el</code> i <code>template</code> .	9.7
Zaprozentuj wartość danych użytkownika.	Zdefiniuj właściwość danych i użyj wiązania danych.	9.8
Zareaguj na działania użytkownika za pomocą zdarzeń.	Skorzystaj z dyrektywy <code>v-on</code> .	9.9
Wygeneruj wartości danych, które wymagają obliczeń.	Zdefiniuj właściwość obliczaną.	9.10
Utwórz wielokrotnego użytku funkcjonalną jednostkę aplikacji.	Utwórz i zastosuj komponent.	9.11 – 9.13
Zdefiniuj szablon komponentu jako kod HTML.	Skorzystaj z elementu <code>template</code> .	9.14 – 9.16

Przygotowania do rozdziału

Aby utworzyć przykładowy projekt, otwórz wiersz poleceń, przejdź do wybranego katalogu i wykonaj polecenie z listingu 9.1. To polecenie wymaga posiadania narzędzi z rozdziału 1. Musisz wykonać opisane w nim operacje, aby utworzyć projekt.

Listing 9.1. Tworzenie przykładowego projektu

```
vue create nomagic
```

-
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.
-

Gdy zostaniesz poproszony o wybór ustawień, wybierz opcję `default`. Projekt zostanie utworzony, a pakiety do wszystkich narzędzi deweloperskich zostaną pobrane i zainstalowane, co może trochę potrwać.

-
- **Uwaga** W trakcie pisania niniejszej książki pakiet `@vue/cli` był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratum dostępną w repozytorium oryginalnego wydania książki pod adresem <https://github.com/Apress/pro-vue-js-2>.
-

Po utworzeniu projektu do katalogu `nomagic` dodaj plik `vue.config.js` o zawartości jak w listingu 9.2. Plik ten jest używany do konfigurowania narzędzi deweloperskich Vue.js, które opisuję w rozdziale 10.

Listing 9.2. Zawartość pliku nomagic/vue.config.js

```
module.exports = {
  runtimeCompiler: true
}
```

Dodawanie frameworka Bootstrap CSS

Aby dodać pakiet Bootstrap CSS do projektu, wykonaj polecenie z listingu 9.3 w katalogu `nomagic`. Jest to framework CSS używany do stylowania treści HTML w tym rozdziale.

Listing 9.3. Dodawanie pakietu Bootstrap

```
npm install bootstrap@4.0.0
```

Po zakończeniu instalacji otwórz plik `src/main.js` i dodaj instrukcje z listingu 9.4.

Listing 9.4. Dodawanie Bootstrapa w pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
  render: h => h(App)
}).$mount('#app')
```

Ta instrukcja dołączy style CSS Bootstrapa do treści przesyłanych do przeglądarki.

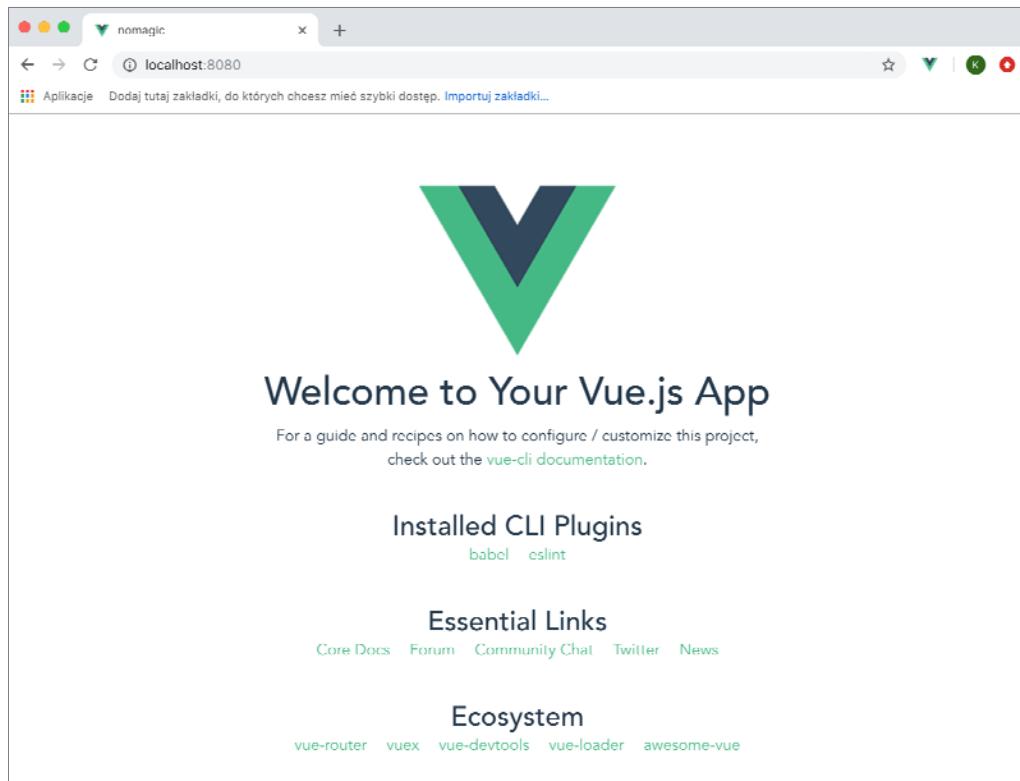
Uruchamianie przykładowej aplikacji

Projekt utworzony za pomocą polecenia z listingu 9.1 dołącza narzędzia wymagane w procesie tworzenia aplikacji w Vue.js. Więcej na temat zasad użycia tych narzędzi znajdziesz w rozdziale 10. Na razie, aby uruchomić narzędzia deweloperskie, uruchom polecenie z listingu 9.5 w katalogu `nomagic`.

Listing 9.5. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

W tym momencie nastąpi start serwera deweloperskiego HTTP. Otwórz nowe okno przeglądarki i przejdź pod adres <http://localhost:8080>. Zobaczysz treść zastępczą jak na rysunku 9.1.



Rysunek 9.1. Uruchamianie przykładowej aplikacji

Tworzenie aplikacji za pomocą API modelu DOM

Zaczniemy od utworzenia prostej aplikacji webowej bez użycia choćby odrobiny kodu Vue.js. Następnie pokażę, jak osiągnąć ten sam efekt za pomocą podstawowych opcji Vue.js.

W tym celu utworzę plik *main.js*, który zazwyczaj zawiera kod JavaScript odpowiedzialny za inicjalizację aplikacji Vue.js. Plik *main.js* to zwykły plik języka JavaScript, co oznacza, że możemy usunąć kod odpowiedzialny za konfigurację Vue.js i wstawić tam dowolne inne instrukcje.

W listingu 9.6 zamieniam domyślną treść pliku *main.js* na zwykłe instrukcje języka JavaScript, które korzystają z obiektowego modelu dokumentu (DOM). API modelu DOM oferują wszystkie przeglądarki w celu udostępniania treści dokumentu HTML w obrębie skryptów języka JavaScript. Model DOM stanowi podstawę działania praktycznie wszystkich aplikacji webowych.

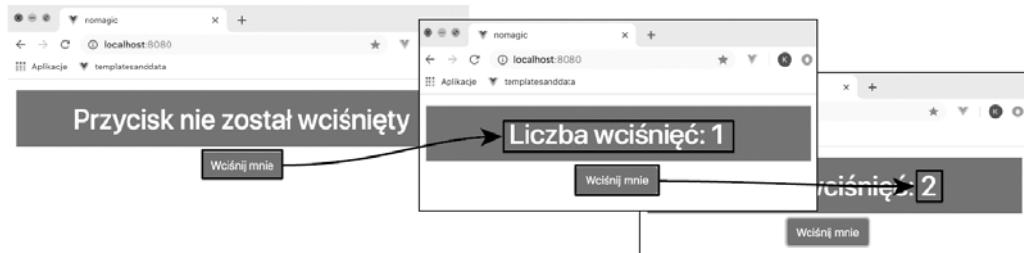
Listing 9.6. Zamiana treści w pliku src/main.js

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
let counter = 1;
let container = document.createElement("div");
container.classList.add("text-center", "p-3");
let msg = document.createElement("h1");
msg.classList.add("bg-primary", "text-white", "p-3");
```

```
msg.textContent = "Przycisk nie został wciśnięty";
let button = document.createElement("button");
button.textContent = "Wciśnij mnie";
button.classList.add("btn", "btn-secondary");
button.onclick = () => msg.textContent = `Liczba wciśnięć: ${counter++}`;
container.appendChild(msg);
container.appendChild(button);
let app = document.getElementById("app");
app.parentElement.replaceChild(container, app);
```

Nie martw się instrukcjami z powyższego kodu. W większości przypadków nie będziesz korzystać z API modelu DOM w projektach Vue.js. Celem tego przykładu jest przedstawienie pliku *main.js* jako zwykłego skryptu, zawierającego zwyczajny kod JavaScript, który może korzystać ze standardowych funkcji udostępnianych przez przeglądarkę.

Narzędzia deweloperskie uruchomione za pomocą polecenia z listingu 9.6 automatycznie wykrywają zmiany w plikach projektu. Zmiany są kompilowane, pakowane w jeden plik zawierający wszystkie funkcje aplikacji i wysypane z powrotem do przeglądarki. Oznacza to, że po zapisaniu pliku *main.js* przeglądarka zostanie odświeżona, a Ty zobaczysz treść jak na rysunku 9.2. Na początku widać prosty komunikat, ale po kliknięciu przycisku zostaje on zastąpiony licznikiem. Licznik będzie zwiększać się po każdym wciśnięciu przycisku.



Rysunek 9.2. Działanie bezpośrednio na modelu DOM

Jak działa aplikacja w modelu DOM?

Z API modelu DOM korzystam w celu utworzenia elementu *div*, który zawiera elementy *h1* i *button*, w momencie wykonywania pliku *main.js*. Jednocześnie wiążę te elementy z klasami, które odpowiadają stylem zdefiniowanym we frameworku Bootstrap CSS, określającymi wygląd tych elementów. Następnie ustawiam słuchacza zdarzeń, który reaguje na kliknięcie przycisku poprzez modyfikowanie stanu licznika i wyświetlanie komunikatu w elemencie *h1*.

Po wykonaniu wyżej opisanych operacji lokalizuję istniejący w dokumencie element o ID równym *app* i zamieniam go na utworzony element *div*.

```
...
let app = document.getElementById("app");
app.parentElement.replaceChild(container, app);
...
```

Poniżej przedstawiam sposób, w jaki użytkownik otrzyma aplikację. Przeglądarka wysyła żądanie pod adres *http://localhost:8000*, a serwer deweloperski HTTP odpowiada zawartością pliku *index.html*:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
```

```
<title>nomagic</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

Zaznaczyłem wyraźnie element zamieniany przez kod w języku JavaScript. To właśnie tu znajdzie się kod HTML generowany w JavaScriptie.

Gdy serwer wygeneruje odpowiedź na żądanie HTTP przeglądarki, automatycznie zostanie wstawiony element `script`, który wczyta plik JavaScript zawierający cały kod projektu. Możesz to sprawdzić, klikając prawym przyciskiem myszy w oknie przeglądarki i wybierając z menu kontekstowego opcję *Wyświetl źródło strony*.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>nomagic</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="/app.js"></script>
  </body>
</html>
```

Dostarczony plik JavaScript zawiera kod z pliku `main.js` i wszelkie jego zależności. Przeglądarka wykonuje instrukcje w pliku JavaScript, co prowadzi do wygenerowania omówionej wcześniej treści.

Kod HTML powstały w wyniku połączenia statycznych fragmentów z pliku `index.html` i dynamicznych znaczników wygenerowanych przez plik `main.js` można podejrzeć poprzez kliknięcie prawym przyciskiem w oknie przeglądarki i wybranie opcji *Zbadaj w menu kontekstowym*. W ten sposób zobaczymy widok na żywo dokumentu, utworzony za pomocą API modelu DOM.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <title>nomagic</title>
  </head>
  <body>
    <div class="text-center p-3">
      <h1 class="bg-primary text-white p-3">Przycisk nie został wciśnięty</h1>
      <button class="btn btn-secondary">Wciśnij mnie</button>
    </div>
    <script type="text/javascript" src="/app.js"></script>
  </body>
</html>
```

Efekt nie jest zbyt imponujący, ale w ten sposób pokazujemy, że zwykły kod JavaScript, z pomocą API modelu DOM, również może być używany do zamiany elementów w dokumentach HTML, a także do tworzenia nowej treści i reagowania na zdarzenia pochodzące od użytkownika. To są podstawy każdej aplikacji webowej — w tym także tej tworzonej w Vue.js.

Tworzenie obiektu Vue

Aplikacje Vue.js również korzystają z API modelu DOM do tworzenia treści HTML, reagowania na zdarzenia i aktualizowania wartości danych. Różnica polega na tym, że podejście z zastosowaniem Vue.js jest bardziej eleganckie, łatwiejsze do zrozumienia i bardziej skalowalne.

Przed chwilą skorzystałem z pliku *main.js* jako wygodnej metody wykonania kodu JavaScript, jednak jego głównym celem w aplikacji Vue.js jest utworzenie obiektu Vue, który stanowi punkt startowy całej aplikacji. W listingu 9.7 zamieniam kod API modelu DOM w pliku *main.js* na zestaw instrukcji, który tworzy obiekt Vue, i przywracam tym samym plikowi *main.js* jego pierwotne przeznaczenie.

Listing 9.7. Tworzenie obiektu Vue w pliku src/main.js

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
  el: "#app",
  template: `<div class="text-center p-3">
    <h1 class="bg-secondary text-white p-3">
      Vue: Przycisk nie został wcisnięty
    </h1>
    <button class="btn btn-secondary">
      Wciśnij mnie
    </button>
  </div>`});
});
```

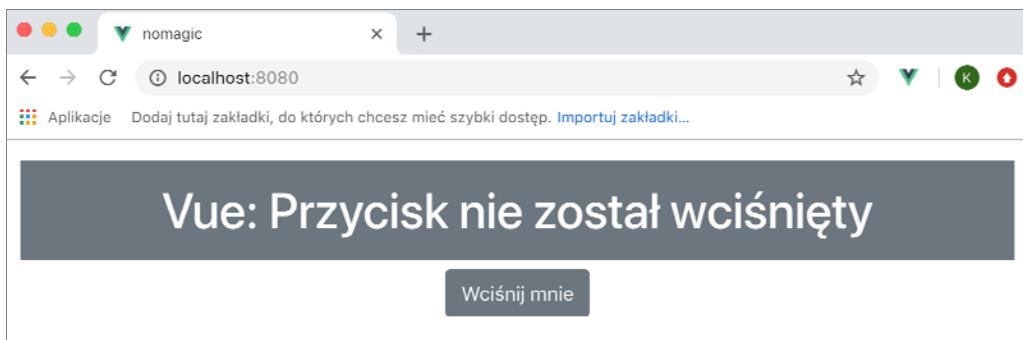
Instrukcja `import` jest niezbędna do zimportowania obiektu `Vue` z modułu `vue`, pobranego do katalogu `node_modules` w momencie utworzenia projektu. Obiekt `Vue` został utworzony za pomocą słowa kluczowego `new`, a konstruktor przyjął obiekt konfiguracji, którego właściwości pozwalają na kontrolę zachowania aplikacji i definiują treści przedstawiane użytkownikowi.

W tym przykładzie mamy do czynienia z dwoma właściwościami konfiguracji: `el` i `template`. Obiekt `Vue` korzysta z właściwości `el`, aby zidentyfikować element, który zostanie zamieniony przez zawartość aplikacji w pliku `index.html`. Właściwość `template` dostarcza zawartość HTML, która zamieni dotychczasową zawartość elementu ukrytego za właściwością `el`. Jak pewnie pamiętasz, te zadania musiałem z pomocą API modelu DOM wykonywać ręcznie w poprzednim przykładzie.

W listingu 9.7 ustawiam właściwość `el` na wartość `#app`, która pobiera element o id równym `app`. Ustawiam właściwość `template` w taki sposób, aby zawierała tę samą strukturę kodu HTML co w listingu 9.6. Tym razem mogę jednak po prostu napisać kod HTML, zamiast tworzyć identyczną konstrukcję za pomocą wywołań kodu JavaScript. Obiekt konfiguracji jest zdefiniowany w kodzie JavaScript, co oznacza, że kod HTML w nim osadzony musi być zawarty jako prawidłowy łańcuch znaków JavaScript. Skorzystałem z grawisa (znaku `'`), dzięki czemu jestem w stanie pisać kod w wielu wierszach, przez co jest on łatwiejszy do odczytu.

- **Ostrzeżenie** Możesz korzystać z właściwości `template` tylko, jeśli przesłoniłeś ustawienia projektu jak w listingu 9.2. Domyślnie funkcja do przetwarzania tekstów zawierających szablony jest wyłączona w Vue.js.

Po zapisaniu zmian w pliku *main.js* przeglądarka zostanie odświeżona, a treść będzie przypominać tę z rysunku 9.3. Zmieniła się treść komunikatu, a także kolor tła.



Rysunek 9.3. Zastosowanie obiektu Vue

Stosowanie obiektu Vue

Mój obiekt Vue wyświetla zawartość HTML użytkownikowi, ale to tylko część funkcji, które są mi potrzebne. Kolejnym krokiem jest dodanie danych i wyświetlenie ich użytkownikowi. W listingu 9.8 dodaję zmienną counter i modyfikuję łańcuch template tak, aby został wyświetlony użytkownikowi.

Listing 9.8. Dodawanie zmiennej do pliku src/main.js

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
  el: "#app",
  template: `<div class="text-center p-3">
    <h1 class="bg-secondary text-white p-3">
      Liczba wciśnięć: {{ counter }}
    </h1>
    <button class="btn btn-secondary">
      Wciśnij mnie
    </button>
  </div>`,
  data: {
    counter: 0
  }
});
```

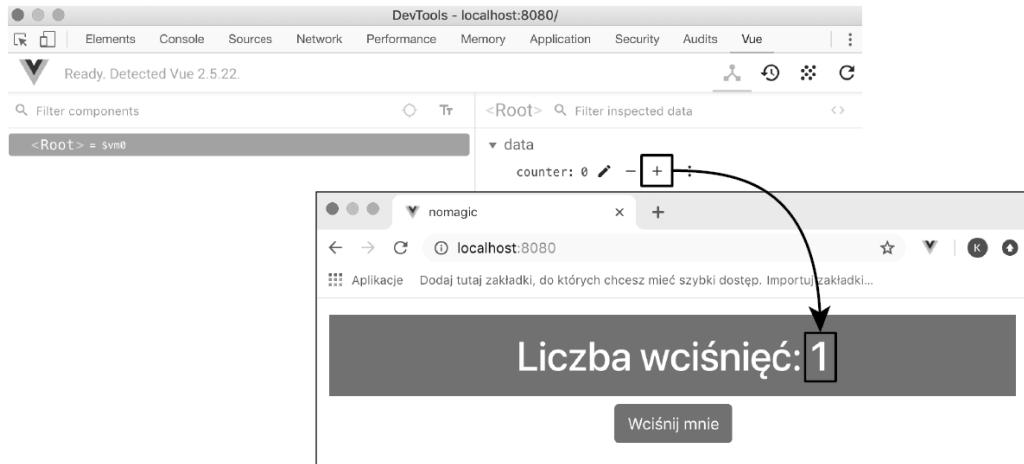
Dodalem właściwość data do obiektu konfiguracyjnego aplikacji. Obiekt definiuje właściwość counter z wartością początkową równą zero. Aby wyświetlić wartość licznika, stosuję proste wiązanie danych w elemencie h1.

```
...
<h1 class="bg-secondary text-white p-3">
  Liczba wciśnięć: {{ counter }}
</h1>
...
```

Wiązania danych to niezwykle ważna funkcja Vue.js, która łączy obiekt data z treścią właściwości template. Gdy Vue.js wyświetla treść szablonu, poszukuje w nim wiązań danych, rozpoznaje je jako wyrażenia języka JavaScript i dołącza wynik operacji jako kod HTML. W tym przypadku wiązanie danych odwołuje się do jednej z właściwości obiektu data. W rezultacie wartość zmiennej counter zostanie wstawiona na stronę w elemencie h1.

Dane są **reaktywne, odświeżane na żywo** — oznacza to, że zmiana wartości właściwości counter zostanie automatycznie odzwierciedlona w elemencie h1. Po utworzeniu obiektu Vue następuje przetworzenie obiektu data i zamiana właściwości, przez co możliwe jest wykrycie zmian w ich wartościach.

Konsekwencję reaktywności danych widać w narzędziu Vue Devtools. Otwórz narzędzia deweloperskie za pomocą klawisza F12, przejdź na zakładkę *Vue* i kliknij element <Root>. W panelu po prawej stronie przesuń kursor na element counter i kliknij przycisk +, aby zwiększyć jego wartość. Vue.js wykryje zmianę wartości właściwości i przetworzy kod szablonu ponownie, generując efekt jak na rysunku 9.4. Szczegóły instalacji tej wtyczki znajdziesz na stronie <https://github.com/vuejs/vue-devtools>.



Rysunek 9.4. Zmiana wartości zmiennej reaktywnej

Dodawanie funkcji obsługi zdarzenia

Kolejnym krokiem w tworzeniu tego przykładu jest zwiększenie wartości zmiennej counter automatycznie po kliknięciu przycisku. Poprzednio mogłem powiązać funkcję obsługi zdarzenia bezpośrednio za pomocą instrukcji języka JavaScript. Tym razem zastosuję inne podejście (listing 9.9).

Listing 9.9. Obsługa zdarzenia w pliku src/main.js

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
  el: "#app",
  template: `<div class="text-center p-3">
    <h1 class="bg-secondary text-white p-3">
      Liczba wciśnięć: {{ counter }}
    </h1>
    <button class="btn btn-secondary" v-on:click="handleClick">
      Wciśnij mnie
    </button>
  </div>`,
  data: {
    counter: 0
  },
  methods: {
    handleClick() {
      this.counter++;
    }
  }
})
```

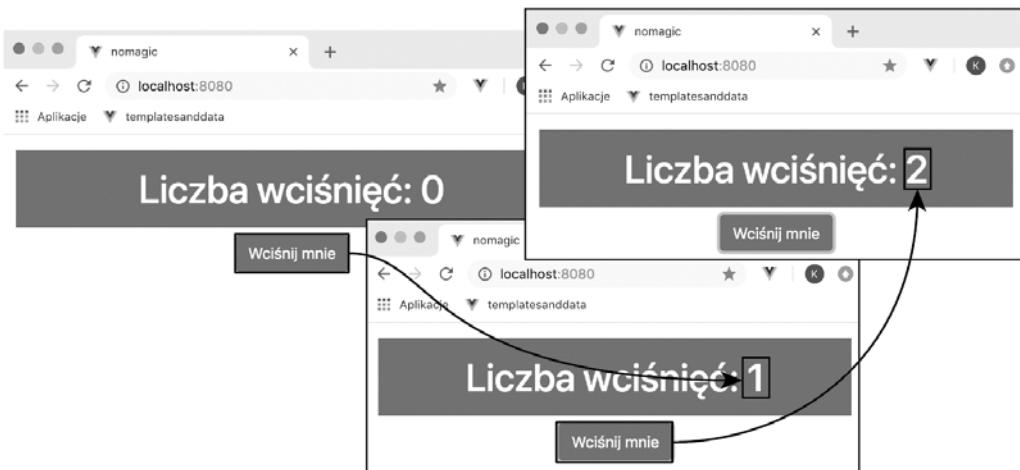
```

        }
    });
}
);

```

Atrybut, który dodaję do elementu button, to **dyrektywa**. Jest to mechanizm Vue.js, który pozwala na rozszerzenie funkcjonalności elementu HTML. W tym przypadku skorzystałem z dyrektywy `v-on`, aby umożliwić obsługę zdarzenia `click` przycisku. Wiążę je z metodą `handleClick`, która została zadeklarowana we właściwości `methods` obiektu konfiguracyjnego Vue. Obiekt `methods` definiuje funkcję `handleClick`, która zwiększa wartość zmiennej `counter`.

Skoro Vue.js działa reaktywnie, zmiana wartości zmiennej `counter` pociąga za sobą ponowną ewaluację wiązania danych w elemencie `h1`, a w konsekwencji — zmianę komunikatu wyświetlanego użytkownikowi (rysunek 9.5).



Rysunek 9.5. Obsługa zdarzenia

Modyfikacja komunikatu

Ostatnią zmianą niezbędną do zrównania tego przykładu z poprzednim jest wyświetlenie innego komunikatu, gdy użytkownik nie zdąży jeszcze kliknąć przycisku. Aby osiągnąć ten sam efekt za pomocą obiektu Vue, wprowadziłem zmiany w listingu 9.10.

Listing 9.10. Wyświetlanie warunkowego komunikatu w pliku src/main.js

```

require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue"
new Vue({
    el: "#app",
    template: `<div class="text-center p-3">
        <h1 class="bg-secondary text-white p-3">
            {{ message }}
        </h1>
        <button class="btn btn-secondary" v-on:click="handleClick">
            Wciśnij mnie
        </button>
    </div>`,
}
);

```

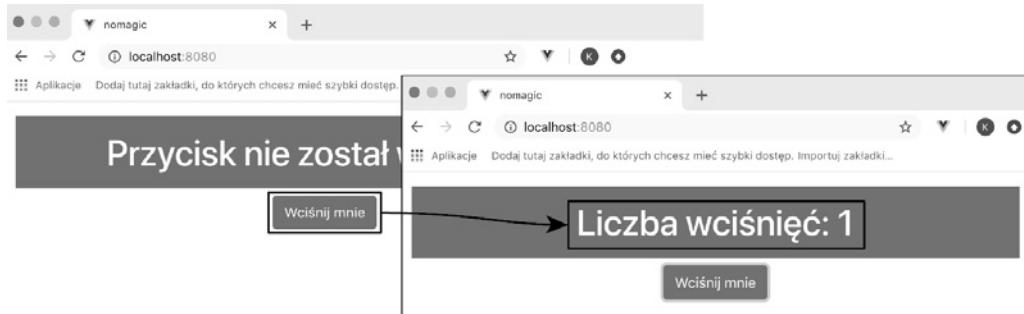
```

data: {
    counter: 0
},
methods: {
    handleClick() {
        this.counter++;
    }
},
computed: {
    message() {
        return this.counter == 0 ?
            "Przycisk nie został wciśnięty" :
            `Liczba wciśnięć: ${this.counter}`;
    }
}
);

```

Właściwość computed określa wartości danych, które wymagają dokonania pewnych operacji (często obliczeń), aby uzyskać rezultat. W tym przykładzie korzystamy z wyrażenia, aby sprawdzić wartość właściwości counter i na podstawie tego sprawdzenia zwracamy tekst. Nowo obliczona właściwość nazywa się message i wyświetlamy jej wartość w elemencie h1, umieszczając nazwę w wiązaniu danych, tak jak wcześniej zrobiliśmy to ze zmiennej counter.

Gdy użytkownik kliką przycisk, dyrektywa v-on wywołuje metodę handleClick, która zwiększa wartość zmiennej counter. Vue.js wykrywa zmianę i ponownie ewaluuje wyrażenia wiążące danych zdefiniowane w treści szablonu. Otrzymujemy nową wartość dla obliczonej właściwości, która ta wartość jest wyświetlana użytkownikowi za pomocą elementu h1 (rysunek 9.6).

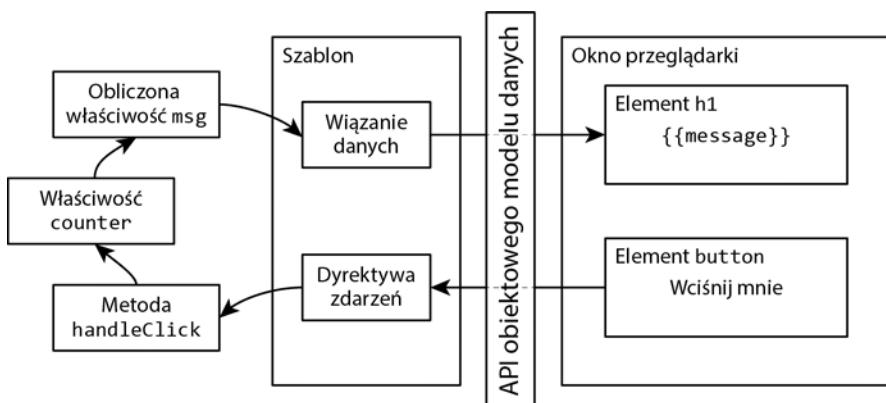


Rysunek 9.6. Zastosowanie właściwości obliczanej

Zasada działania obiektu Vue

Obiekt Vue ma taki sam zakres funkcjonalny jak kod zastosowany w listingu 9.6, jednak dzięki Vue.js aplikacja ma znacznie lepszą strukturę, co pozwala na łatwiejsze zarządzanie i rozwijanie oprogramowania niż w przypadku pracy z modelem DOM. Trudno docenić Vue.js, patrząc tylko na obiekt z listingu 9.10, jednak już w tym przykładzie widać strukturę, w której istnieje podział na treść, logikę i dane aplikacji jak na rysunku 9.7.

Szablon obiektu Vue dostarcza treść HTML, którą widzi użytkownik. Ponadto otrzymujemy wiązanie danych i dyrektywę zdarzenia, która zapewnia interaktywność aplikacji. Dane aplikacji zawierają właściwość counter i właściwość obliczoną msg, a logika, która łączy je ze sobą, jest zawarta w metodzie handleClick.



Rysunek 9.7. Struktura aplikacji

Komponenty w praktyce

Obiekt Vue pomaga oddzielić różne aspekty aplikacji, jednak rezultat wcale nie jest satysfakcjonujący. W większości projektów obiekt Vue jest używany do skonfigurowania aplikacji, ale jej zasadnicze funkcje są definiowane za pomocą **komponentów**, które rozszerzają możliwości obiektu Vue. Ponadto komponenty mogą być definiowane w plikach zawierających użyteczną mieszankę różnego rodzaju treści.

Komponenty definiuje się w plikach z rozszerzeniem .vue. Tuż po utworzeniu projekt zawiera plik *App.vue*; w listingu 9.11 przedstawiam jego nową zawartość.

Listing 9.11. Zamiana zawartości pliku src/App.vue

```

<script>
export default {
    template: `<div class="text-center p-3">
        <h1 class="bg-secondary text-white p-3">
            {{ message }}
        </h1>
        <button class="btn btn-secondary" v-on:click="handleClick">
            Wciśnij mnie
        </button>
    </div>`,
    data: function () {
        return {
            counter: 0
        }
    },
    methods: {
        handleClick() {
            this.counter++;
        }
    },
    computed: {
        message() {
            return this.counter == 0 ?
                "Przycisk nie został wciśnięty" :
                "Liczba wciśnięć: " + this.counter;
        }
    }
}
</script>

```

Komponent przedstawiony w listingu 9.11 zawiera te same funkcje co w przypadku definicji w obiekcie Vue, z wyjątkiem tego, że właściwości języka JavaScript są zdefiniowane w elemencie `script`. Na pierwszy rzut oka może wydawać się, że nie jest to duże usprawnienie, ale jest to dobry przykład na to, że komponenty są obiektami Vue, podobnymi do tego z pliku `main.js`. Komponenty pozwalają na tworzenie aplikacji z wielu komponentów, które można łączyć, aby zapewnić użytkownikowi zaawansowane możliwości.

Kod w języku JavaScript jest zawarty w elemencie `script` z tym samym zakresem funkcjonalnym co obiekt Vue. W porównaniu z wersją ze zwykłym obiektem Vue są dwie zmiany. Pierwsza z nich dotyczy konieczności wyeksportowania obiektu konfiguracji:

```
...
export default {
    // ...właściwości konfiguracji zostały pominięte...
}
...
...
```

W ten sposób obiekt może zostać zimportowany ze swojego modułu, co osiągamy w listingu 9.12. Druga zmiana polega na wyrażeniu właściwości `data` w formie funkcji:

```
...
data: function () {
    return {
        counter: 0
    }
},
...
...
```

Właściwość `data` otrzymuje funkcję, która zwraca obiekt. Właściwości tego obiektu dostarczają wartości danych dla komponentu. Jest to nieco dziwne podejście, ale Vue.js wymaga tego, aby zapewnić, że każdy komponent dysponuje własnymi danymi.

Rejestracja i wdrażanie komponentu

Komponenty należy zarejestrować w Vue.js przed pierwszym użyciem. Z reguły odbywa się to w pliku `main.js`. W listingu 9.12 zamieniam konfigurację obiektu Vue w taki sposób, aby skorzystać z nowego komponentu.

Listing 9.12. Konfiguracja komponentu w pliku src/main.js

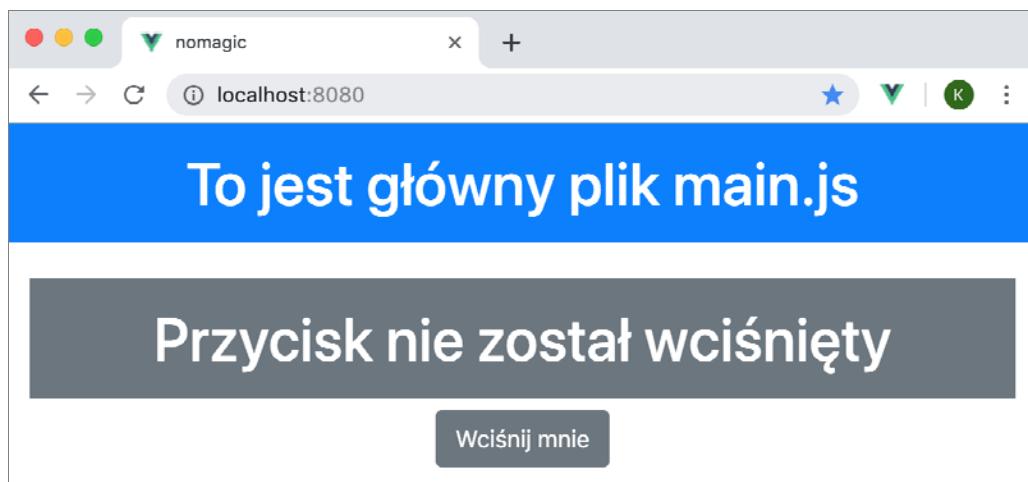
```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue";
import MyComponent from "./App";
new Vue({
    el: "#app",
    components: { "custom": MyComponent },
    template: `<div class="text-center">
        <h1 class="bg-primary text-white p-3">
            To jest główny plik main.js
        </h1>
        <custom />
    </div>`
});
```

Pierwsza nowa instrukcja korzysta ze słowa kluczowego `import`, które importuje moduł z pliku `App.vue` i nadaje mu nazwę `MyComponent`. Zgodnie z konwencją nazwa pliku powinna być używana jako nazwa importu, ale możesz skorzystać z takiej nazwy, która wydaje Ci się najbardziej sensowna.

Kolejna zmiana polega na dodaniu właściwości `components` do konfiguracji obiektu Vue. Ta właściwość powoduje otrzymanie obiektu, który dostarcza do Vue.js listę elementów HTML i skojarzonych z nimi komponentów. W tym przypadku skonfigurowałem właściwość `components` tak, aby element `custom` został skojarzony z komponentem `MyComponent`, do którego odwołuję się w pliku `App.vue`.

Zmieniłem także treść szablonu (`template`), dzięki czemu zawiera on prostszy zbiór elementów HTML, wśród których zawarty jest element `custom`.

Gdy obiekt Vue podczas przetwarzania swojego szablonu napotka element `custom`, element ten zostanie zastąpiony przez treść zdefiniowaną w ramach komponentu w pliku `App.vue`. W rezultacie użytkownik otrzyma połączenie kodu HTML z właściwości `template` komponentu, a także z właściwości `template` obiektu Vue. Po zapisaniu zmian przeglądarka zostanie odświeżona, a Ty zobaczysz treść jak na rysunku 9.8.



Rysunek 9.8. Komponent w praktyce

Obiekt Vue zazwyczaj nie wyświetla swojej własnej treści. Zgodnie z konwencją kod HTML umieszcza się w szablonie komponentu, jak w listingu 9.13.

Listing 9.13. Usuwanie treści szablonu w pliku `src/main.js`

```
require('../node_modules/bootstrap/dist/css/bootstrap.min.css')
import Vue from "vue";
import MyComponent from "./App";
new Vue({
  el: "#app",
  components: { "custom": MyComponent },
  template: "<custom />"
});
```

Obiekt Vue zastępuje element `div` w pliku `index.html` treścią elementu `custom`, zadeklarowaną w elemencie `template` komponentu. Po zapisaniu zmian zobaczysz kompletny przykład aplikacji, jak na rysunku 9.9.

Oddzielanie szablonu od kodu JavaScript

Szablon komponentu można definiować za pomocą właściwości `template`, jednak warto rozważyć inne podejście, które preferuje wielu programistów. W listingu 9.14 usuwam właściwość `template` i zastępuję ją elementem `template`, zgodnie z typową metodą deklarowania komponentów.



Rysunek 9.9. Kompletna przykładowa aplikacja

Listing 9.14. Zastosowanie elementu template w pliku src/App.vue

```
<template>
  <div class="text-center p-3">
    <h1 class="bg-secondary text-white p-3">
      {{ message }}
    </h1>
    <button class="btn btn-secondary" v-on:click="handleClick">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
export default {
  data: function () {
    return {
      counter: 0
    },
  methods: {
    handleClick() {
      this.counter++;
    }
  },
  computed: {
    message() {
      return this.counter == 0 ?
        "Przycisk nie został wcisnięty" : `Liczba wciśnięć: ${this.counter}`;
    }
  }
}
</script>
```

Element template zawiera tę samą treść co właściwość template, ale to podejście pozwala na traktowanie kodu w większości edytorów jako HTML, co owocuje wsparciem na poziomie edytora, takim jak autouzupełnianie nazw czy oznaczanie błędów. Nic się nie zmienia w wyglądzie i zachowaniu — ta zmiana ma na celu uczynienie procesu deweloperskiego łatwiejszym.

Tworzenie odrębnych plików JavaScript i HTML

Nie każdy lubi łączyć kod JavaScript z HTML-em w tym samym pliku. Jeśli tak jest i w Twoim przypadku, to możesz tworzyć odrębne pliki na oba rodzaje kodu. W listingu 9.15 przedstawiam przykład pliku zawierającego wyłącznie kod HTML.

Listing 9.15. Zawartość pliku *src/App.html*

```
<div class="text-center p-3">
  <h1 class="bg-secondary text-white p-3">
    {{ message }}
  </h1>
  <button class="btn btn-secondary" v-on:click="handleClick">
    Wciśnij mnie
  </button>
</div>
```

W listingu 9.16 usuwam zawartość elementu *template* i dodaję atrybut *src*, który informuje Vue.js, że treść komponentu można znaleźć w odrębnym pliku HTML.

Listing 9.16. Odwołanie do pliku HTML w pliku *src/App.vue*

```
<template src="./App.html" />
<script>
export default {
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    handleClick() {
      this.counter++;
    }
  },
  computed: {
    message() {
      return this.counter == 0 ?
        "Przycisk nie został wciśnięty" :
        `Liczba wciśnięć: ${this.counter}`;
    }
}
</script>
```

Podsumowanie

W tym rozdziale wyjaśniłem, że Vue.js korzysta z tego samego API modelu DOM do tworzenia aplikacji webowych, które może być używane w czystym, „waniliowym” JavaScriptie. Za Vue.js nie kryje się bowiem żadna magia. Jak przekonasz się w kolejnych rozdziałach, choć można bezpośrednio korzystać z API modelu DOM, Vue.js zapewnia dużo przydatnych funkcji, które pozwalają na tworzenie bardziej eleganckich, możliwych do zarządzania i skalowalnych funkcji. W kolejnym rozdziale objaśnię strukturę projektów Vue.js i działanie narzędzi deweloperskich.

ROZDZIAŁ 10.



Projekty i narzędzia Vue.js

W rozdziale 9. przedstawiłem pokrótce zasadę działania aplikacji Vue.js, aby dalsze czynności wykonywane w niniejszej książce umieścić w pewnym kontekście. Projekt został utworzony za pomocą pakietu `@vue-cli`. W trakcie realizacji projektu korzystaliśmy także z innych narzędzi. W tym rozdziale zajmę się objaśnieniem struktury projektu Vue.js i przedstawieniem każdego z używanych narzędzi. Podsumowuje to tabela 10.1.

Tabela 10.1. Umiejscowienie projektów Vue.js w szerszym kontekście

Pytanie	Odpowiedź
Czym są projekty?	Projekty utworzone za pomocą pakietu <code>@vue-cli</code> są przeznaczone do tworzenia zaawansowanych aplikacji.
Dlaczego są użyteczne?	Ten rodzaj projektu zawiera zestaw narzędzi, które upraszczają tworzenie aplikacji w Vue.js i pozwalają na łatwe użycie zaawansowanych funkcji oferowanych przez Vue.js.
Jak się z nich korzysta?	Projekt jest tworzony za pomocą paczki <code>@vue/cli</code> . Seria pytań zadawanych w trakcie tworzenia projektu pozwala określić jego początkową zawartość.
Czy są jakieś pułapki lub ograniczenia?	Ten rodzaj projektu może być „armatą na wróbla”, jeśli chcesz tylko eksperymentować z Vue.js.
Czy są jakieś rozwiązania alternatywne?	Möżesz tworzyć własne projekty bez pakietu <code>@vue/cli</code> . W takiej sytuacji możesz zastosować własny zestaw narzędzi, jednak ten proces może zająć sporo czasu.

Tabela 10.2 przedstawia podsumowanie rozdziału.

Tworzenie projektu aplikacji Vue.js

Większość nowoczesnych frameworków funkcjonujących po stronie klienta zawiera własne narzędzia deweloperskie i Vue.js nie jest tu wyjątkiem. Oznacza to, że możesz utworzyć projekt i zacząć tworzyć aplikację za pomocą narzędzi, które zostały przygotowane specjalnie dla Vue.js i dobrze przetestowane przez liczną społeczność.

Tabela 10.2. Podsumowanie rozdziału

Problem	Rozwiążanie	
Utwórz nowy projekt.	Skorzystaj z polecenia <code>vue create</code> i wybierz funkcje aplikacji wymagane przez Twoją aplikację.	10.1
Uruchom narzędzia deweloperskie.	Skorzystaj z polecenia <code>npm run serve</code> .	10.2 – 10.8
Unikaj typowych błędów kodu i treści.	Skorzystaj z funkcji <code>linter</code> .	10.9 – 10.11
Zdebuguj aplikację.	Skorzystaj z rozszerzenia Vue Devtools przeglądarki.	10.12
Zmień ustawienia narzędzi deweloperskich.	Dodaj do projektu plik <code>vue.config.js</code> z ustawieniami, których potrzebujesz.	10.13
Przygotuj aplikację do wdrożenia.	Skorzystaj z polecenia <code>npm run build</code> .	10.14 – 10.19

Pakiet `@vue/cli` zainstalowany w rozdziale 1. jest dostarczany przez zespół Vue.js w celu uproszczenia procesu tworzenia projektów Vue.js i instalacji wszystkich niezbędnych narzędzi deweloperskich. Wykonaj polecenie z listingu 10.1, aby utworzyć nowy projekt za pomocą polecenia `vue create`, dostępnego z poziomu pakietu `@vue/cli`.

Listing 10.1. Tworzenie nowego projektu

```
vue create projecttools
```

- **Uwaga** W trakcie pisania niniejszej książki pakiet `@vue/cli` był dostępny w wersji beta. W związku z możliwymi zmianami warto zapoznać się z erratum dostępne w repozytorium oryginalnego wydania książki pod adresem <https://github.com/Apress/pro-vue-js-2>.

Polecenie `vue create` wywołuje interaktywny proces związany z wyborem opcji tworzenia projektu (rysunek 10.1).

```
Vue CLI v3.3.0
? Please pick a preset: (Use arrow keys)
> default (babel, eslint)
Manually select features
```

Rysunek 10.1. Wybór ustawień projektu

Tekst z rysunku nie jest zbyt czytelny, dlatego nieco go objaśnię. Polecenie `vue create` zaczyna proces od wybrania zestawu ustawień wstępnych (ang. *preset*). W Vue.js jest dostępny jeden taki zestaw — domyślny (`default`) — który pozwoli nam utworzyć konfigurację taką samą jak we wcześniejszych rozdziałach. Z tej konfiguracji korzystam również w dalszej części książki, ponieważ pozwala mi ona pokazać, jak można dodawać różne komponenty do projektu. Nie ma nic niezwykłego w korzystaniu z narzędzi, które są wygodne, o ile tylko rozumiesz, co niesie ze sobą ich użycie. Alternatywą dla ustawień wstępnych jest ręczny wybór funkcji. Skorzystaj ze strzałek, aby wybrać opcję *Manually Select Features* (wybierz opcje ręcznie), wciśnij `Enter`, a zobaczysz listę funkcji pokazanych na rysunku 10.2.

Skorzystaj z klawiszy strzałek, aby poruszać się w góre i w dół listy funkcji, a ze spacji — aby je włączyć lub wyłączyć. Tabela 10.3 opisuje dostępne funkcje, które mają bezpośredni związek z tworzeniem aplikacji w Vue.js — w tym lub kolejnych rozdziałach.

```

Vue CLI v3.3.0
? Please pick a preset: Manually select features
? Check the features needed for your project: (Press <spa
ce> to select, <a> to toggle all, <i> to invert selection
)
> Babel
  o TypeScript
  o Progressive Web App (PWA) Support
  o Router
  o Vuex
  o CSS Pre-processors
  ● Linter / Formatter
  o Unit Testing
  o E2E Testing

```

Rysunek 10.2. Funkcje projektu dostępne do ręcznego wyboru**Tabela 10.3.** Funkcje projektu Vue.js

Nazwa	Opis
Babel	Babel tłumaczy kod źródłowy JavaScript zawierający nowe, nieobsługiwane przez wszystkie przeglądarki konstrukcje na postać, która może być wykonana również w starszych przeglądarkach.
TypeScript	TypeScript to rozszerzenie języka JavaScript, które dodaje użyteczne mechanizmy, takie jak statyczne typowanie. Dzięki temu korzystanie z JavaScriptu przypomina bardziej C# lub Java. TypeScript nie musi być używany w Vue.js, jednak niektórzy nie przepadają za pracą ze zwykłym JavaScriptem. Więcej szczegółów znajdziesz na stronie https://vuejs.org/v2/guide/typescript.html .
PWA Support (obsługa PWA)	Progresywnych aplikacji webowych można używać nie tylko w przeglądarkach komputerów stacjonarnych, ale również w specjalnej formie na urządzeniach mobilnych. Dzięki specjalnej funkcji tzw. <i>service workers</i> (dosł. pracownicy usługi) możliwe jest korzystanie z niektórych funkcji bez połączenia z internetem. PWA nie jest ograniczone do Vue.js. Mimo ciągłego rozwoju tej technologii pomijam ją w tej książce z uwagi na wciąż niewystarczające przygotowanie. Więcej informacji znajdziesz na stronie https://developer.mozilla.org/en-US/Apps/Progressive .
Router	Ta funkcja instaluje pakiet Vue Router, którego używamy do nakreślenia struktury w większych aplikacjach za pomocą adresu URL przeglądarki. Przykład trasowania znalazł się w aplikacji Sklep sportowy w pierwszej części książki. Trasowanie opisuję szczegółowo w rozdziałach 23. – 25.
Vuex	Ten pakiet pozwala na tworzenie współdzielonych magazynów danych. Również ten pakiet pojawił się w aplikacji Sklep sportowy. Szczegółowo zajmę się nim w rozdziale 20.
CSS Pre-processors (preprocesory CSS)	Preprocesory CSS, takie jak Sass czy Less, ułatwiają tworzenie złożonych stylów CSS, co znaczco pomaga, gdy nie korzystasz z framework'a CSS lub chcesz rozszerzyć jego możliwości. Jeśli nie chcesz korzystać z framework'a CSS, nie musisz wcale używać preprocesora — więcej na ten temat w kolejnych rozdziałach.
Linter/Formatter	Linter/Formatter to funkcja, która instaluje pakiet odpowiedzialny za weryfikację kodu i treści w celu sprawdzenia zgodności z dobrymi praktykami (więcej na ten temat w podrozdziale „Zastosowanie lintera”).
Unit Testing (testy jednostkowe)	Ta funkcja instaluje narzędzia do testów jednostkowych (https://cli.vuejs.org/config/#unit-testing).
E2E Testing (testy E2E)	Ta funkcja instaluje narzędzia do testowania typu end-to-end (od początku do końca — https://cli.vuejs.org/config/#e2e-testing).

Na potrzeby tego rozdziału wybierz funkcje Babel, Router, Vuex i Linter/Formatter (rysunek 10.3).

Vue CLI v3.3.0

- ? Please pick a preset: Manually select features
- ? Check the features needed for your project:
 - Babel
 - TypeScript
 - Progressive Web App (PWA) Support
 - Router
 - > Vuex
 - CSS Pre-processors
 - Linter / Formatter
 - Unit Testing
 - E2E Testing

Rysunek 10.3. Wybór funkcji w przykładowym projekcie

Po wybraniu żądanych funkcji wciśnij *Enter*. Zostaną wyświetcone dodatkowe opcje, które pozwolą skonfigurować wybrane przed chwilą funkcje.

Konfiguracja lintera

Na początku kreator skonfiguruje linter, co pokazano na rysunku 10.4. Jeśli zostanie wyświetlone pytanie o tryb historii (ang. *history mode*) routera, wprowadź odpowiedź *n* (*No*).

Vue CLI v3.3.0

- ? Please pick a preset: Manually select features
- ? Check the features needed for your project: Babel, Router, Vuex, Linter
- ? Use history mode for router? (Requires proper server setup for index fallback in production) No
- ? Pick a linter / formatter config: (Use arrow keys)
 - > ESLint with error prevention only
 - ESLint + Airbnb config
 - ESLint + Standard config
 - ESLint + Prettier

Rysunek 10.4. Konfiguracja lintera

Wybierz opcję *ESLint with error prevention only* (ESLint tylko z zapobieganiem błędów). Linter weryfikuje projekt pod kątem zgodności ze standardami programowania i to pytanie pozwala wybrać ów standard. Więcej na temat poszczególnych opcji znajdziesz w dalszej części rozdziału. Kolejna opcja pozwala wybrać moment zastosowania lintera (rysunek 10.5).

Ta opcja jest ustawiana, jeśli korzystamy z lintera w projekcie. Wybierz opcję *Lint on save* (weryfikuj przy zapisie), co sprawi, że projekt będzie weryfikowany przy każdym zapisie w nim pliku.

Zakończenie konfiguracji projektu

Po wybraniu funkcji projektu zostaniesz zapytany o styl jego konfiguracji (rysunek 10.6).

```
Vue CLI v3.3.0
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Vuex, Linter
? Use history mode for router? (Requires proper server setup for index fallback in production) No
? Pick a linter / formatter config: Basic
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection)
  ⚡ Lint on save
  ○ Lint and fix on commit
```

Rysunek 10.5. Ustawianie momentu zastosowania lintera

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Vuex, Linter
? Use history mode for router? (Requires proper server setup for index fallback in production) No
? Pick a linter / formatter config: Basic
? Pick additional lint features: (Press <space> to select, <a> to toggle all, <i> to invert selection) Lint on save

? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? (Use arrow keys)
  ⚡ In dedicated config files
  ○ In package.json
```

Rysunek 10.6. Wybór stylu pliku konfiguracyjnego

Masz do wyboru dwie opcje. Możesz skorzystać z odrębnych plików konfiguracyjnych dla każdej funkcji, która takiej konfiguracji wymaga. Możesz też załączyć wszystkie ustawienia w pliku *package.json*, który jest używany do śledzenia pakietów wymaganych przez aplikację.

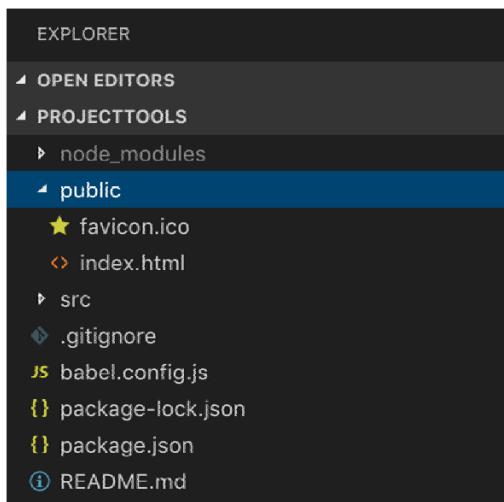
Wybierz opcję *In package.json* (w pliku *package.json*), która dodaje ustawienia do pliku *package.json*. Jest to takie samo podejście jak w pozostałych rozdziałach tej książki.

Wciśnij klawisz *Enter*. Ostatnim krokiem będzie określenie, czy chcesz zapisać konfigurację do przyszłego użycia w innych projektach. Wciśnij *Enter*, aby wybrać opcję *No*, ponieważ nie będziemy korzystać z tej konfiguracji w kolejnych rozdziałach.

Teraz, gdy odpowiedziałeś na wszystkie pytania, projekt zostanie utworzony, a pakiety zostaną pobrane i zainstalowane. Projekt Vue.js wymaga wielu pakietów, a początkowa konfiguracja może zająć dłuższą chwilę.

Omówienie struktury projektu

Skorzystaj z wybranego edytora i otwórz katalog *projecttools*. Powinieneś zobaczyć strukturę projektu jak na rysunku 10.7. Rysunek przedstawia sposób wyświetlania plików utworzonych za pomocą polecenia *vue create*. W zależności od konkretnego edytora i aktualnej wersji szablonu projektu widok może się nieco różnić od przedstawionego na tym rysunku.



Rysunek 10.7. Struktura przykładowego projektu

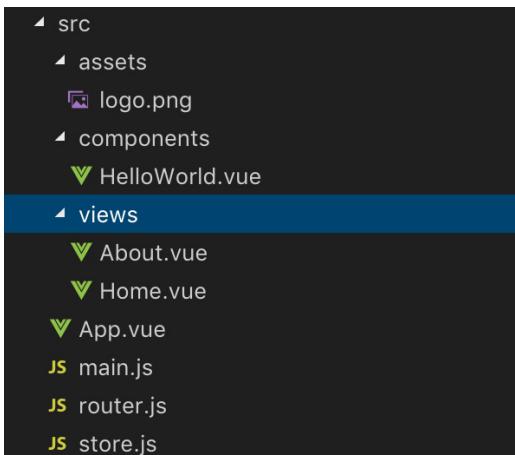
W tabeli 10.4 opisano najważniejsze pliki i katalogi projektu, a także funkcje, które one pełnią.

Tabela 10.4. Najważniejsze pliki i katalogi projektu aplikacji Vue.js

Nazwa	Opis
<i>node_modules</i>	Ten katalog zawiera pakiety wymagane przez aplikację i narzędzia deweloperskie, jak opisano w punkcie „Omówienie katalogu pakietów”.
<i>public</i>	Ten katalog zawiera statyczne zasoby projektu, takie jak obrazki, które nie są dołączane do paczki plików przeglądarki (więcej na ten temat w podrozdziale „Omówienie narzędzi deweloperskich”).
<i>src</i>	Ten katalog zawiera aplikację Vue.js i jej zasoby. To właśnie na tym katalogu koncentruje się większość naszej pracy programistycznej, co opisuję w punkcie „Omówienie katalogu z kodem źródłowym”.
<i>.gitignore</i>	Ten plik zawiera listę plików i katalogów, które są wyłączone z obsługi systemu kontroli wersji Git.
<i>babel.config.js</i>	Ten plik zawiera ustawienia kompilatora Babel (opisuję je szerzej w podrozdziale „Omówienie narzędzi deweloperskich”).
<i>package.json</i>	Ten plik zawiera listę pakietów wymaganych przez Vue.js, a także polecenie używanych przez narzędzia deweloperskie (więcej na ten temat w punkcie „Omówienie katalogu pakietów”).
<i>package-lock.json</i>	Ten plik zawiera pełną listę pakietów wymaganych przez projekt i jego zależności. W ten sposób masz pewność, że otrzymasz ten sam zestaw pakietów po wykonaniu polecenia <code>npm install</code> .

Omówienie katalogu z kodem źródłowym

Katalog *src* stanowi najważniejszą część każdego projektu Vue.js i zawiera kod HTML aplikacji, kod źródłowy i inne zasoby. To właśnie na tym katalogu skupia się większość sesji programistycznych. Rysunek 10.8 przedstawia zawartość katalogu *src* dla projektu utworzonego za pomocą funkcji wybranych na początku rozdziału.



Rysunek 10.8. Zawartość katalogu src

Struktura katalogu src zacznie się komplikować w miarę rozwoju projektu. Na razie treści zastępcze zapewniają wystarczająco dużo funkcji, abyśmy byli w stanie zacząć pracę. Nie zamierzam opisywać szczegółowo struktury katalogu src, ponieważ jest ona omawiana w pozostałych fragmentach książki. Dla formalności w tabeli 10.5 omawiam za to pliki, które są dodawane w momencie tworzenia nowego projektu.

Tabela 10.5. Początkowa zawartość katalogu src

Nazwa	Opis
assets	Ten katalog jest używany do przechowywania statycznych zasobów wymaganych przez aplikację, dołączanych do paczki w procesie budowania (por. „Omówienie narzędzi deweloperskich”). Polecenie vue create dodaje do tego katalogu plik obrazka — logo Vue.js.
components	Ten katalog jest używany do przechowywania komponentów aplikacji. Aplikacja może zawierać wiele komponentów, a dodatkowo tworzone podkatalogi pomagają grupować powiązane komponenty. Polecenie vue create dodaje komponent zastępczy o nazwie HelloWorld.
views	Ten katalog zawiera komponenty, które są wyświetlane za pomocą funkcji trasowania URL. Nie korzystam z tej zasady w niniejszej książce, dlatego wszystkie komponenty umieszczam w katalogu components.
App.vue	W tym pliku znajduje się główny komponent aplikacji. To właśnie w tym miejscu zaczyna się cykl życia aplikacji Vue.js.
main.js	Ten plik tworzy obiekt Vue, co opisuję w rozdziale 9.
router.js	Ten plik odpowiada za konfigurację systemu trasowania URL, dzięki któremu możemy kontrolować, które komponenty są wyświetlane użytkownikowi. Funkcję trasowania opisuję szczegółowo w rozdziałach 22. – 24., w których podążam nieco inną ścieżką niż dotychczas — tworzę katalog router i zawarty w nim plik index.js. W tym pliku przechowuję instrukcje konfiguracyjne. Dzięki takiemu podejściu można podzielić konfigurację na wiele plików, które są traktowane jak pojedynczy moduł JavaScript.
store.js	Ten plik jest używany do skonfigurowania magazynu danych odpowiedzialnego za współdzielenie danych w całej aplikacji. Magazyny danych opisuję w rozdziale 20., gdzie korzystam z odrębnego katalogu store i pliku index.js, co pozwala mi na podział konfiguracji na wiele plików.

Omówienie katalogu pakietów

Tworzenie aplikacji w języku JavaScript opiera się na bogatym ekosystemie pakietów, począwszy od tych, które będą przesyłane do przeglądarki, aż po małe pakiety wspomagające proces samego tworzenia. Vue.js wymaga wielu pakietów. Przykładowy projekt utworzony na początku rozdziału wymaga ponad 900 pakietów.

Pomiędzy tymi pakietami istnieje skomplikowana hierarchia zależności, którą nie sposób zarządzać ręcznie. Do tego celu wykorzystuje się program zwany **menedżerem pakietów** (ang. *package manager*). Projekty Vue.js można tworzyć za pomocą dwóch różnych menedżerów: **NPM** (ang. *Node Package Manager*), zainstalowanego razem z Node.js w rozdziale 1., lub **Yarn** — modnego ostatnio konkurenta, zaprojektowanego, aby usprawnić zarządzanie pakietami. W tej książce dla uproszczenia korzystam z NPM.

-
- **Wskazówka** W niniejszej książce korzystam z NPM, ale jeśli chcesz użyć Yarna, to więcej informacji na jego temat znajdziesz pod adresem <https://yarnpkg.com/>.
-

Po utworzeniu projektu menedżer pakietów otrzymuje listę pakietów wymaganych przez Vue.js. Menedżer analizuje każdy pakiet, aby ustalić listę pakietów, od których dany pakiet zależy. Proces jest wykonywany rekurencyjnie, aby uzyskać pełne drzewo zależności pakietów. Następnie menedżer pobiera i instaluje wszystkie pakiety w katalogu *node_modules*.

Początkowy zbiór pakietów jest zdefiniowany w pliku *package.json* za pomocą właściwości *dependencies* i *devDependencies*. Właściwość *dependencies* pozwala na określenie pakietów wymaganych przez aplikację do uruchomienia. Naturalnie sekcja ta może zawierać różną treść w zależności od projektu, natomiast w moim przykładowym projekcie znajdziesz sekcję o następującej postaci:

```
...
"dependencies": {
  "vue": "^2.5.16",
  "vue-router": "^3.0.1",
  "vuex": "^3.0.1"
},
...
```

Tylko trzy pakiety muszą znaleźć się w sekcji *dependencies* projektu Vue.js: pakiet *vue*, który zawiera główne funkcje Vue.js, pakiet *vue-router* odpowiedzialny za nawigację (por. rozdziały 22. – 24.), a także pakiet *vuex* odpowiedzialny za magazyn danych (rozdział 20.). Dla każdego pakietu w pliku *package.json* znajdą się numery dopuszczalnych wersji, zgodnie z formatem opisany w tabeli 10.6.

Tabela 10.6. System numerowania wersji pakietów

Format	Opis
2.5.16	Dopuszczalny jest jedynie pakiet o wersji identycznej z podaną.
*	Dopuszczalna jest dowolna wersja pakietu.
>2.5.16 >= 2.5.16	Dopuszczalny jest pakiet o wersji nowszej bądź nowszej lub równej podanej.
<2.5.16 <= 2.5.16	Dopuszczalny jest pakiet o wersji starszej bądź starszej lub równej podanej.
-2.5.16	Poprzedzenie numeru wersji tylde (znakiem ~) dopuszcza instalację pakietu o identycznej wersji głównej (2) i wersji dodatkowej (5). Ostatnia część wersji może być większa od podanej (np. 2.5.20 i 2.5.16 są poprawne, ale 2.6.0 już nie).
^2.5.16	Poprzedzenie numeru wersji znakiem karety (^) dopuszcza instalację pakietu o identycznej wersji głównej. Pozostałe elementy wersji mogą być dowolne (np. 2.5.20 i 2.6.0 są poprawne, ale 3.0.0 już nie).

Numery wersji określone w sekcji `dependencies` pliku `package.json` dopuszczają inne wersje dodatkowe (ang. *minor*) i łatki (ang. *patch*). Elastyczność w dopuszczaniu wersji jest jeszcze ważniejsza w przypadku sekcji `devDependencies`, gdzie umieszczamy listę pakietów niezbędnych w czasie tworzenia aplikacji, ale pomijanych na etapie jej produkcyjnego działania. Oto sekcja `devDependencies` w moim projekcie:

```
...
"devDependencies": {
  "@vue/cli-plugin-babel": "^3.0.0-beta.15",
  "@vue/cli-plugin-e2e-cypress": "^3.0.0-beta.15",
  "@vue/cli-plugin-eslint": "^3.0.0-beta.15",
  "@vue/cli-plugin-unit-mocha": "^3.0.0-beta.15",
  "@vue/cli-service": "^3.0.0-beta.15",
  "@vue/test-utils": "^1.0.0-beta.16",
  "chai": "^4.1.2",
  "vue-template-compiler": "^2.5.16"
},
...
```

Te pakiety zapewniają dostęp do narzędzi deweloperskich. Jak wspomniałem na początku rozdziału, pakiet `@vue/cli` jest dostępny w wersji beta, co widać po numerach wersji pakietów.

Różnice między lokalnymi i globalnymi pakietami

Menedżery pakietów potrafią instalować pakiety w zasięgu pojedynczego projektu (lokalnie) lub dostępne zawsze (globalnie). Nieliczne pakiety muszą być instalowane globalnie. W naszym przypadku takim pakietem jest `@vue/cli`, zainstalowany w rozdziale 1. Pakiet ten musi być zainstalowany globalnie, ponieważ korzystam z niego w celu tworzenia nowych projektów Vue.js. Pakiety niezbędne dla poszczególnych projektów są instalowane lokalnie, w katalogu `node_modules`.

Wszystkie pakiety niezbędne do tworzenia aplikacji są automatycznie pobierane i instalowane w katalogu `node_modules` w momencie tworzenia projektu. Tabela 10.7 przedstawia niektóre polecenia narzędzia NPM, które mogą się przydać w czasie tworzenia aplikacji. Wszystkie polecenia powinny być wykonywane w katalogu projektu, czyli tym, w którym znajduje się plik `package.json`.

Tabela 10.7. Użyteczne polecenia narzędzia NPM

Polecenie	Opis
<code>npm install</code>	To polecenie instaluje lokalnie pakiety zdefiniowane w pliku <code>package.json</code> .
<code>npm install package@version</code>	To polecenie instaluje lokalnie pakiet o podanej wersji i aktualizuje plik <code>package.json</code> , umieszczając informacje o pakiecie w sekcji <code>dependencies</code> .
<code>npm install --save-dev package@version</code>	To polecenie instaluje lokalnie pakiet o podanej wersji i aktualizuje plik <code>package.json</code> , umieszczając informacje o pakiecie w sekcji <code>devDependencies</code> .
<code>npm install --global package@version</code>	To polecenie instaluje globalnie pakiet o podanej wersji.
<code>npm list</code>	To polecenie wyświetla listę wszystkich lokalnych pakietów i ich zależności.
<code>npm run</code>	To polecenie wykonuje jeden ze skryptów zdefiniowanych w pliku <code>package.json</code> .

Ostatnie polecenie w tabeli 10.7 ma nieco inny charakter, niemniej menedżery pakietów od dawna pozwalają na wykonywanie dodatkowych poleceń, które w przypadku NPM są deklarowane w sekcji scripts pliku *package.json*. W projektach Vue.js uzyskujemy w ten sposób dostęp do narzędzi używanych w czasie tworzenia aplikacji, a także podczas przygotowywania się do wdrożenia. Oto sekcja scripts pliku *package.json* w przykładowym projekcie:

```
...
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "lint": "vue-cli-service lint"
},
...

```

Powyższe polecenia podsumowano w tabeli 10.8. Przykłady ich użycia znajdziesz w kolejnych podrozdziałach.

Tabela 10.8. Polecenia zdefiniowane w sekcji scripts pliku *package.json*

Nazwa	Opis
serve	To polecenie uruchamia narzędzia deweloperskie (por. „Omówienie narzędzi deweloperskich”).
build	To polecenie wykonuje proces budowania aplikacji (por. „Budowanie aplikacji do wdrożenia”).
lint	To polecenie uruchamia linter języka JavaScript (por. „Zastosowanie lintera”).

Polecenia z tabeli 10.8 są wykonywane przez podanie komendy `npm run`, po której następuje nazwa polecenia. Musisz wykonać je w katalogu projektu, zawierającym plik *package.json*. Jeśli więc chcesz uruchomić polecenie `lint` w swoim projekcie, przejdź do katalogu *projecttools* i wykonaj polecenie `npm run list`.

Omówienie narzędzi deweloperskich

Narzędzia deweloperskie pozwalają na automatyczne wykrywanie zmian, komplikację aplikacji i generowanie pakietów gotowych do użycia w przeglądarce. Wszystkie te zadania możesz wykonywać samodzielnie, jednak dzięki automatycznym aktualizacjom możesz znacznie ułatwić sobie tworzenie aplikacji w Vue.js. Aby uruchomić narzędzia deweloperskie, otwórz wiersz poleceń, przejdź do katalogu *projecttools* i wykonaj polecenie z listingu 10.2.

Listing 10.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Najważniejszym pakietem wchodząącym w skład narzędzi deweloperskich jest *webpack*. Stanowi on podstawę wielu narzędzi deweloperskich i frameworków. *Webpack* tworzy paczki z modułami, dzięki czemu mogą one być używane w przeglądarce. Jest to dość ogólny opis tego niezwykle przydatnego narzędzia — niebawem sam się przekonasz, jak ważnym narzędziem w czasie tworzenia aplikacji Vue.js jest właśnie *webpack*.

Po wykonaniu polecenia z listingu 10.2 zobaczysz serię komunikatów w trakcie przygotowywania przez *webpack* paczki z plikami niezbędnej do uruchomienia aplikacji. *Webpack* zaczyna od pliku *main.js*, a następnie wczytuje wszystkie moduły, do których są odwołania, za pomocą instrukcji `import` lub `require`, aby utworzyć zbiór zależności. Proces ten jest powtarzany rekurencyjnie dla każdego z modułów wymaganych przez plik *main.js*, aż w końcu zostaje utworzone pełne drzewo zależności dla całej aplikacji. Drzewo to jest łączone w jeden duży plik, nazywany **paczką** (ang. *bundle*).

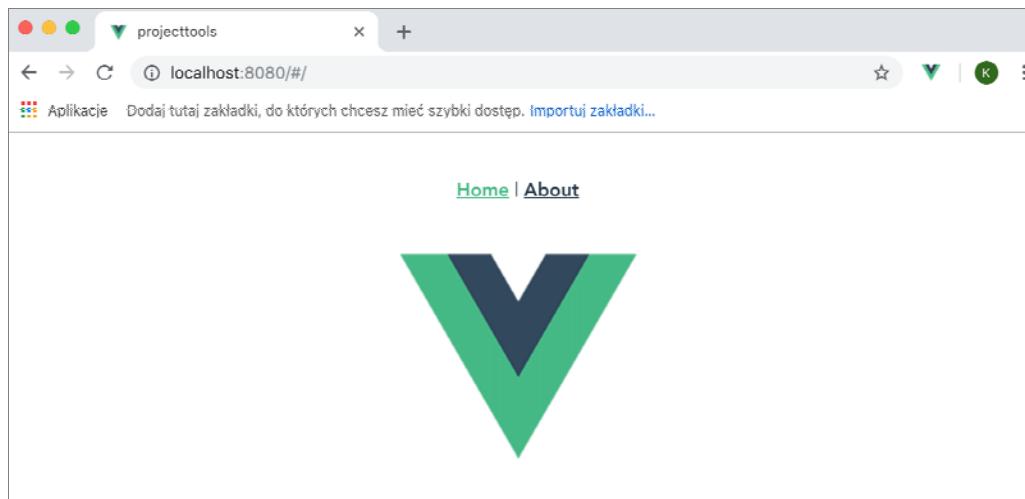
Podczas procesu budowania *webpack* informuje nas o postępcach:

```
...
10% building modules 4/7 modules 3 active ...\\node_modules\\webpack\\hot\\emitter.js
...
```

Proces pakowania może zająć dłuższą chwilę, ale jest on wykonywany tylko raz, przy starcie narzędzi deweloperskich. Po zakończeniu wstępnej inicjalizacji zobaczysz komunikat podobny do poniższego, który informuje nas o zakończeniu komplikacji i utworzeniu paczki:

```
...
DONE Compiled successfully in 2099ms
App running at:
- Local: http://localhost:8080/
- Network: http://192.168.0.77:8080/
Note that the development build is not optimized.
To create a production build, run npm run build.
...
```

Otwórz przeglądarkę i przejdź pod adres *http://localhost:8080*, aby obejrzeć przykładową aplikację (rysunek 10.9).



Rysunek 10.9. Zastosowanie narzędzi deweloperskich

- **Wskazówka** Z pewnością zauważysz, że adres URL wyświetlany w przeglądarce to *http://localhost:8080/#/*. Dodatkowe znaki występujące w adresie wynikają z użycia pakietu *vue-router*, dodanego do projektu na początku rozdziału (por. rozdziały 22. – 24.).

Omówienie procesów komplikacji i transformacji

Mimo że *webpack* koncentruje się na czystym języku JavaScript, jego funkcjonalność można rozszerzyć na inne rodzaje treści. Wystarczy użyć specjalnych rozszerzeń nazywanych **loaderami**. Narzędzia deweloperskie Vue.js zawierają wiele loaderów, ale warto zwrócić szczególnie uwagę na dwa z nich.

Nie musisz używać ich bezpośrednio, ponieważ loadery te są stosowane automatycznie, jednak warto wiedzieć, na czym polega ich praca.

Pierwszym ważnym loaderem jest vue-loader, który odpowiada za przekształcenie mieszanej treści, obecnej w plikach .vue, dzięki czemu można ją skompilować i zapakować przed przesłaniem do przeglądarki. Dzięki temu loaderowi możemy definiować komponenty w pojedynczych plikach, które łączą ze sobą kod HTML, JavaScript i CSS. Drugi ważny loader aktywuje kompilator Babel, odpowiedzialny za komplikację kodu JavaScript zawierającego najnowsze mechanizmy tego języka do postaci kompatybilnej ze starszymi przeglądarkami, które nie wspierają najnowszych mechanizmów. Aby pokazać działanie kompilatora Babel, do pliku *main.js* dodaję instrukcje, które korzystają z najnowszych możliwości języka JavaScript (listing 10.3).

Listing 10.3. Dodawanie instrukcji do pliku *src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
let first = "Witaj,";
const second = "świecie!";
console.log(`Komunikat ${first}, ${second}`);
new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app')
```

Powyższe instrukcje zawierają trzy odwołania do najnowszych możliwości języka JavaScript — słowo kluczowe `let`, słowo kluczowe `const` i łańcuch tekstowy zawierający szablon. Te funkcje języka nie są dostępne we wszystkich przeglądarkach, tak więc Babel musi je przekształcić.

- **Wskazówka** Po zapisaniu zmian w pliku *main.js* w wierszu poleceń zobaczysz ostrzeżenie. Jest ono generowane przez linter, który opisuję w podrozdziale „Zastosowanie lintera” — na razie możesz je zignorować.

Aby zobaczyć efekt z listingu 10.3, musimy się trochę namęczyć, ponieważ kod aplikacji jest pakowany przez narzędzie *webpack*. Przejdź na zakładkę *Sources* narzędzi deweloperskich przeglądarki. Powinieneś zobaczyć zawartość paczki z aplikacją, dostępną w sekcji *webpack-internal*.

- **Wskazówka** Nie martw się, jeśli nie możesz znaleźć skompilowanego kodu źródłowego. Nie wszystkie przeglądarki wspierają tę funkcję, a nawet jeśli tak jest, nie zawsze skorzystanie z niej jest możliwe. Najważniejsze, abyś zapamiętał, że kompilator transformuje kod języka JavaScript, aby zapewnić jego maksymalną kompatybilność ze starszymi przeglądarkami.

Znajdź plik *main.js* w oknie przeglądarki, a zobaczy, że instrukcje z listingu 10.3 zostały przekształcone w następujący sposób:

```
...
var first = "Witaj,";
var second = "świecie!";
console.log('Komunikat' + first + ',' + second);
...
```

Proces komplikacji spowodował zastąpienie słów kluczowych `let` i `const` słowem `var`, a także zamianę łańcucha szablonu na zwykłe złączenie tekstów. W efekcie otrzymaliśmy kod kompatybilny z wszystkimi przeglądarkami kompatybilnymi z Vue.js.

Ograniczenia kompilatora Babel

Babel to doskonałe narzędzie, ale zakres jego działania obejmuje jedynie mechanizmy języka JavaScript jako takiego. Babel nie doda do naszej aplikacji możliwości oferowanych przez nowoczesne API przeglądarek, jeśli dana przeglądarka tych API nie implementuje. Oczywiście nic nie stoi na przeszkodzie, aby stosować te API w swoich aplikacjach (np. API lokalnej pamięci, z którego korzystaliśmy w części I tej książki), ale siłą rzeczy ograniczasz w ten sposób listę przeglądarek kompatybilnych z Twoją aplikacją.

W listingu 10.4 oznaczam komentarzem instrukcje dodane w listingu 10.3, aby zapobiec wyświetaniu ostrzeżenia przez narzędzia deweloperskie.

Listing 10.4. Oznaczanie komentarzem instrukcji w pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'
Vue.config.productionTip = false
// let first = "Witaj,"
// const second = "świecie!";
// console.log('Message ${first}, ${second}');
new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app')
```

Omówienie serwera deweloperskiego HTTP

Aby uprościć proces tworzenia aplikacji, projekt zawiera pakiet `webpack-dev-server`, który uruchamia serwer HTTP zintegrowany z narzędziem `webpack`. Domyślny port tego serwera to 8080. Nasłuchiwanie na tym porcie zaczyna się zaraz po zakończeniu procesu pakowania.

Po otrzymaniu żądania HTTP serwer deweloperski zwraca zawartość pliku `public/index.html`. W trakcie przetwarzania pliku `index.html` serwer dodaje niezwykle ważną rzecz, którą zauważysz po kliknięciu prawym przyciskiem myszy okna przeglądarki i wybraniu opcji *Pokaż źródło strony*.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="/favicon.ico">
    <title>projecttools</title>
    <link as="script" href="/app.js" rel="preload">
  </head>
  <body>
    <noscript>
      <strong>
```

```

    We're sorry but projecttools doesn't work properly without
    JavaScript enabled. Please enable it to continue.

</strong>
</noscript>
<div id="app"></div>
<!-- built files will be auto injected --&gt;
&lt;script type="text/javascript" src="/app.js"&gt;&lt;/script&gt;&lt;/body&gt;
&lt;/html&gt;
</pre>

```

Serwer deweloperski dodaje element script, aby dołączyć plik *app.js*, generowany przez *webpack* w czasie inicjalizacji naszej aplikacji dla narzędzi deweloperskich. Paczka nie zawiera tylko kodu aplikacji — do przeglądarki są wysyłane również dodatkowe mechanizmy, które pozwalają na usprawnienie procesu deweloperskiego Vue.js, co opisuję w dalszych fragmentach.

Omówienie mechanizmu zamiany modułów na gorąco

Paczka tworzona przez *webpack* wspiera mechanizm *Hot Module Replacement* (**HMR** — zamiana modułów na gorąco). Dzięki niemu wprowadzenie zmiany w pliku z kodem źródłowym aplikacji lub w innych plikach projektu z treścią prowadzi do przetworzenia tego pliku i jego loaderów przez *webpack*. Zmieniony plik jest przesyłany do przeglądarki. W większości przypadków zamianie ulega tylko niewielki fragment całej aplikacji i co więcej, aplikacja jest odświeżana bez utraty jej aktualnego stanu. Przykład działania tego mechanizmu przedstawiono w listingu 10.5.

Listing 10.5. Zmiana treści w pliku src/App.vue

```

<template>
  <div id="app">
    <div>Liczba kliknięć: {{ counter }}</div>
    <button v-on:click="incrementCounter">Wciśnij mnie</button>
    <div id="nav">
      <router-link to="/">Strona główna</router-link> |
      <router-link to="/about">O mnie</router-link>
    </div>
    <router-view />
  </div>
</template>
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter() {
      this.counter++;
    }
  }
}
</script>
<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}

```

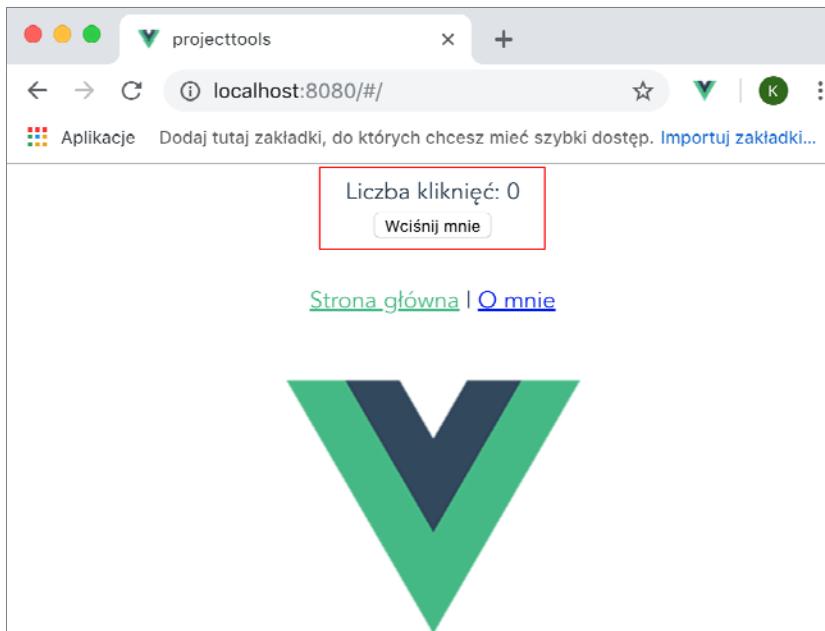
```

        }
        #nav {
            padding: 30px;
        }
        #nav a.router-link-exact-active {
            color: #42b983;
        }
    
```

</style>

Oryginalna przesłana do przeglądarki paczka zawiera kod, który otwiera trwałe połączenie HTTP z serwerem i czeka na instrukcje. Po zapisaniu zmian w pliku zobaczysz komunikaty w wierszu poleceń, w trakcie gdy plik komponentu będzie przekształcany, komplikowany i pakowany.

Trwałe połączenie HTTP pozwala poinformować przeglądarkę o powstaniu modułu do zamiany. Jest on wysyłany do przeglądarki, po czym następuje jego wdrożenie w aplikacji (rysunek 10.10).



Rysunek 10.10. Funkcja zamiany modułu na gorąco

W przypadku niektórych zmian funkcja HMR pozwala na aktualizację aplikacji bez czyszczenia jej stanu. Najlepiej działa to dla zmian wprowadzanych w kodzie HTML. W listingu 10.6 zmieniam element używany do wyświetlania komunikatu użytkownikowi.

Listing 10.6. Wprowadzanie zmiany w kodzie HTML w pliku src/App.vue

```

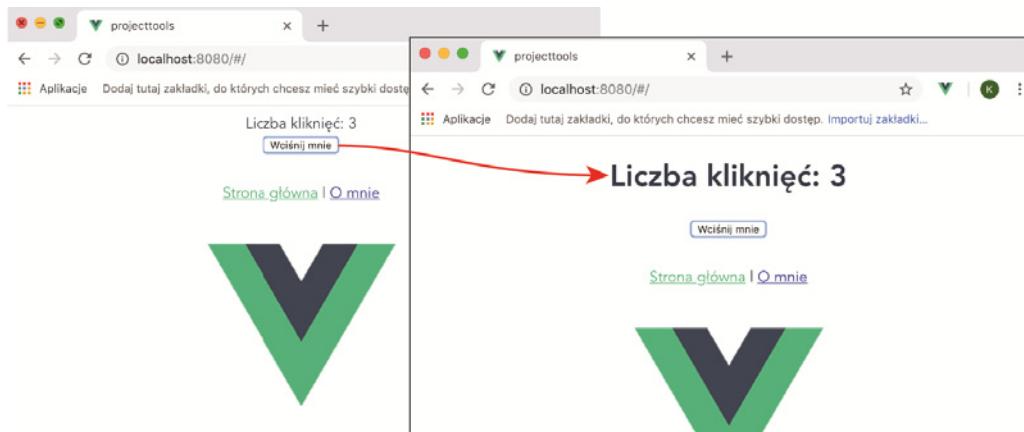
...
<template>
    <div id="app">
        <h1>Liczba kliknięć: {{ counter }}</h1>
        <button v-on:click="incrementCounter">Wciśnij mnie</button>
        <div id="nav">
            <router-link to="/">Strona główna</router-link> |
            <router-link to="/about">O mnie</router-link>
        </div>
        <router-view />
    </div>

```

```
</div>
</template>
...

```

Kliknij kilkakrotnie przycisk w przeglądarce, a następnie zapisz zmiany w pliku *App.vue*. Zobaczysz, że aplikacja została zaktualizowana bez utraty swojego stanu — wartość właściwości *counter* została zachowana (rysunek 10.11). Zachowanie stanu jest niezwykle użyteczne, gdy pracujesz nad funkcją niedostępną z głównego ekranu aplikacji, wymagającą przejścia przez kilka innych ekranów lub wysłania formularza. Unikasz wtedy żmudnego powtarzania tych samych operacji przy każdej zmianie wprowadzonej w pliku.



Rysunek 10.11. Aktualizacja z zachowaniem stanu

Niestety nie wszystkie zmiany można wprowadzić z zachowaniem stanu aplikacji. Jeśli zmienisz fragment elementu *script* komponentu, Vue.js będzie musiał usunąć istniejącą instancję komponentu i utworzyć nową. Stan innych komponentów pozostanie na szczęście niezmieniony. W nielicznych sytuacjach HMR działa po prostu źle i będziesz zmuszony odświeżyć całą aplikację, aby zobaczyć wprowadzone zmiany.

Omówienie wyświetlania błędów

Jedną z konsekwencji działania funkcji HMR jest ograniczenie czasu spędzonego na obserwowania komunikatów wyświetlanego w wierszu poleceń. Twoja uwaga będzie skupiona głównie na oknie przeglądarki. Takie zachowanie rodzi pewne ryzyko — treści w przeglądarce nie będą ulegały zmianie, jeżeli kod będzie zawierać błędy. Wynika to z faktu, że w trakcie komplikacji z powodu błędów nie dojdzie do powstania nowej wersji modułu, która mogłaby być przesłana za pomocą mechanizmu HMR.

Aby rozwiązać ten problem, paczka wygenerowana przez *webpack* zawiera wbudowany mechanizm wyświetlania błędów, który przedstawia problemy w oknie przeglądarki. Aby zademonstrować ten mechanizm, dodaj instrukcję w listingu 10.7 do elementu *script* w pliku *App.vue*.

Listing 10.7. Dodawanie nieprawidłowej instrukcji do pliku src/App.vue

```
...
<script>
export default {
  ta instrukcja nie jest prawidłowa!
  data() {
    return {
      counter: 0
    }
  }
}

```

```

        }
    },
    methods: {
        incrementCounter() {
            this.counter++;
        }
    }
}
</script>
...

```

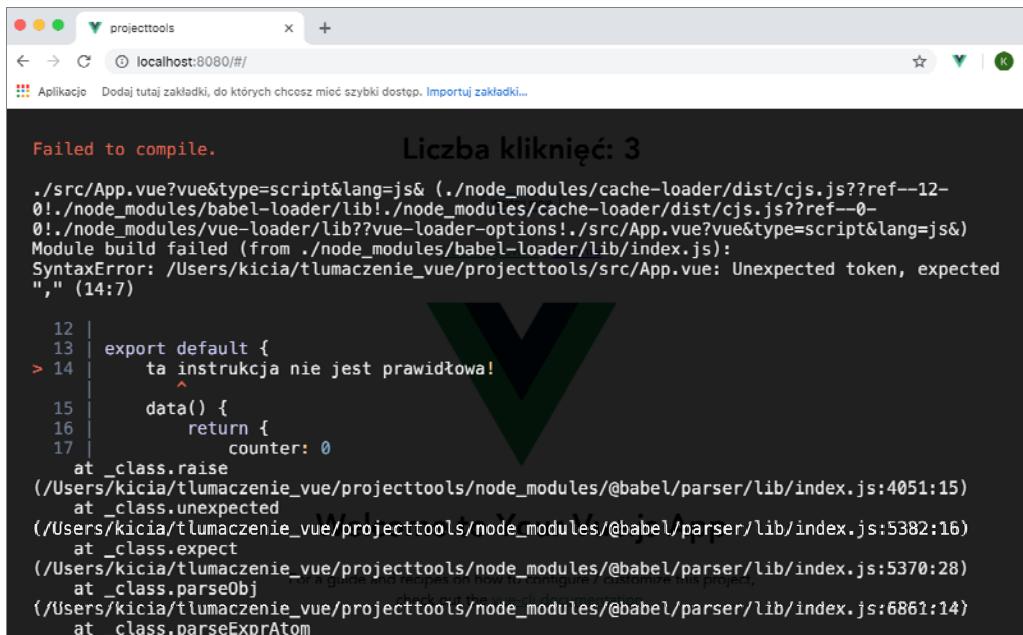
Dodana linia nie jest prawidłową instrukcją języka JavaScript. Po zapisaniu pliku w procesie budowania nastąpi próba skompilowania kodu, co zakończy się błędem widocznym w wierszu poleceń:

```

ERROR Failed to compile with 1 errors      00:48:25
error  in ./src/App.vue?vue&type=script&lang=js&
Syntax Error: SyntaxError: /Users/kicia/tłumaczenie_vue/projecttools/src/App.vue: Unexpected
↳ token, expected "," (14:7)
12 |
13 | export default {
> 14 |     ta instrukcja nie jest prawidłowa!
|         ^
15 |     data() {
16 |         return {
17 |             counter: 0

```

Ten sam komunikat o błędzie jest wyświetlany w oknie przeglądarki, co widać na rysunku 10.12. Nie ma więc szans, że ten błąd umknie Twojej uwadze, nawet jeśli nie śledzisz tego, co się dzieje w wierszu poleceń.



Rysunek 10.12. Wyświetlanie błędu w oknie przeglądarki

W listingu 10.8 oznaczyłem komentarzem instrukcję, która spowodowała błąd. Dzięki temu aplikacja może być zbudowana bez problemu.

Listing 10.8. Oznaczenie komentarzem problematycznej instrukcji w pliku src/App.vue

```
...
<script>
export default {
    // ta instrukcja nie jest prawidłowa!
    data() {
        return {
            counter: 0
        }
    },
    methods: {
        incrementCounter() {
            this.counter++;
        }
    }
}
</script>
...
```

Stosowanie lintera

Do funkcji, które wybrałem w przykładowym projekcie, zalicza się linter — pakiet ESLint — odpowiedzialny za analizę projektu pod kątem zgodności kodu ze zbiorem rozmaitych reguł. Konfiguracja, którą wybrałem dla lintera, oznacza, że weryfikacja będzie wykonywana przy zmianie dowolnego pliku w podkatalogu *src*.

Po dodaniu lintera w trakcie konfiguracji projektu Vue.js otrzymujesz możliwość wyboru szeregu opcji. Są one opisane w tabeli 10.9.

Tabela 10.9. Opcje reguł lintera w Vue.js

Opcja	Opis
<i>Error Prevention Only</i> (zapobiegaj jedynie błędom)	W tej opcji stosowane są jedynie najważniejsze reguły Vue.js i zalecane reguły ESLint opisane poniżej.
<i>Airbnb</i>	W tej opcji stosowany jest zbiór reguł opracowanych przez Airbnb, opisany na stronie https://github.com/airbnb/javascript . Te reguły są stosowane obok podstawowych reguł Vue.js opisanych poniżej.
<i>Standard</i> (standardowy)	W tej opcji stosowany jest zbiór reguł standardowych, opisanych na stronie https://github.com/standard/standard . Te reguły są stosowane obok podstawowych reguł Vue.js opisanych poniżej.
<i>Prettier</i>	Ta opcja korzysta z narzędzia Prettier, które wymusza formatowanie kodu. Narzędzie to opisane jest na stronie https://prettier.io/ . Formatowanie jest wykonywane obok podstawowych reguł opisanych poniżej.

W naszym projekcie wybieram opcję *Error Prevention Only*, która oznacza zastosowanie najmniejszego zbioru reguł i koncentruje się na znalezieniu problemów, które mogą doprowadzić do powstawania błędów. Pozostałe opcje wykraczają poza zapobieganie błędom — ich celem jest zapewnienie odpowiedniego stylu programowania. Osobiście nie przepadam za tymi wytycznymi, dlatego z nich nie korzystam — moim zdaniem standardowe reguły są niezwykle frustrujące, ponieważ dotyczą formatowania kodu — więcej informacji na ten temat poniżej.

Cienie i blaski stosowania lintera

Linter jest niezwykle przydatnym narzędziem, zwłaszcza w zespołach programistów składających się z osób o różnym doświadczeniu i umiejętnościach. Linter pozwala na wykrycie typowych problemów i niewielkich, acz bolesnych błędów, które mogą skutkować nieoczekiwany zachowaniem lub trudnościami w zarządzaniu aplikacją w dłuższej perspektywie. Jestem zwolennikiem stosowania lintera w takich sytuacjach i lubię przetwarzać mój kod za jego pomocą, zwłaszcza po zakończeniu pracy nad nową funkcją w aplikacji lub przed wysłaniem moich zmian do systemu kontroli wersji.

Z drugiej strony linter może stać się narzędziem wprowadzającym podziały i wywołującym walkę w zespole. Poza wykrywaniem błędów w kodzie linter może narzuścić reguły związane z wcięciami w kodzie, umiejscowieniem nawiasów, użyciem średników i spacji, a także wieloma innymi kwestiami. Większość programistów ma swoje preferencje dotyczące stylu i jest przekonana, że każdy inny programista powinien przestrzegać ich reguł. Z pewnością mam tak i ja: jeden poziom wcięcia to dla mnie cztery spacje, a nawias otwierający powinien znaleźć się w tym samym wierszu co wyrażenie, do którego się odnosi. Wiem, że moje przekonania stanowią moją „jedyną prawdziwą drogę” wiodącą do pisania dobrego kodu. Dlatego też fakt, że inni programiści lubią stosować dwie spacje jako wcięcie jest dla mnie źródłem cichej, acz nieustającej radości, odkąd tylko zacząłem programować.

Lintery pozwalają stanowczym osobom wymusić na innych respektowanie wytycznych dotyczących formatowania. Osoby te uzasadniają to tym, że programiści spędzają zbyt dużo czasu na kłótniach o styl kodowania i lepiej byłoby, gdyby wszyscy pisali tak samo. Moim zdaniem programiści i tak znajdą zastępczy temat do kłótni i próba wymuszenia stylu kodowania to tylko sposób narzucenia zdania jednej osoby całemu zespołowi programistycznemu.

Często pomagam czytelnikom, u których przykłady nie działają (mój adres e-mail to adam@adam-freeman.com, gdybyś kiedykolwiek potrzebował pomocy), przez co mam do czynienia z różnymi stylami programowania praktycznie każdego tygodnia. W głębi serca wiem, że każdy, kto nie podąża za moim stylem kodowania, po prostu nie ma racji. Nie staram się jednak tego na nikim wymusić — zamiast tego mój edytor reformatuje kod, co w większości przyzwoitych edytorów nie stanowi problemu.

Moim zdaniem linter powinien być stosowany oszczędnie, a jego działanie należy ograniczyć do wykrywania faktycznych błędów. Decyzje dotyczące formatowania pozostaw w rękach poszczególnych programistów. Korzystaj z edytora kodu, aby reformatować istniejący kod innego programisty.

Gdy wybierzesz opcję *Error Prevention Only*, wobec Twojego kodu zostaną zastosowane reguły zaliczające się do jednej z dwóch grup. Pierwsza grupa reguł została opracowana przez programistów ESLint, a druga — przez innych programistów Vue.js.

Reguły ESLint są opisane na stronie <https://eslint.org/docs/rules> i koncentrują się na ogólnych błędach, które można napotkać w pracy z językiem JavaScript. Reguły Vue.js, jak możesz się domyślić, skupiają się na tworzeniu aplikacji w Vue.js. Istnieją trzy zbiory reguł, opisane w tabeli 10.10. Szczegółowe opisy każdej z grup znajdziesz na stronie <https://github.com/vuejs/eslint-plugin-vue>.

Tabela 10.10. Poziomy reguł lintera Vue.js

Nazwa	Opis
<i>Essential</i> (niezbędne)	Te reguły są związane głównie z prawidłowym stosowaniem dyrektyw. Dyrektywy opisano w rozdziałach 12. – 15.
<i>Strongly Recommended</i> (wysoce zalecane)	Te reguły są związane z poprawą czytelności kodu. Mają związek głównie z formatowaniem i układem kodu.
<i>Recommended</i> (zalecane)	Te reguły są związane z zapewnieniem spójności.

Tylko reguły typu *Essential* są włączone domyślnie. Ten stan rzeczy możesz zmienić, konfigurując sekcję `eslintConfig` w pliku `package.json`. Oto fragment tego pliku według stanu z początku rozdziału:

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/essential",
    "eslint:recommended"
  ],
  "rules": {},
  "parserOptions": {
    "parser": "babel-eslint"
  }
},
...
```

Sekcja `extends` pliku konfiguracyjnego jest używana do określenia zastosowanego przez nas zbioru reguł. Trzy wartości stosowane wobec zbiorów reguł charakterystycznych dla Vue.js to: `plugin:vue/recommended`, `plugin:vue/strongly-recommended` i `plugin:vue/essential`, analogicznie do wartości opisanych w tabeli 10.10. W listingu 10.9 włączyłem regułę *Strongly Recommended*. Gdy włączasz dany zbiór reguł, reguły o wyższym priorytecie zostają włączone automatycznie. Jeśli więc wybierzesz zbiór *Recommended*, pozostałe dwa zbiory również zostaną włączone.

Listing 10.9. Zmiana reguł lintera w pliku `projecttools/package.json`

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/strongly-recommended",
    "eslint:recommended"
  ],
  "rules": {},
  "parserOptions": {
    "parser": "babel-eslint"
  }
},
...
```

- **Wskazówka** Jeśli zaznaczyłeś opcję przechowywania ustawień funkcji w odrębnych plikach konfiguracyjnych w momencie tworzenia projektu, powyższe zmiany możesz wprowadzić w pliku `.eslintrc.json` zamiast `package.json`.

Zmiany w konfiguracji lintera nie zostaną wdrożone, dopóki nie zrestartujesz narzędzi deweloperskich. Wciśnij kombinację klawiszy `Ctrl+C`, aby zatrzymać działanie narzędzi, a następnie uruchom je ponownie, wpisując polecenie z listingu 10.10 w katalogu `projecttools`

Listing 10.10. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Linter zostanie uruchomiony w ramach procesu budowania. Niektóre reguły włączone w listingu 10.9 odnoszą się do liczby spacji stanowiących pojedynczy poziom wcięcia w kodzie. Jeśli zastosowałeś te same konwencje w kodzie, zobaczysz ostrzeżenie takie jak poniżej:

```
...
Module Warning (from ./node_modules/eslint-loader/index.js):
warning: Expected indentation of 2 spaces but found 4 spaces (vue/html-indent) at
↳src/App.vue:2:1:
1 | <template>
> 2 |   <div id="app">
|   ^
3 |     <h1>Liczba kliknięć: {{ counter }}</h1>
4 |     <button v-on:click="incrementCounter">Wciśnij mnie</button>
5 |   <div id="nav">
...

```

Być może jednak nie zobaczysz tego ostrzeżenia — to zależy od Twojego stylu programowania — z pewnością za to ujrzesz to przedstawione poniżej, które odnosi się do zmian wprowadzonych w pliku *App.vue* w listingu 10.5:

```
warning: Expected '@' instead of 'v-on:' (vue/v-on-style) at src/App.vue:4:21:
2 |   <div id="app">
3 |     <h1>Liczba kliknięć: {{ counter }}</h1>
> 4 |     <button v-on:click="incrementCounter">Wciśnij mnie</button>
|     ^
5 |     <div id="nav">
6 |       <router-link to="/">Strona główna</router-link> |
7 |       <router-link to="/about">O mnie</router-link>
```

Jak objaśniam w rozdziale 14., istnieją dwie metody podłączenia się pod zdarzenia — krótsza i dłuższa. Jedna z reguł z grupy *Strongly Recommended* wymusza stosowanie krótkiej formy, zgłaszając ostrzeżenie, jeśli zostanie wykryta dłuższa forma.

■ **Wskazówka** Jeśli chcesz skorzystać z lintera bez uruchamiania procesu budowania, możesz użyć polecenia `npm run lint`.

Dostosowywanie reguł lintera

Żadna z reguł, które wygenerowały dotychczas ostrzeżenia, nie jest związana z moimi osobistymi preferencjami programowania. Jestem oddany idei czterech wcięć, a ponadto preferuję długą formę stosowania funkcji Vue.js w elementach HTML. Nie chciałbym jednak wyłączać całego zbioru reguł tylko z tej przyczyny. Zamiast tego mogę zmienić bądź wyłączyć pojedyncze reguły, zachowując te, z którymi nie mam problemu. W listingu 10.11 wyłączam regułę badania wcięć w kodzie i zmieniam regułę dotyczącą dyrektyw, dzięki czemu akceptuje ona wyłącznie długie, a nie krótkie postaci.

Listing 10.11. Konfiguracja reguł w pliku *projecttools/package.json*

```
...
"eslintConfig": {
  "root": true,
  "env": {
```

```

    "node": true
},
"extends": [
    "plugin:vue/strongly-recommended",
    "eslint:recommended"
],
"rules": {
    "vue/html-indent": "off",
    "vue/v-on-style": [ "warn", "longform" ]
},
"parserOptions": {
    "parser": "babel-eslint"
}
},
...

```

Każda reguła ma swoje ustawienia, które można znaleźć na stronie podsumowania reguł grup, dla których załączylem adresy URL w poprzedniej sekcji. W tym przypadku ustawiam dwie reguły charakterystyczne dla Vue.js. Ustawiam wartość reguły vue/html-indent na off, dzięki czemu wyłączam badanie wcięć dla kodu HTML zawartego w elemencie template komponentu. Reguła vue/v-on-style pozostaje włączona, ale zmieniam jej ustawienia — akceptuję tylko długą postać i wyświetlam ostrzeżenie w przypadku użycia krótszej formy. Aby przekonać się o wprowadzonych zmianach, zrestartuj narzędzia deweloperskie.

Wyłączenie lintera dla pojedynczych instrukcji i plików

Możesz zauważyc, że błędy lintera wynikają z pojedynczych instrukcji, których nie jesteś w stanie zmienić. Zamiast wyłączać regułę całkowicie, możesz dodać komentarz w kodzie, który nakaże linterowi zignorować daną regułę dla następnego wiersza, np.:

```

...
<!-- eslint-disable-next-line vue/v-on-style -->
...

```

Jeśli chcesz wyłączyć wszystkie reguły dla kolejnej instrukcji, możesz skorzystać z następującej instrukcji:

```

...
<!-- eslint-disable-next-line  -->
...

```

Jeśli chcesz wyłączyć daną regułę w całym pliku, dodaj poniższy komentarz na początku pliku w przypadku reguły Vue.js lub na początku elementu script dla reguły ESLint:

```

...
<!-- eslint-disable vue/v-on-style -->
...

```

Jeśli chcesz wyłączyć linter dla wszystkich reguł w całym pliku, załącz poniższy komentarz:

```

...
<!-- eslint-disable  -->
...

```

Te komentarze pozwolą Ci zignorować kod, który nie jest zgodny z regułami, ale nie może zostać zmieniony bez konieczności wyłączenia reguł w całym projekcie. Możesz także wyłączać z procesu lintowania całe katalogi i typy plików dzięki odpowiednim deklaracjom w pliku *package.json*.

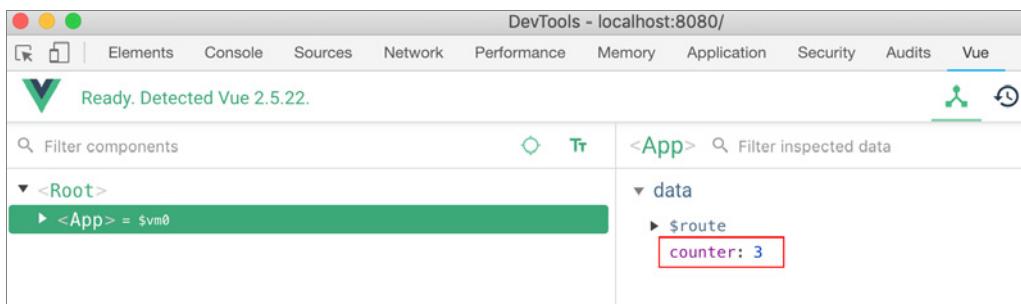
Debugowanie aplikacji

Nie wszystkie problemy zostaną wykryte przez kompilator czy też linter. Istnieje możliwość, że poprawnie komplilujący się kod zacznie sprawiać problemy w czasie wykonywania aplikacji. Istnieją dwie metody analizy zachowania aplikacji; obie opisano poniżej.

Analiza stanu aplikacji

Rozszerzenie Vue Devtools przeglądarki stanowi doskonale narzędzie do analizy stanu aplikacji Vue.js. Rozszerzenie to jest dostępne dla przeglądarek Google Chrome i Mozilla Firefox, a szczegóły znajdziesz pod adresem <https://github.com/vuejs/vue-devtools>. Po zainstalowaniu rozszerzenia w widoku narzędzi deweloperskich przeglądarki znajdziesz dodatkową zakładkę (narzędzia te są dostępne po wcisnięciu klawisza F12 — stąd czasem nazywa się je narzędziami F12).

Zakładka *Vue* pozwala na analizę i zmianę struktury i stanu aplikacji. Dzięki niej możesz przejrzeć komponenty, które tworzą aplikację, a także wszystkie dane w nich zawarte. W przykładowej aplikacji otwarcie zakładki *Vue* spowoduje wyświetlenie danych komponentu *App* w panelu po prawej stronie, w tym właściwości *counter* zdefiniowanej w listingu 10.5 (rysunek 10.13).



Rysunek 10.13. Analiza stanu aplikacji

Jeśli chcesz zmienić wartość danej właściwości, po prostu ją kliknij. Zmiana wartości w tym miejscu spowoduje odświeżenie modelu danych na żywo, co pociągnie za sobą odświeżenie zawartości w przeglądarce, oczywiście, o ile wyświetlasz dane za pomocą wiązania danych.

Rozszerzenie przeglądarki wyświetla informacje dotyczące magazynu Vuex (opisuję go w rozdziale 20.), szczegóły na temat zdarzeń wyzwalanych przez aplikację (rozdział 14.), a także szczegóły systemu trasowania adresów URL (rozdziały 22. – 24.).

Omówienie debuggera w przeglądarce

Nowoczesne przeglądarki zawierają w sobie wyszukane debuggery, które pozwalają kontrolować działanie aplikacji i badanie jej stanu. Narzędzia deweloperskie Vue.js wspierają tworzenie map źródeł (ang. *source maps*), dzięki którym przeglądarka jest w stanie powiązać zminifikowany i zapakowany kod produkcyjny z przyjaznym i łatwym w debugowaniu kodem przeznaczonym dla programisty.

Niektóre przeglądarki pozwalają przeglądać kod źródłowy aplikacji dzięki mapom źródeł. Możliwe jest także tworzenie punktów wstrzymania (ang. *breakpoints*). Aplikacja wstrzymuje swoje działanie w momencie napotkania takiego punktu i przekazuje kontrolę do debugera. W momencie pisania tej książki tworzenie punktów wstrzymania jest dość niestabilną funkcją, która nie działa w przeglądarce Google Chrome i sprawia pewne problemy w innych przeglądarkach. W związku z tym najbezpieczniejszą opcję dotyczącą przekazania kontroli do debugera jest skorzystanie ze słowa kluczowego *debugger* (listing 10.12).

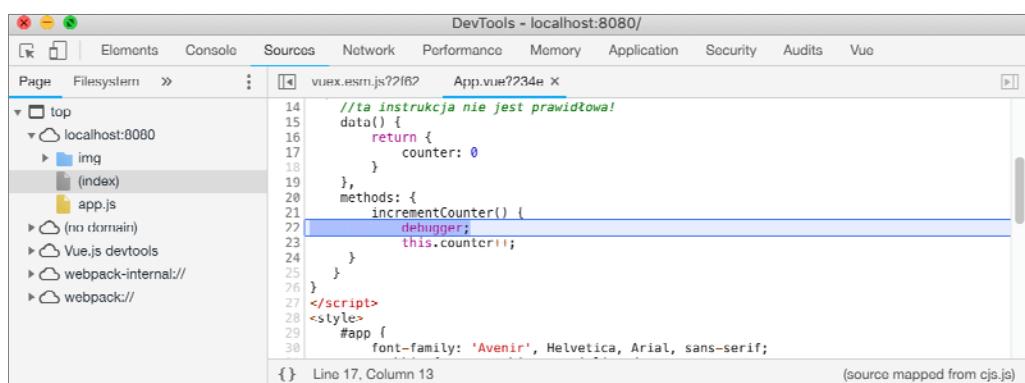
Listing 10.12. Wyzwalanie debugera w pliku *src/App.vue*

```

<template>
  <div id="app">
    <h1>Liczba kliknięć: {{ counter }}</h1>
    <button v-on:click="incrementCounter">Wciśnij mnie</button>
    <div id="nav">
      <router-link to="/">Strona główna</router-link> | 
      <router-link to="/about">O mnie</router-link>
    </div>
    <router-view />
  </div>
</template>
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter() {
      debugger;
      this.counter++;
    }
  }
}
</script>
<style>
  /* ...style zostały pominięte... */
</style>

```

Aplikacja zostanie uruchomiona jak zwykle, ale w momencie kliknięcia przycisku, a tym samym — wywołania metody `incrementCounter` — przeglądarka napotka słowo kluczowe `debugger` i wstrzyma swoje działanie. W tej sytuacji możesz skorzystać z okna narzędzi F12, aby przejrzeć wartości zmiennych w momencie wstrzymania działania skryptu. Możesz też ręcznie sterować działaniem skryptu (rysunek 10.14). Przeglądarka wykonuje zminifikowany i zapakowany kod utworzony przez narzędzia deweloperskie, ale dzięki mapie źródeł w debuggerze widzisz kod w czytelnej postaci.



Rysunek 10.14. Debugger w przeglądarce w praktyce

Większość przeglądarek zignoruje słowo kluczowe debugger, jeśli narzędzia F12 nie będą otwarte. Z pewnością dobrze jest jednak usunąć to słowo po zakończeniu sesji debugowania. Domyślne ustawienia lintera pozwolą Ci wykryć obecność tego słowa, jeśli budujesz aplikację w wersji produkcyjnej, czym zajmiemy się już za chwilę.

Konfiguracja narzędzi deweloperskich

Domyślna konfiguracja zawiera szereg przydatnych ustawień, jednak — przedżej czy później — pojawi się konieczność ich zmiany. W tym celu należy utworzyć plik *vue.config.js* w katalogu głównym projektu lub dodać sekcję vue w pliku *package.json*. W ramach przykładu wprowadzam zmianę w pliku *vue.config.js* w katalogu *projecttools* (listing 10.13).

Listing 10.13. Zawartość pliku *projecttools/vue.config.js*

```
module.exports = {
  runtimeCompiler: true
}
```

Tę samą zmianę wprowadziłem w rozdziale 9., dzięki czemu komponenty mogą definiować swoje szablony w formiełańcuchów znaków zamiast stosowania elementu *template*. Kompletny zbiór opcji konfiguracyjnych jest opisany na stronie <https://cli.vuejs.org/config/#vue-config-js>, jednak w tabeli 10.11 znajdziesz zestawienie opcji przydatnych w większości projektów.

Tabela 10.11. Przydatne opcje konfiguracyjne

Nazwa	Opis
baseUrl	Ta opcja jest stosowana do określenia prefiku adresu URL używanego do przejścia do aplikacji, jeśli jest ona wdrażana jako element większej witryny. Domyślnie adres ma wartość /.
outputDir	Ta opcja jest używana do określenia katalogu do zbudowania produkcyjnej wersji aplikacji (por. z podrozdziałem „Budowanie aplikacji do wdrożenia”). Domyślnie przyjmuje wartość <i>dist</i> .
devServer	Ta opcja jest używana do skonfigurowania serwera deweloperskiego HTTP za pomocą opcji opisanych pod adresem https://webpack.js.org/configuration/dev-server .
runtimeCompiler	Ta opcja włącza kompilator czasu wykonania, który pozwala na definiowanie szablonów komponentów za pomocą właściwości <i>template</i> , zwiększać tym samym ilość kodu przesłanego do przeglądarki.
chainWebpack	Ta opcja jest używana do konfiguracji narzędzia <i>webpack</i> . Wykorzystuję ją w rozdziale 21., aby wyłączyć narzędzie.

Budowanie aplikacji do wdrożenia

Pliki utworzone przez narzędzia deweloperskie nie są dostosowane do użycia produkcyjnego. Zawierają one dodatkowe funkcje, takie jak wsparcie zamiany modułów na gorąco (HRM). Gdy zdecydujesz się na wdrożenie swojej aplikacji, musisz utworzyć zbiór paczek zawierających tylko kod aplikacji i niezbędne dodatkowe treści, wraz z koniecznymi zależnościami. Tylko te pliki zostaną wdrożone na produkcyjnym serwerze HTTP.

W zależności od wybranych funkcji może zainstnieć konieczność wprowadzenia zmian w konfiguracji, aby przygotować aplikację do wdrożenia (co można było zaobserwować w rozdziale 8. na przykładzie aplikacji *Sklep sportowy*).

W przypadku projektu z niniejszego rozdziału dobrze jest usunąć słowo kluczowe `debugger`. W listingu 10.14 słowo to zostało oznaczone komentarzem w pliku `App.vue`.

Listing 10.14. Oznaczenie komentarzem słowa kluczowego `debugger` w pliku `src/App.vue`

```
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter() {
      //debugger;
      this.counter++;
    }
  }
}
</script>
```

Reguły lintera włączone w listingu 10.9 sprawdzają dodatkowe reguły, które obowiązują jedynie w przypadku przygotowania produkcyjnej wersji kodu. Choć ostrzeżenia te mogą wydać Ci się przydatne, wyłączam je, powracając w listingu 10.15 tylko do niezbędnych reguł lintera Vue.js.

Listing 10.15. Zmiana reguł lintera w pliku `projecttools/package.json`

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/essential",
    "eslint:recommended"
  ],
  "rules": {
    "vue/html-indent": "off",
    "vue/v-on-style": [ "warn", "longform" ]
  },
  "parserOptions": {
    "parser": "babel-eslint"
  }
},
...
}
```

Aby utworzyć wersję produkcyjną, uruchom polecamie z listingu 10.16 w katalogu `projecttools`.

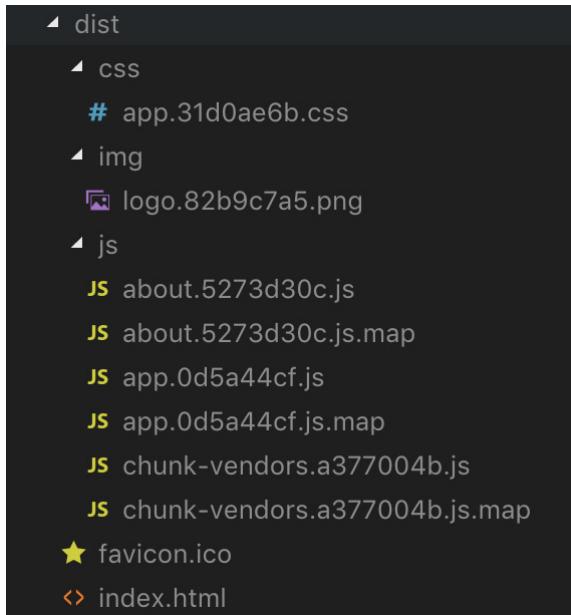
Listing 10.16. Tworzenie wersji produkcyjnej

```
npm run build --modern
```

Argument `--modern` to nieobowiązkowa opcja, która tworzy dwie wersje aplikacji — jedną przeznaczoną tylko do użycia w nowoczesnych przeglądarkach, które wspierają najnowsze możliwości języka JavaScript, i drugą kompatybilną ze starszymi przeglądarkami, w których konieczne jest zastosowanie dodatkowego kodu i bibliotek wspierających. Gdy korzystasz z tej opcji, nie musisz podejmować żadnych dodatkowych

działania — wybór odpowiedniej wersji jest wykonywany automatycznie (więcej informacji znajdziesz pod adresem <https://cli.vuejs.org/guide/browser-compatibility.html#modern-mode>).

Proces budowania może zająć dłuższą chwilę, zwłaszcza w przypadku dużych aplikacji. W efekcie otrzymamy katalog *dist*, który zawiera wszystkie pliki niezbędne do wdrożenia aplikacji (rysunek 10.15). Nazwy plików mogą różnić się od tych, które znajdziesz w swoim projekcie.



Rysunek 10.15. Zawartość katalogu *dist*

Katalog *dist* zawiera plik *index.html*, będący punktem startowym aplikacji. W tym pliku znajdziesz elementy *script* i *link*, dzięki którym zostaną wczytane pliki CSS i JavaScript:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset=utf-8>
  <meta http-equiv=X-UA-Compatible content="IE=edge">
  <meta name=viewport content="width=device-width,initial-scale=1">
  <link rel=icon href=/favicon.ico>
  <title>projecttools</title>
  <link as= style href=/css/app.b938236b.css rel=preload>
  <link as=script href=/js/app.b671844a.js rel=preload>
  <link as=script href=/js/chunk-vendors.02da9ab1.js rel=preload>
  <link href=/css/app.b938236b.css rel=stylesheet>
</head>
<body>
  <noscript><strong>We're sorry but projecttools doesn't work properly without JavaScript enabled. Please enable it to continue.</strong></noscript><div id=app></div>
  <script src=/js/chunk-vendors.02da9ab1.js></script>
  <script src=/js/app.b671844a.js></script>
</body>
</html>
  
```

Pliki języka JavaScript zawierają kod aplikacji, a także niezbędne do działania moduły Vue.js. Dzięki temu plik HTML nie wymaga wprowadzenia żadnych dodatkowych zmian.

Instalacja i zastosowanie serwera HTTP

W rzeczywistym świecie pliki z katalogu *dist* zostałyby skopiowane do produkcyjnego serwera HTTP, który może działać np. w dedykowanym lub współdzielonym hostingu albo w lokalnym centrum danych. Aby zwieńczyć niniejszy rozdział, zamierzam zainstalować prosty, lecz funkcjonalny serwer HTTP napisany w języku JavaScript, który to serwer będzie można pobrać i zainstalować za pomocą narzędzia NPM.

-
- **Ostrzeżenie** Serwer HTTP wykorzystany w trakcie tworzenia aplikacji nie jest dostosowany do użycia produkcyjnego. Musisz skorzystać z innego serwera WWW, aby dostarczyć swoją aplikację do użytkowników.
-

Wykonaj polecenia z listingu 10.17, aby zainstalować pakiet *Express* i dodatkowy pakiet, który pozwoli nam obsługiwać trasowanie adresów URL.

Listing 10.17. Dodawanie pakietów do produkcyjnej wersji aplikacji

```
npm install --save-dev express@4.16.3
npm install --save-dev connect-history-api-fallback@1.5.0
```

Następnie do katalogu *projecttools* dodajmy plik *serwer.js* o treści przedstawionej w listingu 10.18. W ten sposób konfiguruje pakiety zainstalowane w listingu 10.17, dzięki czemu obsługą one naszą przykładową aplikację.

Listing 10.18. Zawartość pliku *projecttools/server.js*

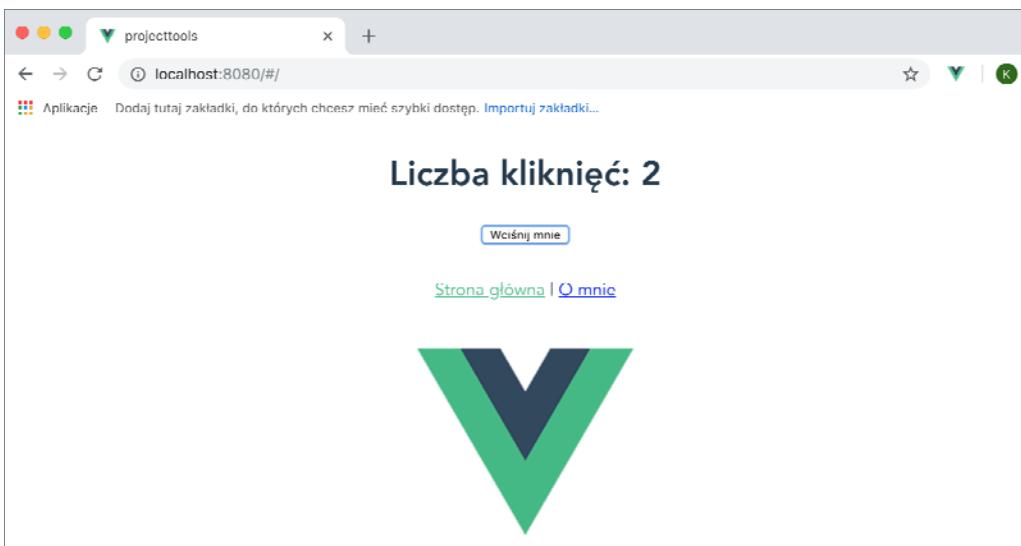
```
const express = require("express");
const history = require("connect-history-api-fallback");
const bodyParser = require('body-parser');
const app = express();
app.use(bodyParser.json());
app.use(history());
app.use("/", express.static("./dist"));
app.listen(80, function () {
    console.log("Serwer HTTP jest uruchomiony na porcie 80.");
});
```

Uruchom polecenie przedstawione w listingu 10.19 w katalogu *projecttools*, aby zweryfikować działanie aplikacji.

Listing 10.19. Uruchamianie wdrożonej aplikacji

```
node server.js
```

To polecenie wykonuje instrukcje z pliku JavaScript przedstawionego w listingu 10.18. W ten sposób uruchamiamy serwer WWW na porcie 80 i nasłuchujemy żądań HTTP. Przejdz na stronę <http://localhost> i przekonaj się, że aplikacja działa (rysunek 10.16).



Rysunek 10.16. Uruchamianie wdrożonej aplikacji

Podsumowanie

W tym rozdziale pokazałem, jak utworzyć projekt Vue.js za pomocą pakietu `@vue/cli`, i objaśniłem działanie narzędzi deweloperskich przezeń dostarczanych. Przedstawiłem strukturę projektu Vue.js, a także sposób komplikowania i pakowania aplikacji. Omówiłem proces zgłaszania błędów zarówno z poziomu kompilatora, jak i opcjonalnego lintera. Na zakończenie przedstawiłem narzędzia, z których można skorzystać w celu zdebugowania aplikacji. Wyjaśniłem także proces przygotowania projektu do wdrożenia. W kolejnym rozdziale opiszę obsługę Vue.js w zakresie wiązań danych.

ROZDZIAŁ 11.



Omówienie wiązań danych

W tym rozdziale omówię jedno z najważniejszych zagadnień dotyczących tworzenia aplikacji webowej: zastosowanie wiązania danych do wyświetlania wartości danych. Pokażę, jak dodać podstawowe wiązanie danych, czyli **interpolację tekstu** (ang. *text interpolation binding*), do szablonu komponentu, a także jak utworzyć wartość danych dla wiązania. Pokażę, jak ewaluować wyrażenia wiązań danych, a także przedstawię różne sposoby generowania i formatowania wartości wyświetlanych użytkownikowi. Tabela 11.1 umiejscowia interpolację tekstu w szerszym kontekście.

Tabela 11.1. Umiejscowienie interpolacji tekstu w szerszym kontekście

Pytanie	Odpowiedź
Czym są wiązania danych?	Wiązania danych łączą dane komponentu z elementami języka HTML w szablonie. Interpolacja tekstu stanowi przykład jednego z najprostszych wiązań, które obsługuje Vue.js.
Dlaczego są użyteczne?	Wiązania danych pozwalają na interaktywność w aplikacjach Vue.js. Akcje wykonywane przez użytkownika zmieniają stan aplikacji, co z kolei umożliwia odświeżenie elementów interfejsu użytkownika dzięki wiązaniom danych.
Jak się z nich korzysta?	Wiązania interpolacji tekstu są tworzone za pomocą podwójnych nawiasów klamrowych <code>{} i {}</code> . Z tej przyczyny często nazywa się je wiązaniem wąsatym (ang. <i>mustache binding</i>).
Czy są jakieś pułapki lub ograniczenia?	Vue.js pozwala na wykonywanie złożonych wyrażeń w wiązaniach danych. Jest to niezwykle użyteczna funkcja, jednak łatwo jest się nią zatrzymać, umieszczając zbyt dużo logiki w wiązaniu danych. W takiej sytuacji trudno jest testować aplikację i nią zarządzać.
Czy są jakieś rozwiązania alternatywne?	Wiązanie interpolacji tekstu to tylko jeden z przykładów wiązań danych obsługiwanych przez Vue.js — pozostałe wiązania opisuję w kolejnych rozdziałach.

Tabela 11.2 podsumowuje ten rozdział.

Tabela 11.2. Podsumowanie rozdziału

Problem	Rozwiązanie	Listing
Utwórz nowy komponent.	Utwórz plik z rozszerzeniem <i>.vue</i> i dodaj do niego elementy template, script i style.	11.6 – 11.7
Wyświetl wartość danych.	Dodaj wartość danych i wiązanie interpolacji tekstu.	11.8 – 11.9
Oblicz wartość.	Skorzystaj z metody lub właściwości obliczanej.	11.10 – 11.14
Sformatuj właściwość danych.	Skorzystaj z minimum jednego filtru.	11.15 – 11.18

Przygotowania do tego rozdziału

Aby wykonać przykłady przeznaczone do tego rozdziału, wykonaj polecenia z listingu 11.1 w wybranym przez siebie miejscu. W ten sposób utworzysz nowy projekt Vue.js.

Listing 11.1. Tworzenie nowego projektu

```
vue create templatesanddata --default
```

To polecenie utworzy projekt o nazwie *templatesanddata*. Po zakończeniu procesu inicjalizacji wykonaj polecenie z listingu 11.2 w katalogu *templatesanddata*, aby dodać pakiet Bootstrap do projektu.

Listing 11.2. Dodawanie pakietu Bootstrap CSS

```
npm install bootstrap@4.0.0
```

■ **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Dodaj instrukcje z listingu 11.3 do pliku *src/main.js*, aby włączyć obsługę Bootstrapa w aplikacji.

Listing 11.3. Dołączanie pakietu Bootstrap do pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
  render: h => h(App)
}).$mount('#app')
```

Dodaj instrukcje z listingu 11.4, aby wyłączyć regułę lintera, która generuje ostrzeżenie w momencie użycia konsoli JavaScript w przeglądarce, ponieważ będziesz z niej korzystać w dalszej części rozdziału.

Listing 11.4. Konfiguracja lintera w pliku templatesanddata/package.json

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
```

```

    "plugin:vue/essential",
    "eslint:recommended"
],
"rules": {
  "no-console": "off"
},
"parserOptions": {
  "parser": "babel-eslint"
}
},
"postcss": {
  "plugins": {
    "autoprefixer": {}
  }
},
...

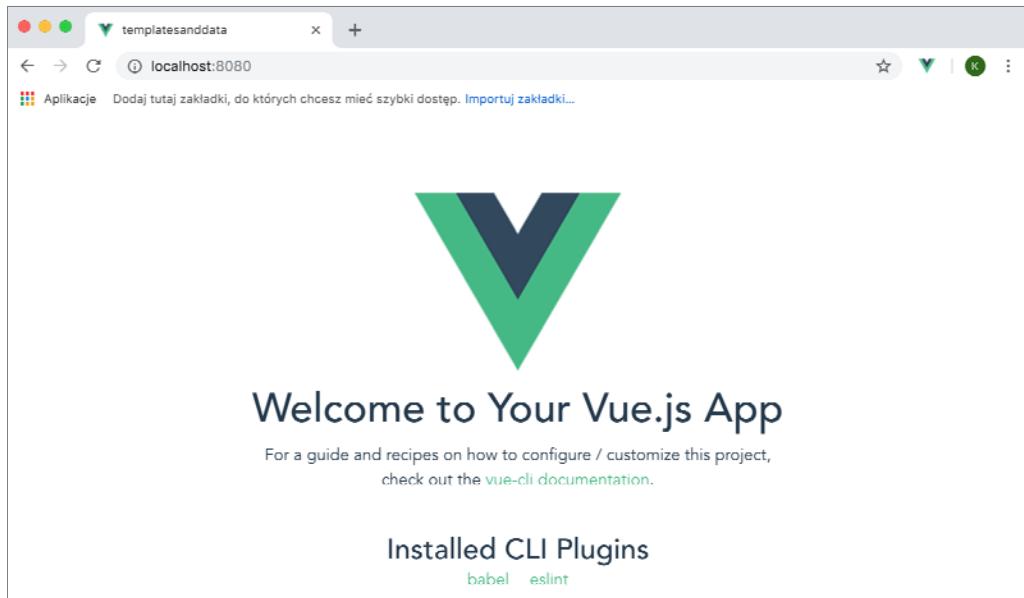
```

Wykonaj polecenie z listingu 11.5 w katalogu *templatesanddata*, aby uruchomić narzędzia deweloperskie.

Listing 11.5. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Zostanie wykonany początkowy proces budowania aplikacji, po którym zobaczysz komunikat o skutecznym zakończeniu komplikacji. Serwer HTTP będzie nasłuchiwał żądań na porcie 8080. Otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby obejrzeć treść zastępczą projektu (rysunek 11.1).



Rysunek 11.1. Przykładowa aplikacja po uruchomieniu

Omówienie składników komponentu

Komponent stanowi podstawowy element budowy aplikacji Vue.js. Praktycznie wszystkie projekty składają się z co najmniej kilku komponentów. Komponenty są definiowane w plikach `.vue`, które zawierają treści HTML wyświetlane użytkownikowi, dane, kod JavaScript do obsługi treści i style CSS. Zgodnie z popularną konwencją w projektach Vue.js należy zdefiniować komponent główny (nazywany korzeniem, ang. *root*), który koordynuje działanie reszty aplikacji. Jest to komponent o nazwie *App*, zlokalizowany w pliku `src/App.vue`. Domyślne opcje projektu, z których korzystam w przykładowym projekcie, powodują powstanie pliku jak w listingu 11.6.

Listing 11.6. Początkowa zawartość pliku `src/App.vue`

```
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
<script>
import HelloWorld from './components/HelloWorld.vue'
export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>
<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Komponenty są definiowane za pomocą trzech elementów: `template`, `script` i `style`, które opisuję w kolejnych punktach.

Omówienie elementu template

Element `template` zawiera kod HTML związanego z komponentem. Szablony zawierają jeden element najwyższego poziomu (w listingu 11.6 jest to element `div`), który zamienia oryginalny element w dokumencie bazowym (por. rozdział 9.). Szablon komponentu składa się z mieszanki zwykłych elementów HTML i rozszerzeń Vue.js, które pozwalają na stosowanie różnorakich mechanizmów, takich jak wiązania danych, dyrektywy czy własne elementy włączające do dokumentu inne komponenty. Przykłady tych funkcji widziałeś już w poprzednich rozdziałach, jednak dopiero w tym miejscu (a także w dalszych rozdziałach) omawiam je szczegółowo.

Omówienie elementu script

Element `script` zawiera moduł JavaScript komponentu. Jego właściwości konfiguruują działanie komponentu, określają model danych, a także dostarczają logikę, która pozwala obsłużyć możliwości tegoż komponentu. Różne role, jakie pełni element `script`, oznaczają, że nie zawsze łatwo jest zrozumieć wszystko to, co się dzieje w jego ramach, na początku przygody z Vue.js. Z pewnością niebawem przyzwyczaisz się do stosowania typowych opcji konfiguracyjnych.

Omówienie elementu style

Element `style` definiuje style CSS, które można stosować do wybranych składników szablonu HTML lub wdrażać na szerszą skalę — np. w całej aplikacji. Nie wszystkie komponenty muszą mieć własne style CSS, dlatego często w listingach znajdziesz jedynie elementy `template` i `script`, zwłaszcza gdy korzystasz z framework'ów takich jak Bootstrap. Z pewnością można wyróżnić kilka przydatnych możliwości języka CSS zapewianych przez element `script`. Warto wiedzieć, jak można z nich skorzystać, nawet jeśli na co dzień używa się framework'a CSS.

Jak przekonasz się niebawem, to właśnie odpowiedni sposób połączenia elementów `template`, `script` i `style` pozwala na tworzenie użytecznych komponentów. To te elementy wyróżniają Vue.js na tle innych, podobnych technologii.

Zmiany komponentu w przykładowej aplikacji

Komponent w pliku `App.vue` nie jest zbyt funkcjonalny, jednak zanim zacznijemy dodawać do niego nowe możliwości, oczyszczę go z istniejących mechanizmów. W listingu 11.7 upraszczam elementy `template` i `script`, a także usuwam element `style`, dzięki czemu otrzymuję prosty, przejrzysty kod do dalszej pracy w tym rozdziale.

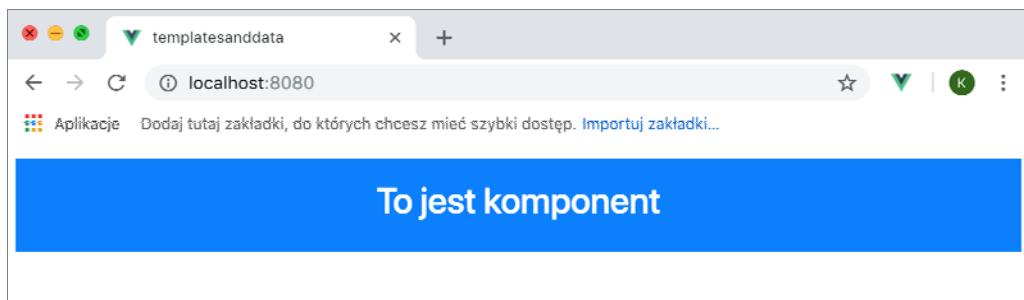
Listing 11.7. Czyszczenie treści w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>To jest komponent</h3>
  </div>
</template>
<script>
  export default {
    name: "MyComponent"
  }
</script>
```

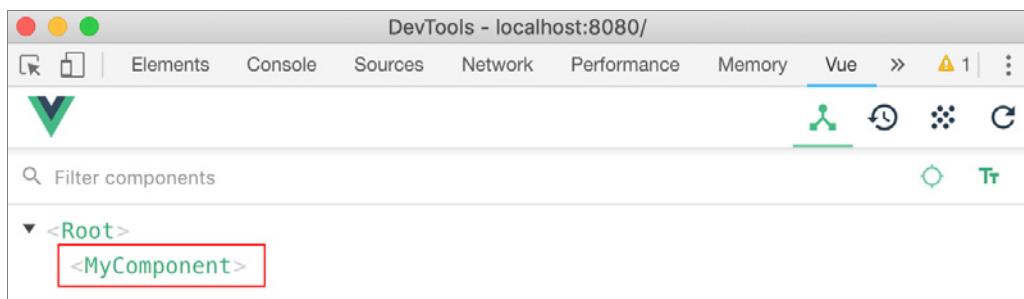
Element `template` składa się z elementu `div` na najwyższym poziomie — do elementu `div` przypisałem kilka klas Bootstrapa. W ten sposób mogę zastosować style bez deklaracji elementu `style` w komponencie. Element `div` zawiera pojedynczy element `h3` zawierający prosty komunikat tekstowy.

Po zapisaniu zmian w pliku `App.vue` projekt zostanie skompilowany, przeglądarka odświeży swoją zawartość, a Ty zobaczysz treść jak na rysunku 11.2.

Jedyną właściwością zdefiniowaną w elemencie `script` jest `name`. Rozszerzenie Vue Devtools przeglądarki (opisane w rozdziale 10.) wykorzystuje opcjonalną właściwość `name` do przedstawienia struktury aplikacji. Wybór wyróżniającej i treściowej nazwy ułatwi analizę komponentu i jego stanu. Otwórz narzędzia deweloperskie F12 i przejdź na zakładkę `Vue`. Zobaczysz, że nazwa komponentu jest wyświetlana jak na rysunku 11.3. W tym momencie ta zakładka nie jest zbyt użyteczna, ale w miarę rozwoju aplikacji jej przydatność z pewnością wzrośnie.



Rysunek 11.2. Uproszczenie komponentu w przykładowej aplikacji



Rysunek 11.3. Analiza struktury aplikacji

Wyświetlanie wartości danych

Jednym z najważniejszych zadań, które może zrealizować komponent, jest wyświetlanie wartości danych. Proces ten składa się z dwóch kroków: zdefiniowania właściwości danych w elemencie `script` i dodania wiązania danych do elementu `template` w celu wyświetlania wartości, jak w listingu 11.8.

Listing 11.8. Wyświetlanie wartości danych w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak"
      }
    }
  }
</script>
```

Wartości danych są definiowane za pomocą właściwości `data` w module JavaScript. Aby zdefiniować wartość, trzeba skorzystać ze specjalnego wzorca: właściwość `data` zwraca funkcję, funkcja zwraca obiekt, i to w ramach tego obiektu możemy definiować właściwości używane do wyświetlania wartości danych użytkownikowi.

Definiowanie wartości danych krok po kroku

Dodanie obsługi wartości danych wymusza na nas spełnienie bardzo specyficznych wymagań. Błędy w tym procesie należą do najczęściej popełnianych w Vue.js. Zaczniemy od zdefiniowania właściwości data, która zwraca funkcję:

```
...
<script>
  export default {
    name: "MyComponent",
    data: function() {
    }
}
</script>
...
```

Po nazwie data należy wstawić dwukropki, słowo kluczowe function, a następnie parę nawiasów okrągłych (znaki (i)), po czym parę nawiasów klamrowych (znaki { i }). Kolejny krok polega na zwróceniu obiektu będącego literałem języka JavaScript (por. rozdział 4.):

```
...
<script>
  export default {
    name: "MyComponent",
    data: function() {
      return {
      }
    }
}
</script>
...
```

Wprowadzamy słowo kluczowe return, po którym następuje ponownie para nawiasów klamrowych. Na zakończenie musimy zdefiniować żądane przez nas wartości danych. Pojedyncza wartość jest określana za pomocą nazwy, dwukropka i faktycznej wartości:

```
...
<script>
  export default {
    name: "MyComponent",
    data: function() {
      return {
        myValue: 10
      }
    }
}
</script>
...
```

Jeśli chcesz podać wiele wartości danych, skorzystaj z przecinka, aby je rozdzielić:

```
...
<script>
  export default {
    name: "MyComponent",
    data: function() {
      return {
        myValue: 10, // <-- zwróć uwagę na przecinek
      }
    }
}
</script>
```

```

        myOtherValue: "Witaj, świecie!"
    }
}
</script>
...

```

To podejście jest wymagane jedynie dla wartości zdefiniowanych we właściwości data. Gdy tylko nabierzesz wprawy w tworzeniu aplikacji w Vue.js, zaczniesz korzystać z tej metody automatycznie. Jeśli zapomnisz zdefiniować funkcję, która zwraca obiekt, zobaczysz ostrzeżenie w konsoli JavaScript w przeglądarce, np.:

[Vue warn]: The "data" options should be a function that returns a per-instance value ↵in component definitions.

Ostrzeżenie nie jest wyświetlane w głównym oknie przeglądarki tak jak błędy kompilatora pokazane w rozdziale 10., ale spowoduje wygenerowanie innych ostrzeżeń i uniemożliwi poprawne działanie wiązań danych.

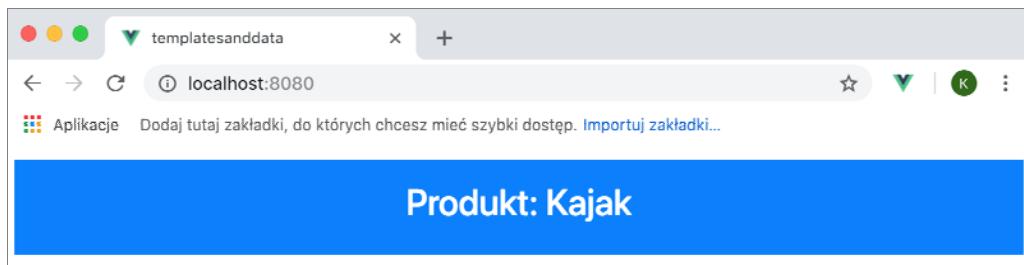
Wartości danych w elemencie script są wiązane z elementami HTML w elemencie template za pomocą wiązań danych. Istnieje wiele różnych rodzajów wiązań, ale zacznijmy od najprostszego, czyli **wiązania interpolacji tekstu** (ang. *text interpolation binding*):

```

...
<h3>Produkt: {{ name }}</h3>
...

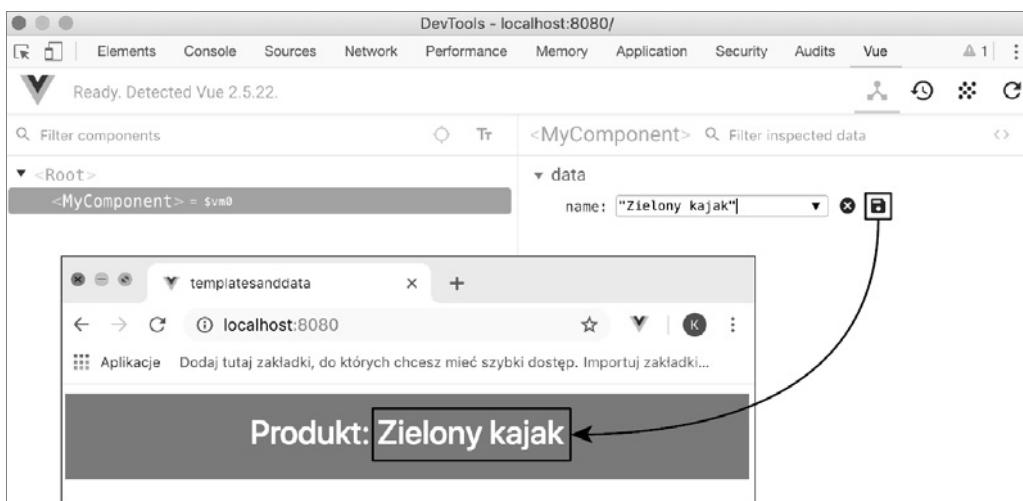
```

Ten rodzaj wiązania wstawia wartość danych do tekstopowej zawartości elementu HTML. Wiązanie jest oznaczone podwójnymi nawiasami klamrowymi ({{ i }}). Podwójne nawiasy swoim wyglądem przypominają wąsy, dlatego też często nazywa się je **wiązaniem wąsatym**. Zapisz zmiany w pliku App.vue, a zobaczyś treść jak na rysunku 11.4.



Rysunek 11.4. Wyświetlenie wartości danych

Zakładka Vue w narzędziach deweloperskich F12 również zostanie zaktualizowana i pokaże wartość danych komponentu. Wartości te funkcjonują na żywo, tak więc zmiana wartości w zakładce Vue spowoduje pokazanie nowej wartości w aplikacji. W przykładowej aplikacji oznacza to, że zmiana wartości właściwości name komponentu spowoduje automatyczne odświeżenie wiązania danych w elemencie template. Aplikacja nie ma obecnie możliwości zmiany wartości name, ale jeśli przesunesz kursor na wartość w panelu Vue Devtools, zobaczysz ikonkę ołówka. Po jej kliknięciu zostanie włączony edytor. Zmień wartość na Zielony kajak (pamiętaj o zachowaniu cudzysłowów), a następnie kliknij ikonkę dyskietki, aby zapisać zmiany. Vue.js wykryje zmianę we właściwości danych i zaktualizuje kod HTML jak na rysunku 11.5.



Rysunek 11.5. Zmiana właściwości danych za pomocą narzędzia Vue Devtools

Stosowanie złożonych wyrażeń w wiązaniach danych

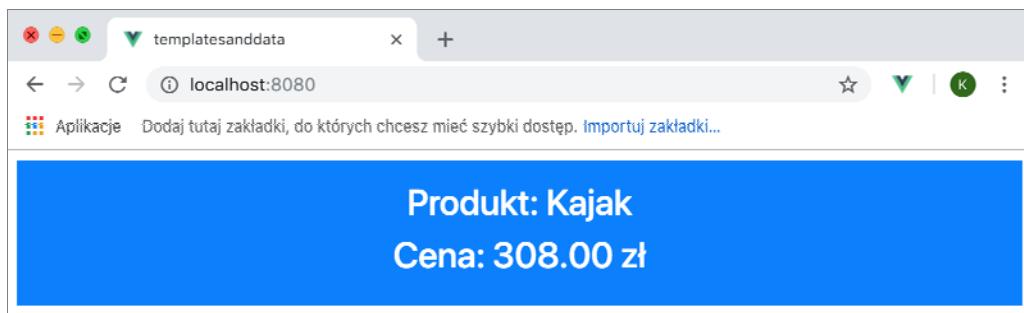
Gdy Vue.js wyświetla wiązanie danych, jego zawartość jest traktowana jako wyrażenie języka JavaScript. Jeśli zawartość wiązania jest nazwą jednej z właściwości data, ewaluacja wyrażenia sprowadza się do pobrania wartości właściwości (tak jak w przypadku wiązania `{} name {}`).

Traktowanie wiązań danych jako wyrażeń pozwala na stosowanie bardziej zaawansowanych instrukcji języka JavaScript w obrębie wiązań. W listingu 11.9 dodaję dwie kolejne właściwości w sekcji data do elementu script, a także bardziej złożone wiązania danych do elementu template.

Listing 11.9. Stosowanie złożonych wyrażeń w wiązaniach danych w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h3>Cena: {{ (price + (price * (taxRate / 100))).toFixed(2) }} zł</h3>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        taxRate: 12
      }
    }
  }
</script>
```

Do komponentu dodałem dwie właściwości danych: `price` i `taxRate`, z których korzystam w nowym wiązaniu danych w celu obliczenia łącznej ceny produktu. Formatuję wynik w taki sposób, aby uzyskać dwie cyfry po kropce, dzięki czemu otrzymuję wynik jak na rysunku 11.6, uzyskany oczywiście po skompilowaniu projektu i odświeżeniu przeglądarki.



Rysunek 11.6. Złożone wyrażenie w wiązaniu danych

Wyrażenia mogą zawierać tylko pojedyncze instrukcje, których wykonanie musi zwracać jakąś wartość. W związku z tym nie możesz umieścić w nich wywołania funkcji ani innych skomplikowanych zadań. Kontekstem wyrażenia jest komponent, co oznacza, że dostęp do danych jest ograniczony tylko do danych zdefiniowanych wewnątrz komponentu. W związku z tym można skorzystać z właściwości name, price i taxRate bez podawania skomplikowanych nazw kwalifikowanych. Z drugiej strony nie można korzystać z globalnych obiektów dostarczanych przez przeglądarki (np. window czy document), a także danych dostępnych w innych komponentach.

Wyrażenia mają dostęp do popularnych obiektów i funkcji globalnych, takich jak obiekty Math czy JSON, dzięki którym można uzyskać dostęp do funkcji matematycznych i obsługi formatu danych JSON. Najbardziej użyteczne funkcje i obiekty globalne, z których możemy skorzystać w szablonie, są opisane w tabeli 11.3.

Tabela 11.3. Użyteczne funkcje i obiekty globalne, dostępne w wyrażeniach w wiązaniach danych

Nazwa	Opis
parseFloat	Ta funkcja konwertuje argument na liczbę zmiennoprzecinkową.
parseInt	Ta funkcja konwertuje argument na liczbę całkowitą.
Math	Ten obiekt oferuje funkcje matematyczne.
Number	Ten obiekt oferuje metody użyteczne w pracy z wartościami liczbowymi.
Date	Ten obiekt oferuje metody użyteczne w pracy z datami.
Array	Ten obiekt oferuje metody użyteczne w pracy z tablicami.
Object	Ten obiekt oferuje metody użyteczne w pracy z obiektami.
String	Ten obiekt oferuje metody użyteczne w pracy z łańcuchami znaków (tekstami).
RegExp	Ten obiekt jest używany do obsługi wyrażeń regularnych.
Map	Ten obiekt jest używany do reprezentowania zbioru par klucz-wartość.
Set	Ten obiekt jest używany do reprezentowania kolekcji unikatowych wartości lub obiektów.
JSON	Ten obiekt jest używany do serializowania i deserializowania danych w formacie JSON.
Intl	Ten obiekt daje dostęp do mechanizmu formatowania zgodnie z ustawieniami lokalizacji (listing 11.15). Szczegóły działania tego mechanizmu znajdziesz na stronie https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl .

Lista z tabeli 11.3 przedstawia funkcje najczęściej używane podczas tworzenia aplikacji w języku JavaScript, jednak nie zawiera opisu obiektów, które mogłyby wykonywać niebezpieczne operacje, takie jak manipulacja drzewem DOM.

Omówienie niezdefiniowanych właściwości danych

Jednym z problemów ewaluowania wiązań danych jako wyrażeń jest trudność w znalezieniu potencjalnych błędów. W JavaScript mamy do czynienia z dość luźnym podejściem do prób dostępu do właściwości, które nie istnieją. W takiej sytuacji zwracana jest specjalna wartość `undefined`.

Vue.js ewaluuje wyrażenie, ale w takiej sytuacji nie wstawia wyniku bezpośrednio do kodu HTML. Zamiast tego do konsoli przeglądarki JavaScript trafia ostrzeżenie:

```
...
[Vue warn]: Property or method "category" is not defined on the instance but referenced
→during render.
...
```

Tego rodzaju błędy wynikają najczęściej z literówek, gdy nazwa właściwości danych nie pasuje do nazwy użytej w wiązaniu danych. Warto więc obserwować konsolę JavaScript podczas prac nad kodem, ponieważ tego rodzaju problem nie spowoduje powstania błędu komilacji ani nie zostanie wyświetlony w przeglądarce.

Przeliczanie wartości we właściwościach obliczanych

Stosowanie prostych wyrażeń w wiązaniach danych stanowi dobrą praktykę, ponieważ znacznie łatwiej czyta się szablony, są one łatwiejsze w utrzymaniu, a kodu można łatwo użyć ponownie. Zachowanie prostych szablonów jest możliwe dzięki zastosowaniu właściwości obliczonych (ang. *computed properties*), które generują wartości na podstawie właściwości danych. Dzięki temu nie musimy umieszczać skomplikowanych wyrażeń w wiązaniach danych. W listingu 11.10 definiuję właściwość obliczaną, która oblicza cenę łączną produktu za pomocą wartości `price` i `taxRate`.

Listing 11.10. Stosowanie właściwości obliczanej w pliku `src/App.vue`

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h3>Cena: {{ totalPrice.toFixed(2) }} zł</h3>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        taxRate: 12
      }
    },
    computed: {
      totalPrice: function() {
        return this.price + (this.price * (this.taxRate / 100));
      }
    }
  }
</script>
```

```

        }
    }
</script>

```

Właściwość obliczona jest dodawana do modułu JavaScript komponentu. Otrzymuje ona literał obiektowy, którego nazwami właściwości są nazwy właściwości obliczanych. Niektóre mechanizmy komponentu są trudne do opisania, ponieważ są tak bardzo zależne od pojęcia **właściwości**, że być może będziesz musiał przeczytać poprzednie zdanie kilkakrotnie, aby dobrze zrozumieć jego sens.

-
- **Uwaga** Zwróć uwagę, że korzystam ze słowa kluczowego `this` w funkcji właściwości obliczanej, aby uzyskać dostęp do właściwości danych. Jest to niezbędne, ponieważ w przeciwnym razie powstanie błąd informujący, że właściwość o podanej nazwie nie istnieje.
-

W listingu definiuję właściwość obliczaną `totalPrice` związaną z funkcją, która wykonuje obliczenia umieszczone poprzednio w wiązaniu danych. Właściwości obliczane są używane tak jak zwykle właściwości w wiązaniach danych, co oznacza, że można z nich korzystać tak jak poniżej:

```

...
<h3>Cena: { totalPrice.toFixed(2) } zł</h3>
...

```

Reaktywność a właściwości obliczane

Vue.js optymalizuje proces odświeżania interfejsu użytkownika, ewaluując właściwość obliczaną tylko, gdy następuje zmiana jednej spośród wartości, od których owa właściwość zależy. Dla przykładu dodaj instrukcję do funkcji właściwości obliczanej, aby łatwiej zaobserwować, kiedy jest ona wywoływana (listing 11.11).

Listing 11.11. Obserwowanie właściwości obliczanej w pliku `src/App.vue`

```

<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h3>Cena: { totalPrice.toFixed(2) } zł</h3>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        taxRate: 12
      }
    },
    computed: {
      totalPrice: function () {
        let tp = this.price + (this.price * (this.taxRate / 100));
        console.log(`Obliczono: ${tp} (${this.taxRate})`);
        return tp;
      }
    }
  }
</script>

```

- **Wskazówka** Nie musisz przejmować się optymalizacją reaktywności w większości projektów. Warto jednak zrozumieć, jak działają różne dostępne mechanizmy, stąd też obecność tego przykładu. W rozdziale 17. nieco szerzej omawiam reaktywność w aplikacjach Vue.js.

Korzystam z metody `console.log`, aby wyświetlić komunikat w konsoli JavaScript w przeglądarce przy każdym wywołaniu funkcji.

Aplikacja nie musi umożliwiać użytkownikowi zmiany wartości danych. Aby wykonać test, korzystam z narzędzi Vue Devtools do zwiększenia wartości właściwości `taxRate`. Przy każdej zmianie właściwości zostanie wyświetlony komunikat odzwierciedlający wprowadzoną zmianę.

```
...
Obliczono: 310.75 (13)
Obliczono: 313.5 (14)
Obliczono: 316.25 (15)
...
```

Vue.js „wie”, że właściwość `totalPrice` zależy od wartości właściwości `taxRate`, dzięki czemu wywołuje funkcję przy każdej zmianie właściwości `taxRate`, co sprawia, że wyrażenie wiążania danych może być ewaluowane ponownie.

Efekty uboczne we właściwościach obliczanych

W funkcjach właściwości obliczanych nie powinno się dodawać instrukcji, które wprowadzają jakiekolwiek zmiany pośród danych — takie zachowanie nosi nazwę efektów ubocznych (ang. *side effects*). Efekty uboczne utrudniają zrozumienie działania aplikacji i zmniejszają wydajność procesu odświeżania Vue.js.

W związku z tym reguły lintera Vue.js (opisane w rozdziale 10.) weryfikują obecność efektów ubocznych we właściwościach obliczanych. Oto przykład komponentu, który zawiera efekt uboczy:

```
<template>
    <div class="bg-primary text-white text-center m-2 p-3">
        <h3>Produkt: {{ name }}</h3>
        <h3>Cena: {{ totalPrice.toFixed(2) }} zł</h3>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "Kajak",
                price: 275,
                taxRate: 12,
                counter: 0
            }
        },
        computed: {
            totalPrice: function () {
                let tp = this.price + (this.price * (this.taxRate / 100));
                console.log(`Obliczono: ${this.counter}`)
                `${tp}(${this.taxRate})`;
                return tp;
            }
        }
    }
</script>
```

Do listingu dodałem także zmienną counter, której wartość jest dołączona do łańcucha znaków w metodzie `console.log`. Efektem ubocznym jest inkrementowanie zmiennej za pomocą operatora `++`. Linter wykrywa efekt uboczy i zgłasza następujący błąd:

```
...
Unexpected side effect in "totalPrice" computed property console.log(`Obliczono:
(${this.counter++)})
...
```

Powyższą regułę, `vue/no-side-effects-in-computed-properties`, można wyłączyć, jednak na pewno należy dbać o ograniczenie efektów ubocznych — mogą one łatwo doprowadzić do nieoczekiwanej zachowania i dlatego powinno się ich unikać.

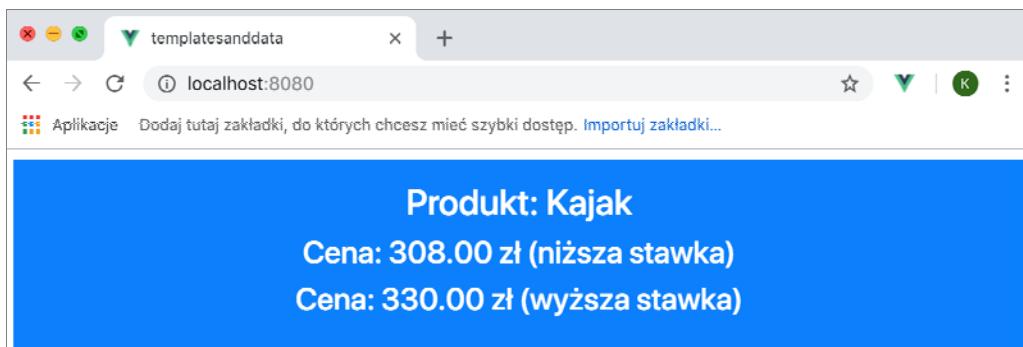
Obliczanie wartości danych za pomocą metody

Głównym ograniczeniem właściwości obliczanych jest fakt, że nie można dostosowywać wartości używanych do wygenerowania wyniku. Jeśli chciałbym obliczyć łączny koszt, korzystając z dwóch stawek podatku, musiałbym utworzyć dwie właściwości zawierające podobne obliczenia (listing 11.12).

Listing 11.12. Obliczanie podobnych wartości w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h4>Cena: {{ lowTotalPrice.toFixed(2) }} zł (niższa stawka)</h4>
    <h4>Cena: {{ highTotalPrice.toFixed(2) }} zł (wyższa stawka)</h4>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        lowTaxRate: 12,
        highTaxRate: 20
      }
    },
    computed: {
      lowTotalPrice: function () {
        let tp = this.price + (this.price * (this.lowTaxRate / 100));
        return tp;
      },
      highTotalPrice: function () {
        let tp = this.price + (this.price * (this.highTaxRate / 100));
        return tp;
      }
    }
  }
</script>
```

Teraz mamy do czynienia z dwoma stawkami podatku i obliczenia są wykonywane dla każdej z nich, co daje efekt jak na rysunku 11.7.



Rysunek 11.7. Wykonywanie wielu obliczeń

Rozwiązywanie z listingu 11.12 działa prawidłowo, jednak nie skaluje się zbyt dobrze. Zdecydowanie nie lubię sytuacji, w których dodanie kolejnej wartości bądź opcji w aplikacji polega na skopiowaniu i wklejeniu istniejącego kodu. To kwestia czasu, aż w końcu pomyślę się i nie zaktualizuję wklejonego kodu.

Zdecydowanie lepiej jest zdefiniować metodę, która pozwoli skorzystać z raz napisanego kodu wiele razy. Taka metoda obliczy wartości dla różnych cen bez duplikowania kodu. W listingu 11.13 definiuję metodę, która oblicza cenę produktu.

Listing 11.13. Definiowanie metody w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h4>Cena: {{ lowTotalPrice.toFixed(2) }} zł (niższa stawka)</h4>
    <h4>Cena: {{ highTotalPrice.toFixed(2) }} zł (wyższa stawka)</h4>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        lowTaxRate: 12,
        highTaxRate: 20
      }
    },
    computed: {
      lowTotalPrice: function () {
        return this.getTotalPrice(this.lowTaxRate);
      },
      highTotalPrice: function () {
        return this.getTotalPrice(this.highTaxRate);
      }
    },
    methods: {
      getTotalPrice(taxRate) {
        return this.price + (this.price * (taxRate / 100));
      }
    }
  }
</script>
```

Aby zdefiniować metodę, do obiektu konfiguracji komponentu dodałem właściwość `methods`, w której zadeklarowałem funkcję. W przeciwnieństwie do właściwości obliczanych metoda może przyjmować parametry. W naszym przykładzie funkcja nosi nazwę `getTotalPrice` i korzysta z parametru `taxRate`, dzięki któremu jest w stanie obliczyć wynik. Następnie wywołuję metodę z funkcji właściwości obliczanych, np.:

```
...
lowTotalPrice: function () {
    return this.getTotalPrice(this.lowTaxRate);
},
...
...
```

Zwróć uwagę, że również w przypadku metody poprzedzam jej nazwę słowem `this` (podobnie jak w przypadku właściwości danych). Słowo `this` nie jest za to wymagane w celu uzyskania dostępu do parametru.

```
...
getTotalPrice(taxRate) {
    return this.price + (this.price * (taxRate / 100));
}
...
...
```

Wywoływanie metod bezpośrednio z wiązań danych

Zastosowanie metody pozwala mi na uproszczenie kodu wymaganego do obliczenia ceny łącznej. Mogę jednak pójść o krok dalej i usunąć właściwości obliczane, wywołując metody bezpośrednio z wiązań danych (listing 11.14).

Listing 11.14. Wywoływanie metod bezpośrednio w pliku src/App.vue

```
<template>
    <div class="bg-primary text-white text-center m-2 p-3">
        <h3>Produkt: {{ name }}</h3>
        <h4>Cena: {{ getTotalPrice(lowTaxRate).toFixed(2) }} zł (niższa stawka)</h4>
        <h4>Cena: {{ getTotalPrice(highTaxRate).toFixed(2) }} zł (wyższa stawka)</h4>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "Kajak",
                price: 275,
                lowTaxRate: 12,
                highTaxRate: 20
            }
        },
        methods: {
            getTotalPrice(taxRate) {
                return this.price + (this.price * (taxRate / 100));
            }
        }
    }
</script>
```

Ta zmiana pokazuje, jak elastycznie komponenty potrafią ze sobą współpracować. Z drugiej strony złożoność w wiązaniach danych rośnie, a chcielibyśmy tego uniknąć. Jak przekonasz się niebawem, tego rodzaju sytuacje będą pojawiać się nieraz — trzeba znaleźć właściwy kompromis w różnych sytuacjach.

Formatowanie wartości danych za pomocą filtrów

Aby wyświetlić poprawną kwotę w złotówkach, połączylem symbol waluty (PLN) z wywołaniem metody `toFixed`, jak niżej:

```
...
<h4>Cena: {{ getTotalPrice(lowTaxRate).toFixed(2) }} zł (niższa stawka)</h4>
...
```

Takie zachowanie można usprawnić poprzez zastosowanie filtru. Jego zadaniem jest sformatowanie wyniku wyrażenia. W listingu 11.15 dodaję filtr, który wyświetla wartość liczbową jako kwotę pieniędzy.

Listing 11.15. Dodawanie filtru do pliku src/App.vue

```
<template>
    <div class="bg-primary text-white text-center m-2 p-3">
        <h3>Produkt: {{ name }}</h3>
        <h4>Cena: {{ getTotalPrice(lowTaxRate) | currency }} (niższa stawka)</h4>
        <h4>Cena: {{ getTotalPrice(highTaxRate) | currency }} (wyższa stawka)</h4>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "Kayak",
                price: 275,
                lowTaxRate: 12,
                highTaxRate: 20
            }
        },
        methods: {
            getTotalPrice(taxRate) {
                return this.price + (this.price * (taxRate / 100));
            }
        },
        filters: {
            currency(value) {
                return new Intl.NumberFormat("pl-PL",
                    { style: "currency", currency: "PLN" }).format(value);
            }
        }
    }
</script>
```

Filtr są definiowane jako funkcje we właściwości `filters` w ramach obiektu konfiguracji komponentu. Funkcja filtru otrzymuje wartość za pomocą parametru, po czym zwraca sformatowany wynik. W listingu definiuję filtr `currency`, który formatuje wartość za pomocą globalnego obiektu `Intl`. Obiekt ten udostępnia mechanizmy formatowania z uwzględnieniem ustawień lokalizacyjnych (językowych). Dzięki metodzie `NumberFormat` jestem w stanie włączyć ustawienie `pl-PL` (język polski). Następnie przekazuję obiekt konfiguracji, w którym określам symbol polskiej waluty.

Filtrów można zastosować za pomocą symbolu pionowej kreski (znak `|`), np.:

```
...
<h4>Cena: {{ getTotalPrice(lowTaxRate) | currency }} (niższa stawka)</h4>
...
```

Lokalizacja aplikacji

Nie omawiam ustawień lokalizacyjnych w tej książce, ponieważ nie jest to funkcja ściśle związana z Vue.js. Z pewnością jednak jest to ważny aspekt tworzenia aplikacji, któremu warto poświęcić czas, zasoby i należną uwagę.

Lokalizacja to temat złożony, a jednocześnie często traktowany po macoszemu. W wielu aplikacjach przyjmuje się mylne założenie, że użytkownicy zrozumieją i zaakceptują zwyczaje przyjęte w Stanach Zjednoczonych. Jako Brytyjczyk jestem przyzwyczajony do oglądania dat w nieprawidłowym formacie, a kwot (wyrażonych w funtach) z symbolem dolara amerykańskiego. Mam o tyle dobrze, że jestem chociaż w stanie zrozumieć komunikaty tekstowe pisane po angielsku — wielu użytkowników nie zna jednak tego języka.

Z drugiej strony brak lokalizacji bywa lepszy niż lokalizacja zrobiona źle. Dzieje się tak często, gdy nie współpracujesz z profesjonalistą, osobą znającą język albo gdy nie poświęcasz lokalizacji dostatecznie dużo zasobów i uwagi. Dostęp do API lokalizacyjnego to tylko niewielki element tego, co potrzeba, aby skutecznie i prawidłowo zlokalizować aplikację. Należy także pamiętać o uwarunkowaniach kulturowych i językowych, z którymi nie poradzi sobie tłumacz taki jak Google Translate.

To wyrażenie sprawi, że Vue.js sformatuje wynik działania metody `getTotalPrice` za pomocą filtru `currency`. Nie zobaczysz żadnej widocznej zmiany w treści, ale dzięki filtrowi komponent stanie się bardziej zwięzły — wiązanie danych znacząco się uprości, a jednocześnie w łatwy sposób zagwarantujemy prawidłowe formatowanie wartości. Filtry mogą wyglądać podobnie jak pozostałe elementy obiektu konfiguracji, ale mają swoje specjalne cechy, o których piszę poniżej.

Konfiguracja filtrów z wykorzystaniem argumentów

Funkcje filtrów nie mają dostępu do pozostałych danych komponentu, co oznacza, że nie możesz wygenerować wyniku filtru na podstawie żadnej innej wartości. Izolacja filtrów zapewnia, że nie wpłyną one na właściwości stanu — dzięki temu Vue.js nie musi obserwować zmian stanu po wywołaniach funkcji filtrów (w przeciwieństwie do wywołań metod).

Formatowanie może jednak być dość skomplikowanym procesem, dlatego filtry Vue.js przyjmują argumenty. W listingu 11.16 dodaję argument do filtru `currency`, który pozwala na określenie liczby miejsc po przecinku, z jaką mają być wyświetlane wartości.

Listing 11.16. Dodawanie argumentu filtru w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3>Produkt: {{ name }}</h3>
    <h4>Cena: {{ getTotalPrice(lowTaxRate) | currency(3) }} (niższa stawka)</h4>
    <h4>Cena: {{ getTotalPrice(highTaxRate) | currency } } (wyższa stawka)</h4>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kajak",
        price: 275,
        lowTaxRate: 12,
        highTaxRate: 20
      }
    },
    methods: {

```

```

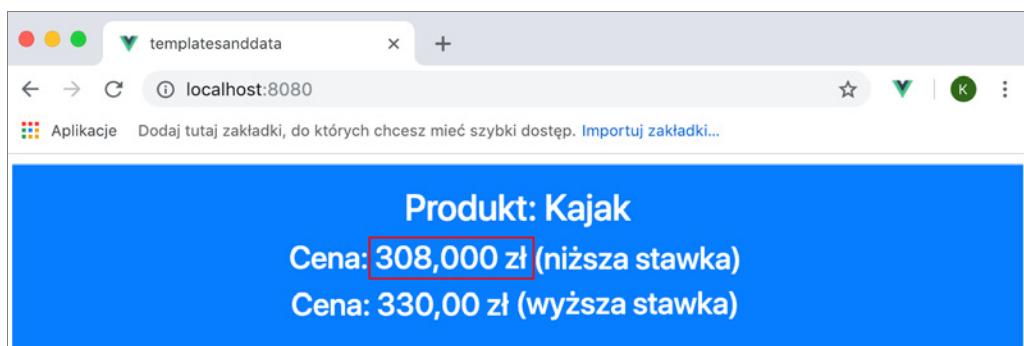
        getTotalPrice(taxRate) {
            return this.price + (this.price * (taxRate / 100));
        }
    },
    filters: {
        currency(value, places) {
            return new Intl.NumberFormat("pl-PL",
            {
                style: "currency", currency: "PLN",
                minimumFractionDigits: places || 2,
                maximumFractionDigits: places || 2
            }).format(value);
        }
    }
}
</script>

```

Do funkcji currency dodałem parametr places, z którego korzystam do określania właściwości minimumFractionDigits i maximumFractionDigits obiektu konfiguracji przekazanego do metody `Intl.NumberFormat`. W ten sposób ustalam liczbę miejsc po przecinku jako stałą wartość.

Gdy definiujesz parametry funkcji filtru, warto podać wartość domyślną. W tym przypadku wartość ta wynosi 2 i zostanie zastosowana, jeśli filtr zostanie wykonany bez argumentu.

Argumenty są przekazywane do filtru w wyrażeniu wiązania danych. Zmieniłem jedno z wiązań, aby określić trzy cyfry po przecinku, co w rezultacie daje wynik jak na rysunku 11.8.



Rysunek 11.8. Dodawanie argumentu do filtru

Łączenie filtrów

Filtryle można łączyć, dzięki czemu wynik jednego filtru staje się argumentem drugiego, co pozwala uzyskać łańcuchy filtrów. W ten sposób możemy formatować wartość, łącząc ze sobą różne operacje i uzyskując w ten sposób bardzo wysoki poziom kontroli nad całym procesem formatowania. W listingu 11.17 definiuję dwa nowe filtry, łącząc je w łańcuchach w szablonie. Zmieniłem także szczegóły dotyczące produktu, aby łatwiej było zauważyc różnicę (słowo *kajak* to palindrom, a jeden z filtrów odwraca kolejność znaków w łańcuchu).

Listing 11.17. Łączenie filtrów w łańcuchach w pliku src/App.vue

```

<template>
    <div class="bg-primary text-white text-center m-2 p-3">
        <h3>Produkt: {{ name | reverse | capitalize }}</h3>
        <h4>Cena: {{ getTotalPrice(lowTaxRate) | currency(3) }} (niższa stawka)</h4>
        <h4>Cena: {{ getTotalPrice(highTaxRate) | currency }} (wyższa stawka)</h4>
    </div>

```

```

</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "Kamizelka ratunkowa",
                price: 48.95,
                lowTaxRate: 12,
                highTaxRate: 20
            }
        },
        methods: {
            getTotalPrice(taxRate) {
                return this.price + (this.price * (taxRate / 100));
            }
        },
        filters: {
            currency(value, places) {
                return new Intl.NumberFormat("pl-PL",
                {
                    style: "currency", currency: "PLN",
                    minimumFractionDigits: places || 2,
                    maximumFractionDigits: places || 2
                }).format(value);
            },
            capitalize(value) {
                return value[0].toUpperCase() + value.slice(1);
            },
            reverse(value) {
                return value.split("").reverse().join("");
            }
        }
    }
</script>

```

Nowe filtry to `capitalize`, który formatuje pierwszą literę argumentu jako wielką, i `reverse`, który odwraca kolejność znaków w tekście. Filtry są łączone w łańcuch za pomocą znaku pionowej kreski:

```

...
<h3>Produkt: {{ name | reverse | capitalize }}</h3>
...

```

To wyrażenie pozwoli na sformatowanie wartości `name` najpierw za pomocą filtru `reverse`, a potem — `capitalize`, co da efekt jak na rysunku 11.9.

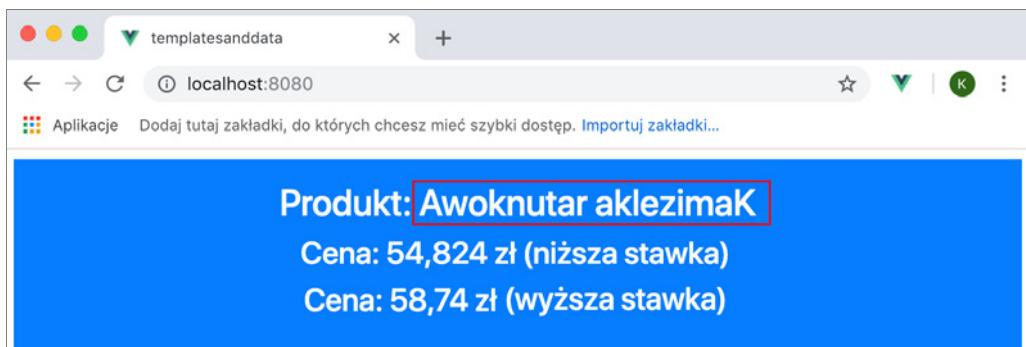
Filtrysą stosowane w kolejności użycia, tak więc jej zmiana prowadzi często do uzyskania innych wyników. W listingu 11.18 zmieniam kolejność filtrów w przykładzie.

Listing 11.18. Zmiana kolejności filtrów w pliku `src/App.vue`

```

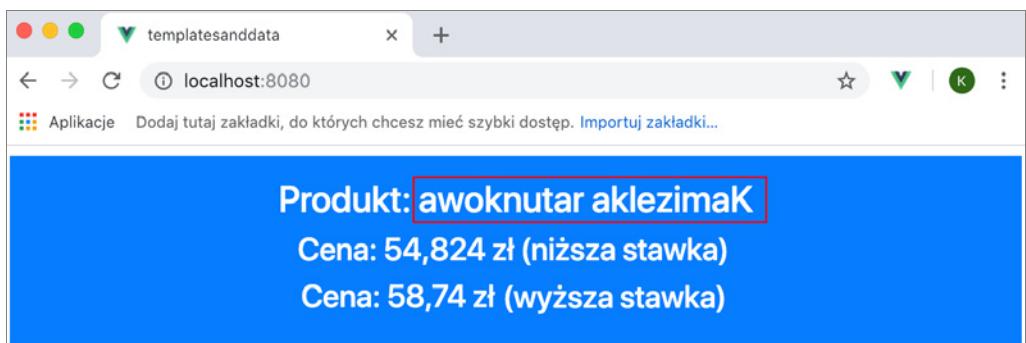
...
<template>
    <div class="bg-primary text-white text-center m-2 p-3">
        <h3>Produkt: {{ name | capitalize | reverse }}</h3>
        <h4>Cena: {{ getTotalPrice(lowTaxRate) | currency(3) }} (niższa stawka)</h4>
        <h4>Cena: {{ getTotalPrice(highTaxRate) | currency } } (wyższa stawka)</h4>
    </div>
</template>
...

```



Rysunek 11.9. Łączenie filtrów w łańcuch

Wartość name jest przekazywana do filtru capitalize, a następnie do filtru reverse, co daje efekt jak na rysunku 11.10. Filtr capitalize nie ma w tym przypadku znaczenia, ponieważ pierwsza litera wartości name jest wielka na samym początku całej operacji.



Rysunek 11.10. Efekt zmiany kolejności filtrów

Definiowanie globalnego filtru

Jeżeli definiujesz filtr w ramach komponentu, możesz skorzystać z niego w jego szablonie, a także w szablonie jego dzieci (komponenty-dzieci zostały omówione w rozdziale 16.). Jeśli chcesz zdefiniować filtr dostępny w całej aplikacji, skorzystaj z metody Vue.filter, aby zarejestrować filtr przed utworzeniem obiektu Vue:

```
...
Vue.filter("currency", (value) => new Intl.NumberFormat("pl-PL", { style: "currency",
  currency: "PLN" }).format(value));
...

```

Pierwszy argument określa nazwę, pod którą filtr będzie dostępny, a drugi — funkcję formatującą wartość danych.

Podsumowanie

W tym rozdziale wprowadziłem wiązanie interpolacji tekstu, stosowane do przedstawiania danych użytkownikowi. Vue.js obsługuje również bardziej zaawansowane sposoby łączenia modelu danych z elementami szablonu, ale interpolacja tekstu jest najprostsza w użyciu i ułatwia zrozumienie zasad działania Vue.js. W kolejnym rozdziale zajmiemy się dyrektywami.

ROZDZIAŁ 12.



Stosowanie podstawowych dyrektyw

Dyrektywy to specjalne atrybuty, które stosują mechanizmy Vue.js do elementów HTML w szablonie komponentu. W tym rozdziale omówię podstawowe, wbudowane dyrektywy dostarczane w Vue.js, dzięki którym można osiągnąć typowe funkcje niezbędne podczas tworzenia aplikacji webowych. W rozdziałach 13. – 15. opisuję bardziej zaawansowane dyrektywy, a w rozdziale 26. przedstawiam metodę tworzenia własnych dyrektyw, jeśli te wbudowane nie dadzą tego, czego oczekujesz. Tabela 12.1 umiejscowia wbudowane dyrektywy w szerszym kontekście.

Tabela 12.1. Umiejscowienie wbudowanych dyrektyw w szerszym kontekście

Pytanie	Odpowiedź
Czym są wbudowane dyrektywy?	Wbudowane dyrektywy zapewniają funkcje niezbędne podczas tworzenia aplikacji webowych. Opisywane przeze mnie w tym rozdziale dyrektywy ułatwiają zarządzanie elementami HTML i tekstowymi, pozwalając na określenie widoczności elementu, a także zarządzanie jego atrybutami. Dyrektywy umożliwiają również reagowanie na zdarzenia użytkownika, powtarzanie treści i obsługę pól formularzy, co opisuję w dalszych rozdziałach.
Dlaczego są użyteczne?	Dyrektywy ułatwiają łączenie kodu z komponentu script z treścią zadeklarowaną w szablonie template.
Jak się z nich korzysta?	Dyrektywy są stosowane wobec elementów HTML w formie specjalnych atrybutów, których nazwy zaczynają się od znaków v-, np. v-text lub v-bind.
Czy są jakieś pułapki lub ograniczenia?	Niektóre wbudowane dyrektywy są niezbyt intuicyjne w użyciu i mogą generować trudne do przewidzenia wyniki. Dotyczy to zwłaszcza dyrektyw opisanych w dalszych rozdziałach.
Czy są jakieś rozwiązania alternatywne?	Nie. Dyrektywy stanowią spojwo łączące kod HTML z kodem JavaScript komponentu.

Tabela 12.2 przedstawia podsumowanie rozdziału.

Tabela 12.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Ustaw zawartość tekstową elementu.	Skorzystaj z wiązania interpolacji tekstu lub dyrektywy v-text.	12.3
Wyświetl kod HTML.	Skorzystaj z dyrektywy v-html.	12.4 – 12.5
Wyświetl wybrane elementy.	Skorzystaj z dyrektyw v-if, v-else lub v-show.	12.6, 12.9 – 12.13
Wyświetl wybrane elementy równorzędne.	Skorzystaj z dyrektywy wobec elementu template.	12.7 – 12.8
Ustaw atrybuty i właściwości.	Skorzystaj z dyrektywy v-bind.	12.14 – 12.19

Przygotowania do tego rozdziału

W tym rozdziale kontynuujemy projekt *templatesanddata* rozpoczęty w rozdziale 11. Aby przygotować się do tego rozdziału, uproszcilem komponent główny (korzeń) aplikacji (listing 12.1).

Listing 12.1. Uproszczenie treści w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: {{ name }}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    },
    methods: {
      handleClick() {
        // nie rób nic
      }
    }
  }
</script>
```

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Skorzystałem z wiązania interpolacji tekstu, aby wyświetlić wartość właściwości data o nazwie name, co opisałem w rozdziale 11. Dodalem także element button, do którego zastosowałem dyrektywę v-on:

```
...
<button v-on:click="handleClick" class="btn btn-primary">
...

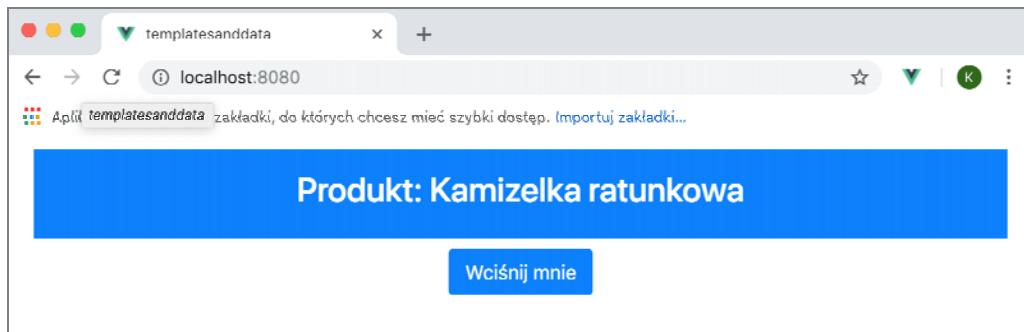
```

Dyrektywa `v-on` jest używana do obsługi zdarzeń, co szerzej opisuję w rozdziale 14. Niektóre dyrektywy opisywane w tym rozdziale i kolejnych są użyteczne jedynie w momencie zmiany stanu aplikacji. Potrzebuję więc kodu, który wyzwoli te zmiany. Wspomniana dyrektywa reaguje na zdarzenie `click`, wyzwalane w momencie kliknięcia elementu `button` przez użytkownika. Efektem działania tej dyrektywy jest wywołanie metody `handleClick`. Zdefiniowałem ją w sekcji `methods` elementu `script` naszego komponentu. W tym momencie nie zawiera ona żadnych instrukcji. W dalszej części rozdziału skorzystam z niej do przedstawienia użytecznych funkcji dyrektyw. Zapisz zmiany w pliku `App.vue` i wykonaj polecenie z listingu 12.2 w katalogu `templatesanddata`, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 12.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Otwórz okno przeglądarki i przejdź pod adres `http://localhost:8080`, aby zobaczyć efekt jak na rysunku 12.1.



Rysunek 12.1. Przykładowa aplikacja

Ustawianie zawartości tekstowej elementu

Warto zacząć od najprostszej dyrektywy, która wykona znane Ci zadanie — ustawi (wyrenderuje) zawartość tekstową elementu. W listingu 12.3 wiązanie interpolacji tekstu zastępuję dyrektywą.

Listing 12.3. Zastosowanie dyrektywy w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: <span v-text="name"></span></h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
```

```

        return {
            name: "Kamizelka ratunkowa"
        }
    },
    methods: {
        handleClick() {
            // nie rób nic
        }
    }
}
</script>

```

Rysunek 12.2 przedstawia dokładnie budowę dyrektywy i jej połączenie z elementem, w którym została zastosowana.

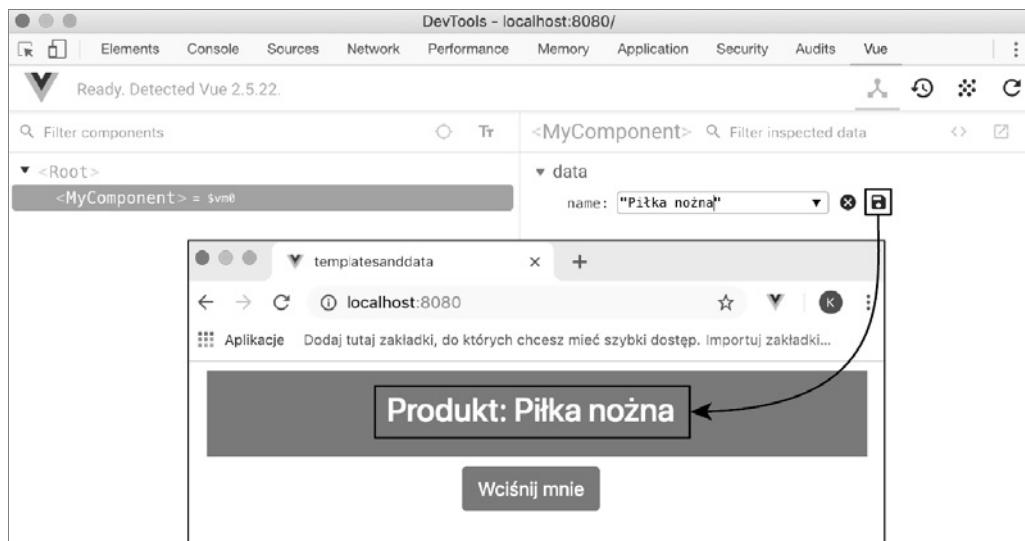


Rysunek 12.2. Budowa dyrektywy v-text

Dyrektyna jest stosowana za pomocą atrybutu `v-text`. Wartością atrybutu jest wyrażenie, które Vue.js ewaluuje, aby uzyskać treść do wyświetlenia użytkownikowi. Ten sam rodzaj wyrażenia jest używany w wiązaniu interpolacji tekstu (opisany w rozdziale 11.), a wynikiem tego wiązania jest wyświetlona wartość właściwości `data` o nazwie `name`.

W przeciwieństwie do wiązania interpolacji tekstu dyrektywa `v-text` całkowicie zmienia zawartość elementu, wobec którego została zastosowana — stąd obecność elementu `span` w szablonie w listingu 12.3.

Vue.js ewaluje wyrażenie w dyrektywie w momencie wykrycia zmiany. Aby zademonstrować ten mechanizm, uruchom narzędzia Vue Devtools i zmień wartość właściwości `name` (rysunek 12.3).



Rysunek 12.3. Wpływ zmiany wartości na dyrektywę

Wyświetlanie czystego kodu HTML

Wiązanie interpolacji tekstu i dyrektywa `v-text` automatycznie oczyszczają wyświetlana treść ze wszelkich znaków, które mogłyby wpływać na strukturę dokumentu HTML. W ten sposób automatycznie usuwają zagrożenia typu XSS (ang. *Cross-Site Scripting* — skrypty międzywitrynowe). Atak XSS wykorzystuje możliwość zinterpretowania tekstu jako kodu HTML przez przeglądarkę, dzięki czemu włamywacz może wstawić treść lub kod na stronę (więcej na temat ataku XSS dowiesz się na stronie https://pl.wikipedia.org/wiki/Cross-site_scripting). Przykład takiego działania znajdziesz w listingu 12.4, w którym dodaję właściwość data o treści zawierającej element `script`.

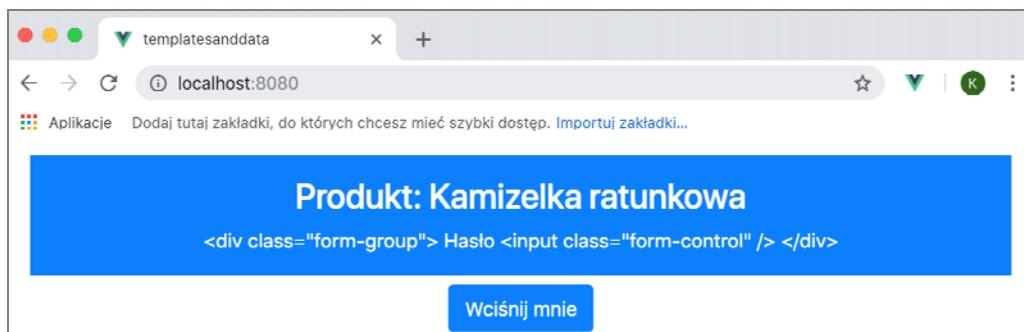
Listing 12.4. Dodawanie właściwości `danych` do pliku `src/App.vue`

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: <span v-text="name"></span></h3>
      <span v-text="fragment"></span>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        fragment: `<div class="form-group">
          Hasło
          <input class="form-control" />
        </div>`
      }
    },
    methods: {
      handleClick() {
        // nie rób nic
      }
    }
  }
</script>
```

Nowa właściwość `data` nosi nazwę `fragment`, a jej wartość stanowi zbiór elementów języka HTML, w skład którego to zbioru wchodzi element `input`. W szablonie komponentu dodaję element `span` i stosuję wiązanie `v-text`, które zamieni zawartość elementu na wartość właściwości `fragment`.

Po zapisaniu zmian aplikacja zostanie odświeżona, a potencjalnie groźny element kodu zostanie oczyszczony (rysunek 12.4).

Oczyszczenie wartości danych to dobry pomysł, który staje się niezbędny, gdy przetwarzasz dane pochodzące od użytkownika. Jednak niekiedy masz pewność, że pracujesz z bezpiecznymi danymi i możesz chcieć wyrenderować je jako kod HTML. Wówczas skorzystaj z dyrektywy `v-html` (listing 12.5).



Rysunek 12.4. Oczyszczenie wartości danych

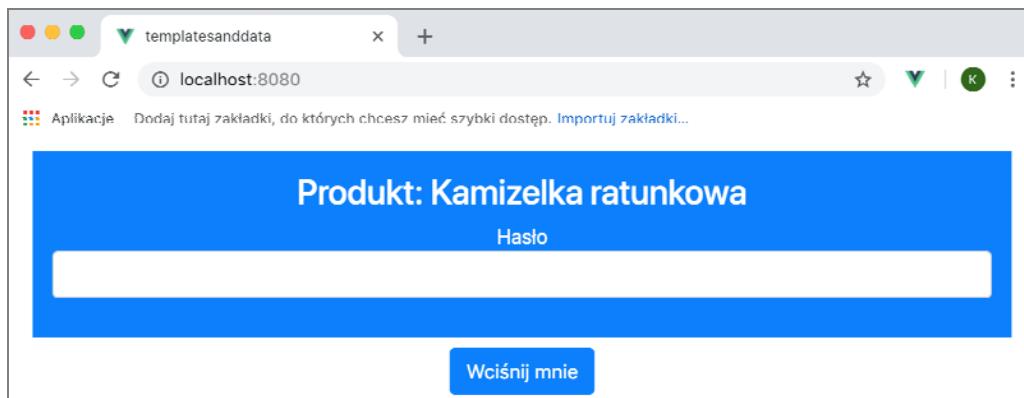
Listing 12.5. Wyświetlanie treści HTML w pliku src/App.vue

```
...
<template>
  <div class="bg-primary text-white text-center m-2 p-3">
    <h3 v-text="name" >Produkt:<span v-text="name"></span></h3>

    <span v-html="fragment"></span>
  </div>
  <button v-on:click="handleClick" class="btn btn-primary">
    Wciśnij mnie
  </button>
</template>
...
```

-
- **Ostrzeżenie** Nie korzystaj z tej dyrektywy, jeśli nie masz całkowitej pewności, że dane, które przetwarzasz, są bezpieczne.
-

Dyrektywa `v-html` jest stosowana w taki sam sposób jak `v-text`, ale wyświetla dane bez ich oczyszczenia (rysunek 12.5). W związku z tym przeglądarka po prostu wyrenderuje kod HTML i wyświetli go użytkownikowi za pomocą pola `input`.



Rysunek 12.5. Wyświetlenie wartości danych HTML

Wyświetlanie wybranych elementów

Nie wszystkie elementy wchodzące w skład komponentu muszą być wyświetlane jednocześnie — często fakt wyświetlania danego elementu zależy od stanu komponentu. Vue.js zawiera dyrektywy, które zmieniają widoczność elementów HTML na podstawie ewaluacji wyrażenia wiążania danych. W listingu 12.6 korzystam z dyrektywy `v-if`, aby kontrolować widoczność elementu HTML.

Listing 12.6. Wyświetlanie wybranych elementów w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: <span v-text="name"></span></h3>
      <h4 v-if="showElements">{{ price }}</h4>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        price: 275,
        showElements: true
      },
      methods: {
        handleClick() {
          this.showElements = !this.showElements;
        }
      }
    }
</script>
```

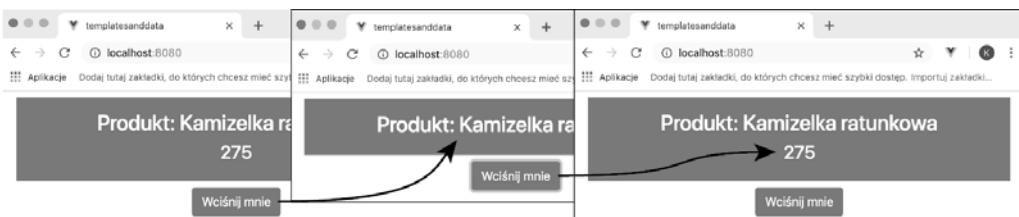
W tym przykładzie zastosowałem dyrektywę `v-if` do elementu `h4`:

```
...
<h4 v-if="showElements">{{ price }}</h4>
...
```

Dyrektyna wykoną wyrażenie i skorzysta z jego wartości, aby określić widoczność elementu `h4`. Element będzie widoczny, jeśli wynik wyrażenia będzie prawdziwy (ang. *truthy*, por. ramka „Prawdziwy a naprawdę prawdziwy”). Efektem tego jest wyświetlenie elementu `h4`, jeśli właściwość danych `showElements` ma wartość `true`, lub ukrycie tego elementu w przeciwnym razie.

Do metody `handleClick` dodałem instrukcję, która przełącza wartość `showElements` w momencie kliknięcia przycisku. W ten sposób demonstrujemy działanie dyrektywy `v-if` (rysunek 12.6).

- **Wskazówka** Aby określić widoczność, dyrektywa `v-if` usuwa i tworzy ponownie elementy i ich zawartość, jeśli elementy są różne, lub wykorzystuje ponownie istniejący element, jeśli typy są takie same. Oznacza to, że tylko widoczne elementy stanowią część drzewa DOM. Jeśli chcesz pozostawić element w drzewie, skorzystaj z dyrektywy `v-show` i zarządzaj jego widocznością za pomocą właściwości CSS.



Rysunek 12.6. Kontrola widoczności elementu

Prawdziwy a naprawdę prawdziwy

Dyrektywy takie jak `v-if` ewaluują wyrażenia, aby określić, czy mają do czynienia z wartościami prawdziwymi (ang. *truthy*, nie *true*), czy fałszywymi (ang. *falsy*, nie *false*), a nie z dosłowną prawdą i fałszem logicznym. Jest to sytuacja dość specyficzna i niekiedy powoduje problemy. Niniejsze wartości są zawsze fałszywe:

- wartość `false` (typu logicznego `boolean`),
- wartość `0` (liczba),
- pusty łańcuch znaków (" "),
- `null`,
- `undefined`,
- `NaN` (*not-a-number*, specjalna wartość liczbowa).

Wszystkie pozostałe wartości są prawdziwe — chociażby tekst o wartości "false". Aby unikać nieporozumień, najlepiej korzystać tylko z wartości logicznych (typu `boolean`) `true` i `false`.

Wyświetlanie wybranych elementów sąsiednich

Typowym przykładem użycia dyrektywy `v-if` jest zastosowanie jej do elementu nadzielnego poziomu, którego widoczność chcemy kontrolować. Takie podejście staje się jednak kłopotliwe, jeśli istnieje wiele elementów na tym samym poziomie, kontrolowanych przez to samo wyrażenie (listing 12.7).

Listing 12.7. Zastosowanie tej samej dyrektywy do elementów równorzędnych w pliku `src/App.vue`

```

...
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: <span v-text="name"></span></h3>
      <ul class="text-left">
        <li>Element listy</li>
        <li v-if="showElements">{{name}}</li>
        <li v-if="showElements">{{price}}</li>
        <li>Inny element listy </li>
      </ul>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
  
```

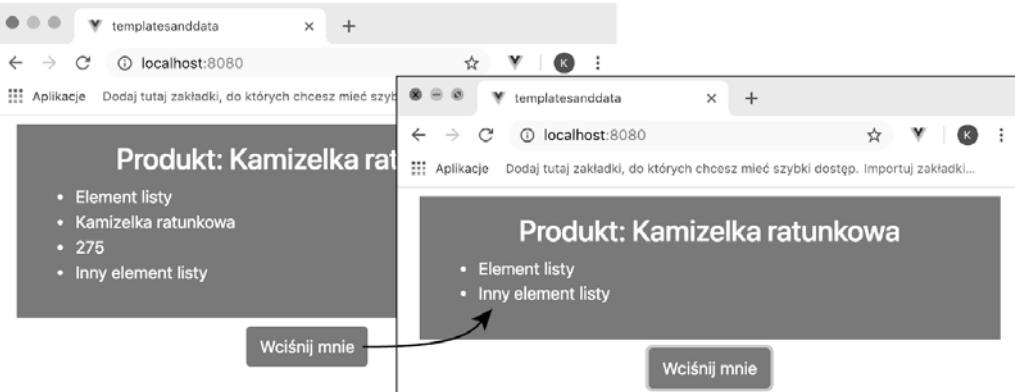
Moim celem jest kontrola widoczności dwóch elementów `li` należących do listy `ul` na podstawie tego samego wyrażenia wiążania danych. Duplikowanie dyrektyw nie jest jednak właściwym rozwiązaniem. W niektórych sytuacjach wystarczy dodać neutralny znacznik HTML jako rodzica dla takich elementów (np. `div` lub `span`), ale tutaj to podejście nie zda egzaminu, bo skutkowałoby powstaniem nieprawidłowego kodu HTML (elementy `ul` nie mogą zawierać elementów `div` lub `span`).

W takiej sytuacji rolą rodzica może pełnić element `template` (listing 12.8).

Listing 12.8. Zastosowanie elementu template w pliku src/App.vue

```
...
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3>Produkt: <span v-text="name"></span></h3>
      <ul class="text-left">
        <li>Element listy</li>
        <template v-if="showElements">
          <li>{{name}}</li>
          <li>{{price}}</li>
        </template>
        <li>Inny element listy</li>
      </ul>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
...
```

Dyrektywa jest stosowana wobec elementu `template`, który zostanie usunięty w czasie komplikacji, nie generując w ten sposób nieprawidłowego kodu HTML. Dzięki temu sąsiadujące elementy `li` są zarządzane za pomocą pojedynczej dyrektywy `v-if` (rysunek 12.7).



Rysunek 12.7. Zastosowanie elementu template do kontroli grupy elementów równorzędnych

- **Wskazówka** Dzięki dyrektywie `v-for` i właściwości obliczanej możesz osiągnąć podobny efekt w przypadku elementów, które nie są obok siebie. Dyrektywa `v-for` jest opisana w rozdziale 13.

Wybór fragmentów zawartości

Jeśli chcesz wyświetlać różne rodzaje treści na podstawie wartości danych, skorzystaj z dyrektywy `v-if` i zaneguj jedno z wyrażeń, jak w listingu 12.9.

Listing 12.9. Wybór treści do wyświetlenia w pliku src/App.vue

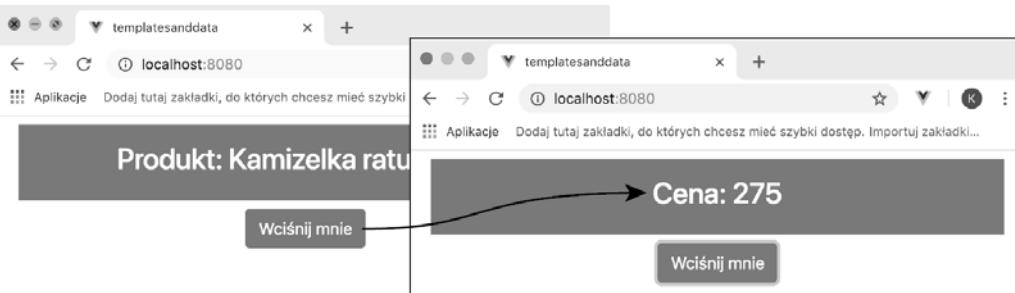
```
...
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-if="showElements">Produkt: {{name}}</h3>
      <h3 v-if="!showElements">Cena: {{price}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
...
```

To podejście działa, ale jest dość nietypowe. Poza tym, jeśli kryterium ulegnie zmianie, musisz zmodyfikować dwa wyrażenia. Sytuacja komplikuje się, gdy wyrażenie jest bardziej złożone. W związku z tym warto skorzystać ze specjalnej dyrektywy `v-else`, która działa wraz z dyrektywą `v-if` i nie wymaga własnego wyrażenia (listing 12.10).

Listing 12.10. Uproszczenie wyboru treści w pliku src/App.vue

```
...
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-if="showElements">Produkt: {{name}}</h3>
      <h3 v-else>Cena: {{price}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
...
```

Dyrekcja `v-else` jest stosowana zaraz po `v-if`, automatycznie zmieniając widoczność elementu przeciwnie do skojarzonej dyrektywy `v-if` (rysunek 12.8).



Rysunek 12.8. Wybór sekcji z treścią

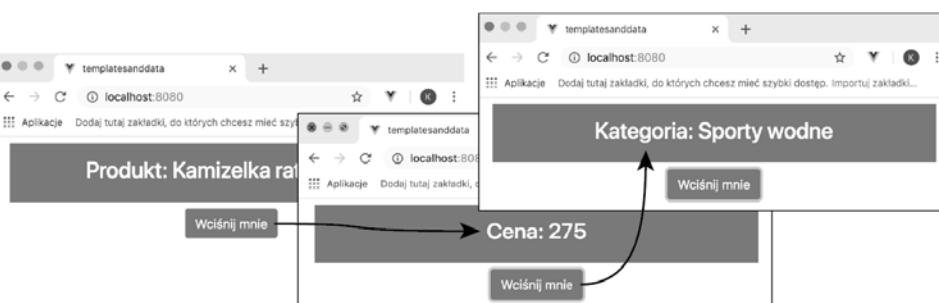
Dokonywanie bardziej skomplikowanych wyborów

Jeśli standardowe podejście `if/else` nie wystarcza, możesz skorzystać z dyrektywy `v-else-if`, stanowiącej połączenie dyrektyw `v-if` i `v-else`. Jeśli wynikiem dyrektywy `v-if` jest `false`, następuje sprawdzenie dyrektywy `v-else-if`, aby określić, czy elementy oznaczone tą dyrektywą powinny być wyświetcone. Dopiero na zakończenie jest sprawdzana wartość dyrektywy `v-else`. Skomplikowane wybory mogą być zarządzane za pomocą wielu dyrektyw `v-else-if` (listing 12.11).

Listing 12.11. Dokonywanie skomplikowanych wyborów w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-if="counter % 3 == 0">Produkt: {{name}}</h3>
      <h3 v-else-if="counter % 3 == 1">Cena: {{price}}</h3>
      <h3 v-else>Kategoria: {{category}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        price: 275,
        category: "Sporty wodne",
        counter: 0
      }
    },
    methods: {
      handleClick() {
        this.counter++;
      }
    }
  }
</script>
```

Wyrażenia dla dyrektyw `v-if` i `v-else-if` polegają na właściwości `counter`, której wartość jest inkrementowana w momencie kliknięcia przycisku. Oprócz tego w kodzie jest zawarta dyrektywa `v-else`, która wyświetli element, jeśli wyrażenia wcześniejszych dyrektyw przyjmą wartości `false` (rysunek 12.9).



Rysunek 12.9. Dokonywanie bardziej zaawansowanych wyborów

Wybór wyświetlanego elementów za pomocą stylów CSS

Dyrektywy `v-if`, `v-else` i `v-else-if` ukrywają elementy, usuwając je z drzewa modelu DOM, a następnie tworzą je ponownie. W związku z tym model DOM zawiera tylko elementy, które są widoczne dla użytkownika. Takie podejście może sprawiać problem w przypadku stosowania stylów CSS, które wybierają elementy na podstawie położenia w drzewie względem rodzica. Listing 12.12 przedstawia przykład tego rodzaju problemu.

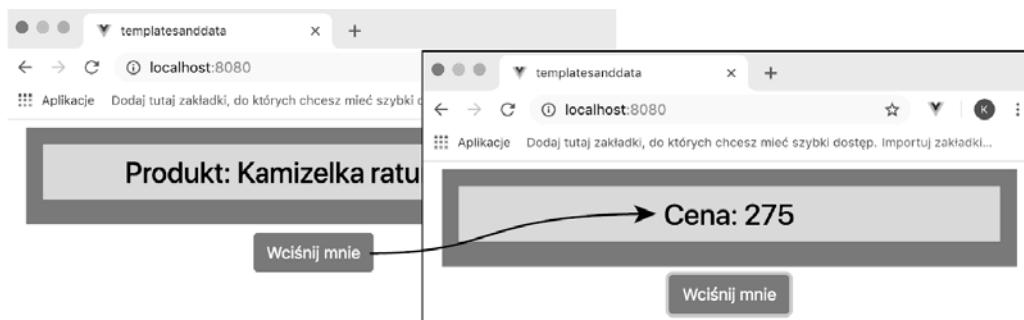
Listing 12.12. Dodawanie stylów CSS zależnych od położenia elementu w pliku `src/App.vue`

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-if="counter % 2 == 0">Produkt: {{name}}</h3>
      <h3 v-else>Cena: {{price}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        price: 275,
        counter: 0
      }
    },
    methods: {
      handleClick() {
        this.counter++;
      }
    }
  }
</script>
<style>
  h3:first-child { background-color: aquamarine; padding: 10px; color: black; }
</style>
```

Uprościłem szablon, zostawiając tylko dwa elementy nagłówkowe, do których zastosowałem dyrektywy `v-if` i `v-else`. Dodałem także element `style`, który zawiera styl z selektorem `h3:first-child`. Ten selektor dopasuje elementy typu `h3`, które są pierwszym dzieckiem swojego rodzica.

Problem zauważysz po zapisaniu zmian i kliknięciu przycisku *Wciśnij mnie*. Ta dyrektywa zapewni, że tylko jeden z elementów `h3` będzie widoczny. Niestety skoro niewidoczne elementy są usuwane z drzewa DOM, widocznym elementem jest pierwsze — i jedyne — dziecko rodzica, które zawsze zostaje wybrane przez selektor CSS (rysunek 12.10).

- **Wskazówka** Aby zauważyc zmiany wprowadzone w elemencie `style`, konieczne może być odświeżenie przeglądarki.



Rysunek 12.10. Elementy wybrane przez selektor CSS

Takie zachowanie staje się problemem, gdy naszym zamiarem jest jedynie podkreślenie nazwy produktu. Style nie powinny działać w odniesieniu do ceny, ponieważ jest ona drugim dzieckiem rodzica w szablonie, niemniej z uwagi na usuwanie niewidocznych elementów z drzewa DOM otrzymujemy nieprawidłowy rezultat.

W listingu 12.12 naprawiam ten problem, zmieniając selektor CSS. Nie zawsze jest to jednak możliwe, zwłaszcza w przypadku pracy z globalnymi stylami, stosowanymi w całej aplikacji lub w przypadku korzystania z frameworków zewnętrznych, takich jak Bootstrap. W takich sytuacjach rozsądne wyjście stanowi dyrektywa `v-show`, ponieważ funkcjonuje jak dyrektywa `v-if`, ale nie usuwa niewidocznych elementów z drzewa DOM. W listingu 12.13 stosuję dyrektywę `v-show` w komponencie.

Listing 12.13. Pozostawienie niewidocznych elementów w drzewie DOM w pliku src/App.vue

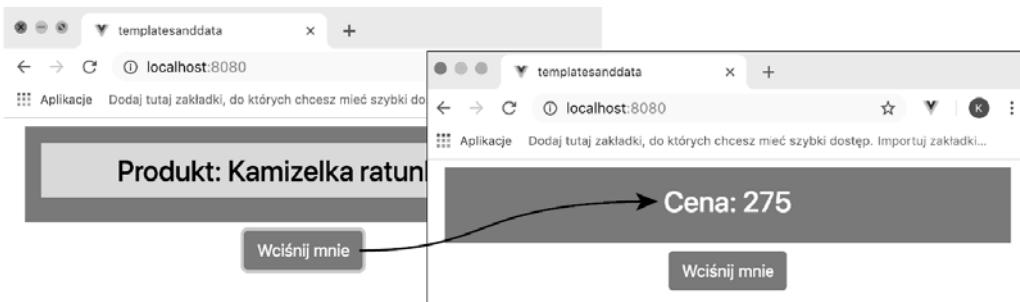
```
...
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-show="counter % 2 == 0">Produkt: {{name}}</h3>
      <h3 v-show="counter % 2 != 0">Cena: {{price}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
...
```

Dyrektywa `v-show` jest stosowana w taki sam sposób jak `v-if` i może być używana zamiennie. Po zapisaniu zmian i kliknięciu przycisku `Wciśnij mnie` licznik zostanie zwiększyony, a dyrektywa `v-show` będzie naprzemiennie (po każdym kliknięciu) pokazywać i ukrywać treść. Sposób działania dyrektywy możesz zweryfikować, analizując drzewo DOM za pomocą narzędzi przeglądarki — przekonasz się, że ukrycie elementu odbywa się za pomocą ustawienia właściwości `display` na `none`:

```
...
<h3 style="display: none;">Cena: 275</h3>
...
```

Skoro elementy są jedynie ukrywane — a nie usuwane — style komponentu zostaną zastosowane prawidłowo (rysunek 12.11).

Ustawienie właściwości `display` może być bardziej wydajne od usuwania elementów z drzewa DOM, ale dyrektywa `v-show` nie może być używana w elemencie `v-template`. Nie istnieje także odpowiednik dla dyrektyw `v-else-if` i `v-else`, przez co dyrektywę `v-show` musisz zastosować do obu elementów `h3`.



Rysunek 12.11. Ukrywanie elementów

Ustawianie atrybutów i właściwości elementu

Dyrektywa `v-bind` jest używana do ustawiania atrybutów lub właściwości elementu. Niniejszy podrozdział rozpoczyna się od atrybutów, a następnie objaśnię, czym różnią się one od właściwości. W listingu 12.14 korzystam z dyrektywy `v-bind`, aby przypisać elementom odpowiednie klasy framework'a Bootstrap. Takie zachowanie jest jednym z najczęstszych przypadków użycia dyrektywy `v-bind`.

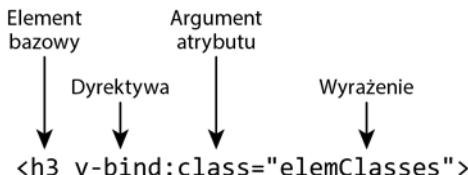
Listing 12.14. Przypisywanie klas elementom w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind:class="elemClasses">Produkt: {{name}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight: false
      }
    },
    computed: {
      elemClasses() {
        return this.highlight
          ? ["bg-light", "text-dark", "display-4"]
          : ["bg-dark", "text-light", "p-2"];
      }
    },
    methods: {
      handleClick() {
        this.highlight = !this.highlight;
      }
    }
  }
</script>
```

Ten przykład może wydawać się bardziej skomplikowany, niż jest w rzeczywistości. Zaczniemy od dyrektywy, którą stosuję do elementu h3 w następujący sposób:

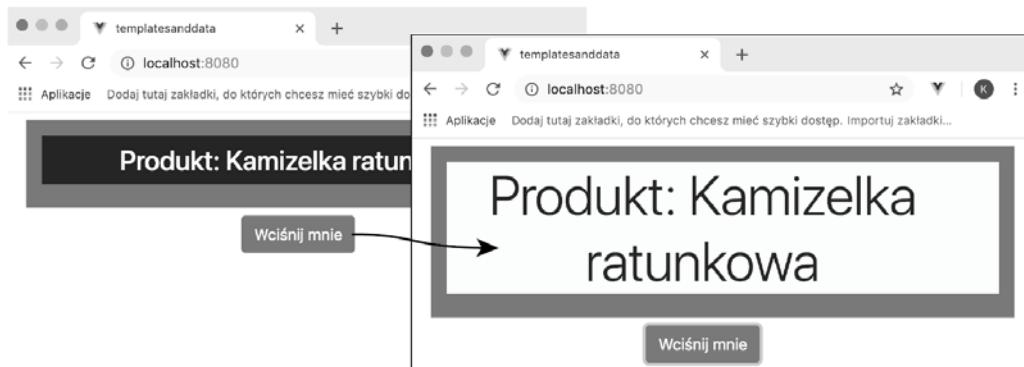
```
...
<h3 v-bind:class="elemClasses">Produkt: {{name}}</h3>
...
```

Dyrektywa v-bind otrzymuje zarówno argument, jak i wyrażenie (rysunek 12.12). Argument określa atrybut elementu, którego dyrektywa dotyczy, a wyrażenie dostarcza wartość tego atrybutu.



Rysunek 12.12. Struktura dyrektywy v-bind

W listingu 12.14 wyrażenie dyrektywy pobiera wartość właściwości obliczonej, która zwraca tablicę stylów. Zawartość tej tablicy jest ustalana na podstawie wartości właściwości danych o nazwie highlight. Takie podejście może wydawać się pośrednim ustwieniem stylów, jednak pokazuje ono elastyczny mechanizm, który pozwala Vue.js przedstawiać treść użytkownikom różnymi metodami. Aby obejrzeć efekt, zapisz zmiany w pliku App.vue i kliknij przycisk *Wciśnij mnie* — przełączysz w ten sposób wartość właściwości highlight (rysunek 12.13).



Rysunek 12.13. Ustawianie atrybutu klasy elementu

Kliknięcie przycisku zmienia przynależność bazowego elementu do dwóch grup klas. Jeśli właściwość highlight ma wartość true, bazowy element staje się członkiem klas bg-light, text-dark i display-4 (jasny kolor tła, ciemny kolor tekstu i duży rozmiar czcionki). W przeciwnym razie wybrane zostają klasy bg-dark, text-light i p-2 (ciemny kolor tła, jasny kolor tekstu i dodatkowe odstępy).

Dyrektywy w postaci skróconej

Dyrektywa v-bind przyjmuje dwie formy. Jedną z nich jest forma dłuższa, pokazana w poprzednim przykładzie, w której łączymy nazwę dyrektywy z nazwą atrybutu za pomocą dwukropka. Forma skrócona z kolei pomija nazwę dyrektywy, dzięki czemu zamiast zapisu v-bind:class możemy zastosować zapis :class. Poniższa konstrukcja:

```
...
<h3 v-bind:class="elemClasses">Produkt: {{name}}</h3>
...
```

może zostać zastąpiona następującą:

```
...
<h3 :class="elemClasses">Produkt: {{name}}</h3>
...
```

Wybór między formą długą a skróconą to kwestia osobistych preferencji i nie zmienia ona w żaden sposób zachowania dyrektywy.

Stosowanie obiektu do konfiguracji klas

Składnia tablicowa, zastosowana przeze mnie w poprzednim punkcie, staje się nieraz dość nietypowa, zwłaszcza, jeżeli zbiór klas elementu bazowego jest uzależniony od wielu różnych informacji. Dyrektywa `v-bind` może także skorzystać z obiektu, którego nazwy właściwości odpowiadają nazwom klas, a wartości tych właściwości określają przynależność do elementu bazowego (listing 12.15).

Listing 12.15. Zastosowanie obiektu do określenia przynależności do klas w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind:class="elemClasses" class="display-4">Produkt: {{name}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
    <button v-on:click="handleOtherClick" class="btn btn-primary">
      Lub mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight1: false,
        highlight2: false
      }
    },
    computed: {
      elemClasses() {
        return {
          "text-dark": this.highlight1,
          "bg-light": this.highlight2
        }
      }
    },
    methods: {
      handleClick() {
        this.highlight1 = !this.highlight1;
      }
    }
  }
</script>
```

```

        },
        handleOtherClick() {
            this.highlight2 = !this.highlight2;
        }
    }
}
</script>

```

Dodałem kolejny przycisk do kodu HTML, a do kodu JavaScript — metodę, która przełącza wartość właściwości `data`. Obliczona właściwość `elemClasses` zwraca następujący obiekt:

```

...
return {
    "text-dark": this.highlight1,
    "bg-light": this.highlight2
}
...

```

Nazwy właściwości obiektu określają przynależność elementu bazowego do klas `bg-light` i `text-dark` na podstawie wartości dwóch właściwości. Dyrektywa `v-bind` dodaje element bazowy do tych klas, których wartości właściwości przyjmują wartość `true`, a usuwa z tych, w których mamy do czynienia z wartością `false`.

- **Wskazówka** Nazwy właściwości w powyższym obiekcie umieszczam w cudzysłowach (w listingu 12.15). Nazwy klas Bootstrapa zawierają znak minus, co uniemożliwia zapisanie ich jako nazw właściwości bez cudzysłowów.

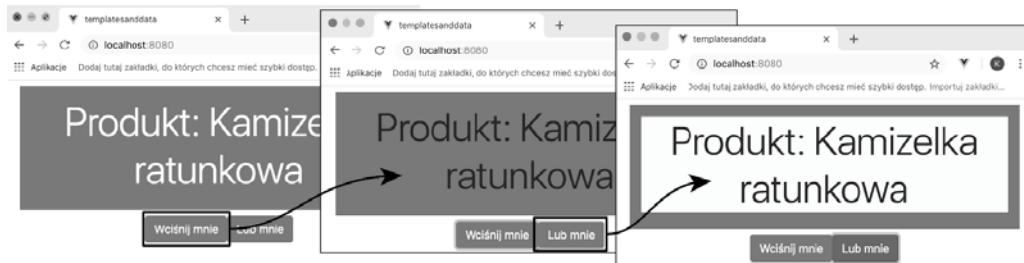
Oczywiście wciąż mogę korzystać z atrybutu `class` w szablonie w przypadku tych klas, do których element bazowy powinien należeć zawsze.

```

...
<h3 v-bind:class="elemClasses" class="display-4">Produkt: {{name}}</h3>
...

```

Z powyższego zapisu można wywnioskować, że element `h3` będzie zawsze należał do klasy `display-4`, podczas gdy przynależność do klas `bg-light` i `text-dark` będzie zależeć od stanu wartości przełączanych za pomocą przycisków (rysunek 12.14).



Rysunek 12.14. Zastosowanie obiektu w celu konfiguracji przynależności do klas

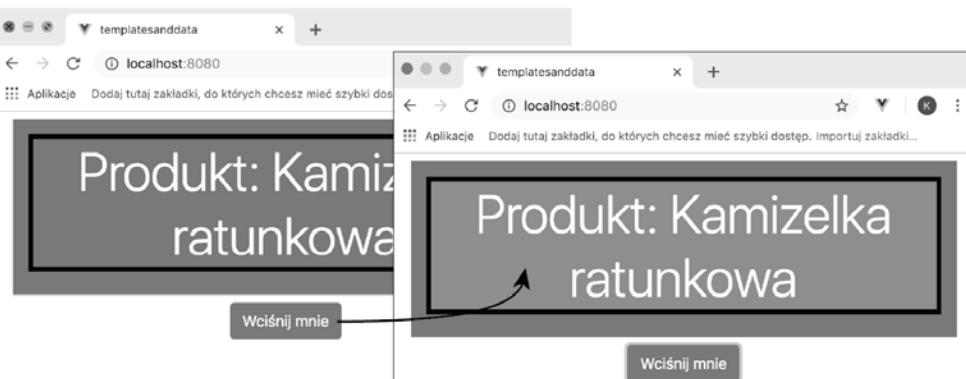
Ustawianie pojedynczych stylów

Atrybut `v-bind` udostępnia te same funkcje do zarządzania atrybutem `style` co w przypadku klas, dzięki czemu możemy zarządzać pojedynczymi właściwościami stylów CSS. W listingu 12.16 do zarządzania wartościami pojedynczych właściwości stylu stosuję dyrektywę.

Listing 12.16. Zarządzanie atrybutami stylu w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind:style="elemStyles" class="display-4">Produkt: {{name}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight: false,
      }
    },
    computed: {
      elemStyles() {
        return {
          "border": "5px solid red",
          "background-color": this.highlight ? "coral": ""
        }
      },
    },
    methods: {
      handleClick() {
        this.highlight = !this.highlight;
      }
    }
  }
</script>
```

Właściwość obliczona elemStyles zwraca obiekt, którego nazwy właściwości odpowiadają nazwom właściwości CSS. Właściwość border otrzymuje wartość stałą, ale już właściwość background-color jest określana dynamicznie na podstawie wartości właściwości highlighted, zmienianej w momencie kliknięcia przycisku. Zmiana właściwości danych wpływa na kolor tła elementu h3, co widać na rysunku 12.15.

**Rysunek 12.15.** Ustawianie pojedynczych właściwości stylu

Ustawianie innych atrybutów

Dyrektywa `v-bind` może być używana do ustawienia dowolnego atrybutu, jednak bez dodatkowego wsparcia, które otrzymujemy w przypadku właściwości `class` i `style`, pozwalającego na użycie obiektów i tablic. W listingu 12.17 korzystam z dyrektywy `v-bind` do ustawienia wartości własnego atrybutu, który dopasowuje selektory zdefiniowane w elemencie `style`.

Listing 12.17. Ustawianie własnego atrybutu w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind:data-size="size" class="display-4">Produkt: {{name}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight: false,
      }
    },
    computed: {
      size() {
        return this.highlight ? "big" : "small";
      }
    },
    methods: {
      handleClick() {
        this.highlight = !this.highlight;
      }
    }
  }
</script>
<style>
  [data-size=big] { font-size: 40pt; }
  [data-size=small] { font-size: 20pt; }
</style>
```

Specyfikacja HTML pozwala na wprowadzanie własnych atrybutów, których nazwy zaczynają się od `data-`. Nie musisz używać prefiksu `data`, jeśli stosujesz Vue.js, ale jest to konwencja, z której korzystam, gdy muszę odróżnić atrybuty charakterystyczne dla mojej aplikacji.

W tym przykładzie do zarządzania wartością własnego atrybutu `data-size` używam dyrektywy `v-bind`. Jej wartość jest pobierana z właściwości obliczonej `size`, która zwraca wartość `big` lub `small`. Wartości są powiązane z selektorami w elemencie `style`, który zmienia rozmiar czcionki, co daje efekt jak na rysunku 12.16.

-
- **Wskazówka** Jeśli nie widzisz zmian po zapisaniu listingu 12.17, odśwież przeglądarkę.
-



Rysunek 12.16. Ustawianie własnego atrybutu

Ustawianie wielu atrybutów

Pojedyncza dyrektywa `v-bind` potrafi ustawić wiele atrybutów jednocześnie. Użycie dyrektywy w elemencie bazowym jest wykonywane bez użycia argumentu. Zamiast tego musimy przekazać wyrażenie, które wygeneruje specjalny obiekt. Właściwości tego obiektu muszą reprezentować nazwy atrybutów, które chcemy ustalić, jak w listingu 12.18.

Listing 12.18. Ustawianie wielu atrybutów w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind="attrValues">Produkt: {{name}}</h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight: false,
      }
    },
    computed: {
      attrValues() {
        return {
          class: this.highlight ? ["bg-light", "text-dark"] : [],
          style: {
            border: this.highlight ? "5px solid red": "",
          },
          "data-size": this.highlight ? "big" : "small"
        }
      }
    },
    methods: {
      handleClick() {
        this.highlight = !this.highlight;
      }
    }
  }
</script>
```

```
</script>
<style>
  [data-size=big] { font-size: 40pt; }
  [data-size=small] { font-size: 20pt; }
</style>
```

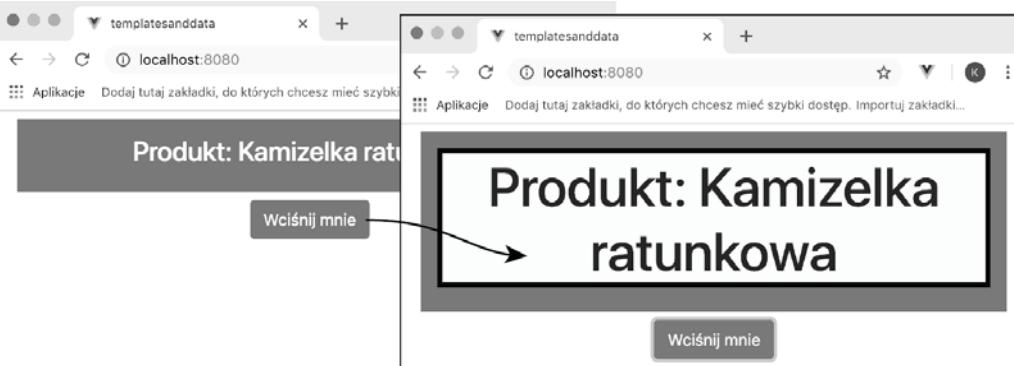
Obiekt zwrócony za pomocą właściwości obliczanej attrValues definiuje właściwości class, style i data-size. Wartości te są określane za pomocą wartości właściwości danych o nazwie highlight. Właściwość ta jest przełączana w wyniku kliknięcia przycisku. Gdy wartość właściwości highlight wynosi false, element h3 jest ustawiany następująco:

```
...
<h3 class="" style="" data-size="small">Produkt: Kamizelka ratunkowa</h3>
...
```

Gdy wartością właściwości highlight jest true, dyrektywa v-bind zmienia element w taki sposób:

```
...
<h3 class="bg-light text-dark" style="border: 5px solid red;" data-size="big">
  Produkt: Kamizelka ratunkowa
</h3>
...
```

Pojedyncze wiązanie zarządza wieloma atrybutami, dając efekt jak na rysunku 12.17.



Rysunek 12.17. Zarządzanie wieloma atrybutami za pomocą pojedynczego wiązania

Ustawianie właściwości HTMLElement

Domyślnie dyrektywa v-bind ustawia atrybuty elementu bazowego, co pokazałem w poprzednim przykładzie. Istnieje możliwość ustawienia dyrektywy v-bind tak, aby ustawić właściwość obiektu w drzewie DOM, który to obiekt reprezentuje dany element.

Gdy przeglądarka przetwarza dokument HTML, tworzony jest obiektowy model dokumentu (ang. *Document Object Model — DOM*). W tym modelu są zawarte obiekty, z których każdy reprezentuje jeden element HTML. Obiekty zawierają właściwości, które nie są związane z atrybutami obsługiwanyimi przez język HTML. Wynika to z faktu, że za pomocą tych właściwości dostarczane są specjalne funkcje lub po prostu z różnicami pomiędzy specyfikacjami HTML i modelem DOM, które historycznie nie zawsze były dobrze zarządzane.

Nie będziesz zapewne korzystać z tej funkcji w każdym projekcie, ale jeśli znajdziesz się w sytuacji, w której ustawienie atrybutu nie wystarczy, skorzystaj z modyfikatora prop dyrektywy v-bind, jak w listingu 12.19.

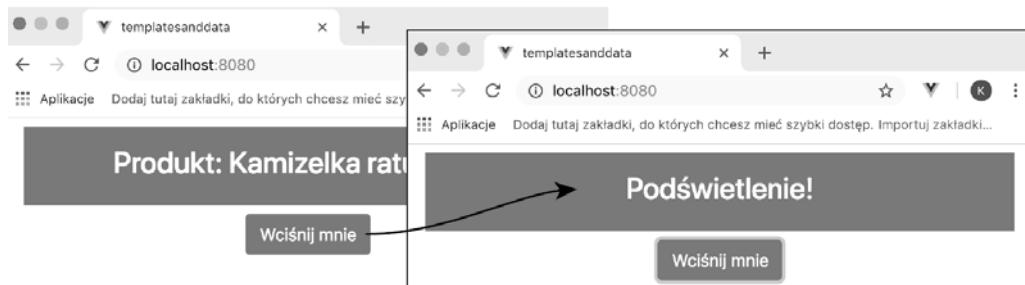
Listing 12.19. Ustawianie właściwości elementu w pliku src/App.vue

```
<template>
  <div class="container-fluid text-center">
    <div class="bg-primary text-white m-2 p-3">
      <h3 v-bind:text-content.prop="textContent"></h3>
    </div>
    <button v-on:click="handleClick" class="btn btn-primary">
      Wciśnij mnie
    </button>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa",
        highlight: false,
      }
    },
    computed: {
      textContent() {
        return this.highlight ? "Podświetlenie!" : `Produkt: ${this.name}`;
      }
    },
    methods: {
      handleClick() {
        this.highlight = !this.highlight;
      }
    }
  }
</script>
```

Modyfikator pojawia się po nazwie właściwości, w miejscu zastosowania dyrektywy do elementu, oddzielony za pomocą kropki:

```
...
<h3 v-bind:text-content.prop="textContent">Product: {{name}}</h3>
...
```

W ten sposób instruujemy dyrektywę v-bind, aby zarządzała wartością właściwości obiektu o nazwie text-content. Właściwość text-content udostępnia zawartość tekstową elementu, a w tym przykładzie ustawiamy zawartość elementu h3, jak widać na rysunku 12.18.

**Rysunek 12.18.** Ustawianie właściwości elementu

Odkryj właściwości elementów

Fundacja Mozilla na stronie developer.mozilla.org/en-US/docs/Web/API/Element udostępnia przydatny wykaz wszystkich obiektów reprezentujących model DOM. Dla każdego elementu Mozilla przedstawia podsumowanie i przeznaczenie dostępnych właściwości. Zaczni od obiektu `HTMLElement` (developer.mozilla.org/en-US/docs/Web/API/HTMLElement), który udostępnia funkcje wspólne dla wszystkich elementów. Następnie możesz zająć się bardziej szczegółowymi elementami, takimi jak `HTMLInputElement`, reprezentującym elementy typu `input`.

Podsumowanie

W tym rozdziale opisałem niektóre z wbudowanych dyrektyw Vue.js, użytecznych w obsłudze elementów HTML. Pokazałem, jak zarządzać treścią elementu za pomocą dyrektyw `v-text` i `v-html`, jak wybiorczo wyświetlić treść przy użyciu dyrektyw `v-if` i `v-show`, a także jak ustawić atrybuty elementu za pomocą dyrektywy `v-bind`. W kolejnym rozdziale zajmiemy się dyrektywą, która jest używana do powtarzania treści dla każdego elementu z tablicy.

ROZDZIAŁ 13.



Obsługa dyrektywy Repeater

W tym rozdziale kontynuuję omawianie wbudowanych dyrektyw Vue.js, skupiając się na dyrektywie `v-for`, która jest używana do generowania list, a także wierszy tabel i układów bazujących na siatce. Tabela 13.1 umiejscawia dyrektywę `v-for` w szerszym kontekście.

Tabela 13.1. Umiejscowienie dyrektywy `v-for` w szerszym kontekście

Pytanie	Odpowiedź
Czym jest dyrektywa <code>v-for</code> ?	Dyrektywa <code>v-for</code> jest używana do duplikowania zbioru elementów HTML dla każdego elementu tablicy lub dla każdej właściwości w danym obiekcie.
Dlaczego jest użyteczna?	Dyrektywa <code>v-for</code> definiuje zmienną, która daje dostęp do przetwarzanego obiektu. Taki obiekt może być używany w wiązaniach danych w celu przedstawienia jego treści lub dostosowania powtarzonych fragmentów kodu HTML.
Jak się z niej korzysta?	Dyrektywa <code>v-for</code> jest stosowana do nadzielnego elementu, który ma zostać poddany replikacji. Określone w tym elemencie wyrażenie wskazuje na źródło obiektów, podczas gdy za pomocą podanej nazwy zmiennej można odnosić się w wiązaniach danych.
Czy są jakieś pułapki lub ograniczenia?	Dyrektywa <code>v-for</code> nie obsługuje specyficznych kolekcji języka JavaScript, takich jak Set czy Map. Warto zwrócić uwagę na kolejność, w jakiej następuje wyliczenie właściwości w obiekcie.
Czy są jakieś rozwiązania alternatywne?	Zawsze możesz utworzyć własną dyrektywę, co opisuję w rozdziale 26. Z drugiej strony dyrektywa <code>v-for</code> zawiera szereg optymalizacji, które pozwalają na wydajne przetwarzanie dużych zbiorów danych.

Tabela 13.2 podsumowuje ten rozdział.

Przygotowania do tego rozdziału

W tym rozdziale kontynuujemy pracę z projektem `templatesanddata` z rozdziału 12. Aby przygotować się do tego rozdziału, uproszcilem komponent główny aplikacji (korzeń), co widać w listingu 13.1.

Tabela 13.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Powtórz ten sam zbiór elementów dla każdego obiektu w tablicy lub każdej właściwości dostępnej w obiekcie.	Skorzystaj z dyrektywy v-for.	13.3, 13.13, 13.15 – 13.16
Odwołaj się do bieżącego obiektu w duplikowanym zbiorze elementów.	Skorzystaj z funkcji alias dyrektywy v-for.	13.4
Skójarz element HTML z wybranym obiektem.	Zdefiniuj atrybut key za pomocą dyrektywy v-bind.	13.5 – 13.7
Odwołaj się do położenia bieżącego obiektu w tablicy.	Skorzystaj z funkcji indeksu dyrektywy v-for.	13.8
Upewnij się, że zmiany w indeksie tablicy są wykrywane.	Skorzystaj z metody Vue.set.	13.9 – 13.11
Powtórz element bez źródła danych.	Skorzystaj z wartości liczbowej zamiast ze źródła danych w wyrażeniu dyrektywy v-for.	13.14

Listing 13.1. Uproszczenie zawartości pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3>Produkt: {{ name }}</h3>
    </div>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    },
    methods: {
      handleClick() {
        // nie rób nic
      }
    }
  }
</script>
```

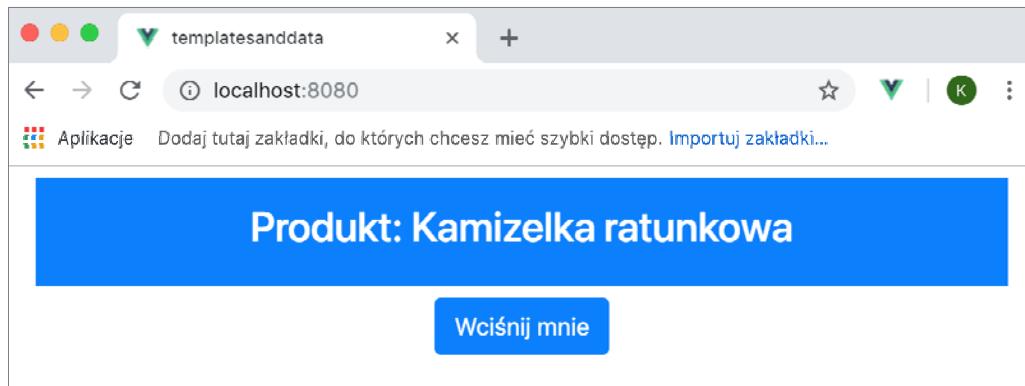
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Zapisz zmiany w pliku *App.vue* i wykonaj polecenie z listingu 13.2 w katalogu *templatesanddata*, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 13.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, a zobaczysz efekt jak na rysunku 13.1.



Rysunek 13.1. Uruchomienie przykładowej aplikacji

Przeglądanie tablicy

Większość aplikacji obsługuje tablice powiązanych obiektów (np. tego samego typu), które w pewnej formie należy przedstawić użytkownikowi, zazwyczaj jako wiersze tabeli lub w układzie siatki (ang. *grid layout*). Dyrektywa *v-for* jest używana do powtarzania zbiorów elementów HTML dla każdego obiektu w tablicy. W listingu 13.3 korzystam z dyrektywy *v-for* do wyliczenia (ang. *enumeration*) tablicy obiektów w celu wypełnienia tabeli danymi.

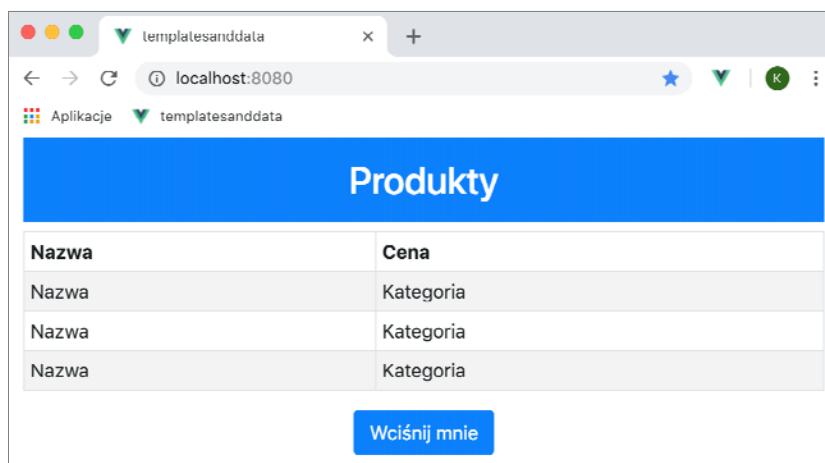
Listing 13.3. Wyliczanie tablicy w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered table-striped text-left">
      <tr><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="p in products">
          <td>Nazwa</td>
          <td>Kategoria</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
```

```
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        products: [
          { name: "Kajak", price: 275 },
          { name: "Kamizelka ratunkowa", price: 48.95 },
          { name: "Piłka nożna", price: 19.50 }
        ]
      },
      methods: {
        handleClick() {
          // nie rób nic
        }
      }
    }
  }
</script>
```

- **Uwaga** W tym podrozdziale możesz zobaczyć ostrzeżenia lintera. Będą one wynikać z wprowadzenia funkcji dostarczonych przez dyrektywę v-for.

Wartość właściwości data o nazwie products stanowi tablicę obiektów. Dyrektywa v-for przetwarza każdy z nich, generując treść widoczną na rysunku 13.2. Nie jest to co prawda zbyt użyteczny przykład, ale dyrektywa v-for może być niejasna, przez co chcę najpierw przedstawić jej czytelne objaśnienie.



Rysunek 13.2. Wyliczenie obiektów w celu utworzenia wierszy tabeli

Dyrektywę v-for w szablonie zastosowałem w następujący sposób:

```
...
<tr v-for="p in products">
  <td>Nazwa</td>
  <td>Kategoria</td>
</tr>
...
```

Wyrażenie w dyrektywie `v-for` musi przyjąć określona formę: `<alias> in <zródło>`. Pojęcie „źródło” odwołuje się do źródła obiektów, które ma być przetworzone. W tym przypadku odwołujemy się do właściwości danych o nazwie `products`. Słowo kluczowe `in` oddziela tablicę od aliasu stanowiącego tymczasową zmienną, do której jest przypisywany każdy obiekt w miarę przetwarzania źródła danych.

W tym przypadku aliasem jest zmienna `p`, a źródłem — właściwość `products`. Zastosowanie dyrektywy spowoduje wygenerowanie elementu `tr` zawierającego dwa elementy `td` dla każdego z obiektów, które są zawarte w tablicy `products`:

```
...
<tbody>
  <tr><td>Nazwa</td><td>Kategoria</td></tr>
  <tr><td>Nazwa</td> <td>Kategoria</td></tr>
  <tr><td>Nazwa</td> <td>Kategoria</td></tr>
</tbody>
...
```

Tablica `products` zawiera trzy obiekty, dlatego dyrektywa `v-for` powtarza elementy `tr` i `td` trzy razy.

Stosowanie aliasu

W poprzednim przykładzie powtórzyliśmy treść dla wszystkich obiektów znajdujących się w źródle danych. Większość aplikacji musi pokazywać treść dostosowaną do każdego obiektu i tu właśnie dużą rolę odgrywa alias. Każdy przetwarzany obiekt z tablicy źródłowej trafia w którymś momencie do tej zmiennej. Z aliasu możemy skorzystać w wiązaniach danych w powtarzanym fragmencie kodu (listing 13.4).

Listing 13.4. Zastosowanie aliasu w dyrektywie `v-for` w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered table-striped text-left">
      <tr><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="p in products">
          <td>{{ p.name }}</td>
          <td>{{ p.price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        products: [
          { name: "Kajak", price: 275 },
          { name: "Kamizelka ratunkowa", price: 48.95 },
          { name: "Piłka nożna", price: 19.50 }
        ],
      },
    },
  }
</script>
```

```

filters: {
    currency(value) {
        return new Intl.NumberFormat("pl-PL",
            { style: "currency", currency: "PLN", }).format(value);
    },
},
methods: {
    handleClick() {
        // nie rób nic
    }
}
}
</script>

```

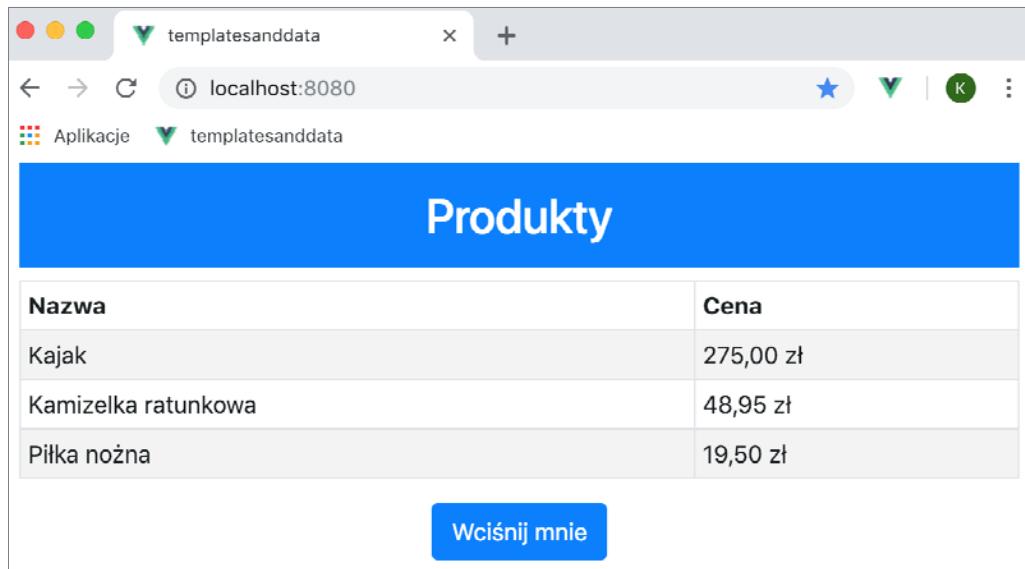
Nazwą zmiennej aliasu w naszym przykładzie jest `p`, dlatego skorzystałem z wiązania interpolacji tekstu, aby wyświetlić nazwę (`name`) i cenę (`price`) każdego obiektu, odwołując się właśnie do tej zmiennej — wyrażenia przyjęły postać `p.name` i `p.price`.

```

...
<tr v-for="p in products">
    <td>{{ p.name }}</td>
    <td>{{ p.price | currency }}</td>
</tr>
...

```

Wszystkie funkcje interpolacji tekstu, które opisałem w rozdziale 11., mogą być używane z aliasem `v-for`. Dodalem do naszego kodu znany z poprzednich rozdziałów filtr `currency` i użyłem go w celu sformatowania ceny, generując efekt jak na rysunku 13.3.



Rysunek 13.3. Zastosowanie aliasu w dyrektywie v-for

To właśnie zdolność zastosowania obiektu umiejscowionego pod zmienną aliasu w wiązaniach danych czyni dyrektywę `v-for` użyteczną. W ten sposób możemy wygenerować różną treść dla każdego obiektu.

Określanie klucza

Jeśli przejdziesz teraz do konsoli narzędzi deweloperskich — czy to w wierszu poleceń, czy też w przeglądarce, zobaczyesz, że linter zgłasza ostrzeżenie.

```
...
error: Elements in iteration expect to have 'v-bind:key' directives (vue/require-v-for-key)
at src/App.vue:7:17:
5 |         <tr><th>Nazwa</th><th>Cena</th></tr>
6 |
7 |         <tbody>
|             <tr v-for="p in products">
|                 ^
|                     <td>{{ p.name }}</td>
|                     <td>{{ p.price | currency }}</td>
10 |                 </tr>
...
...
```

To ostrzeżenie odnosi się do sposobu, w jaki dyrektywa `v-for` obsługuje zmianę kolejności przetwarzanych obiektów. Domyślnie dyrektywa `v-for` reaguje na zmiany kolejności, aktualizując treść wyświetlana przez każdy z utworzonych obiektów. W tym przykładzie oznacza to przejęcie do każdego utworzonego elementu `tr` i aktualizację tekstu zawartego w elementach `td`.

Zobaczmy, jak wygląda proces wprowadzania zmian. W listingu 13.5, w metodzie `handleClick`, dodałem instrukcję, która usuwa pierwszy element z tablicy i wstawia go na końcu. Dodałem także element `style` z wdrożonym stylem, który wybiera element o ID `tagged` i ustawia jego kolor tła.

Listing 13.5. Dodawanie stylu do pliku src/App.vue

```
...
<script>
export default {
    name: "MyComponent",
    data: function () {
        return {
            products: [
                { name: "Kajak", price: 275 },
                { name: "Kamizelka ratunkowa", price: 48.95 },
                { name: "Piłka nożna", price: 19.50 }
            ],
            filters: {
                currency(value) {
                    return new Intl.NumberFormat("pl-PL",
                        { style: "currency", currency: "PLN", }).format(value);
                },
            },
            methods: {
                handleClick() {
                    this.products.push(this.products.shift());
                }
            }
        }
    }
</script>
<style>
    #tagged { background-color: coral; }
</style>
...
...
```

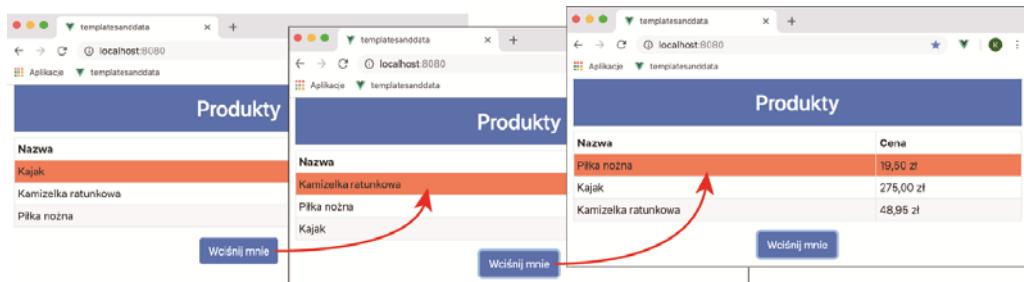
Zapisz zmiany i otwórz narzędzia deweloperskie F12 po odświeżeniu aplikacji. Następnie przejdź do panelu *Console* i wykonaj instrukcję z listingu 13.6.

Listing 13.6. Oznaczanie elementu

```
document.querySelector("tbody > tr").id = "tagged"
```

Ta instrukcja korzysta z API modelu DOM, aby wybrać pierwszy element *tr*, będący dzieckiem elementu *tbody*. Następnie ten element otrzymuje ID *tagged*. Tuż po wykonaniu polecenia pierwszy wiersz tabeli zostanie wyświetlony z innym kolorem tła.

Kliknij przycisk *Wciśnij mnie*, aby przekonać się, w jaki sposób dyrektywa *v-for* radzi sobie domyślnie ze zmianą kolejności obiektów. Przy każdym wywołaniu metody *handleClick* kolejność obiektów w tablicy ulega zmianie, a dyrektywa *v-for* aktualizuje zawartość utworzonych przez nią elementów (rysunek 13.4).



Rysunek 13.4. Aktualizacja zawartości elementów wykonywana w miejscu

Dyrektyna *v-for* musi zaktualizować wszystkie utworzone elementy, ponieważ nie wiadomo, jak można powiązać je z obiektami, skoro tablica została zmieniona.

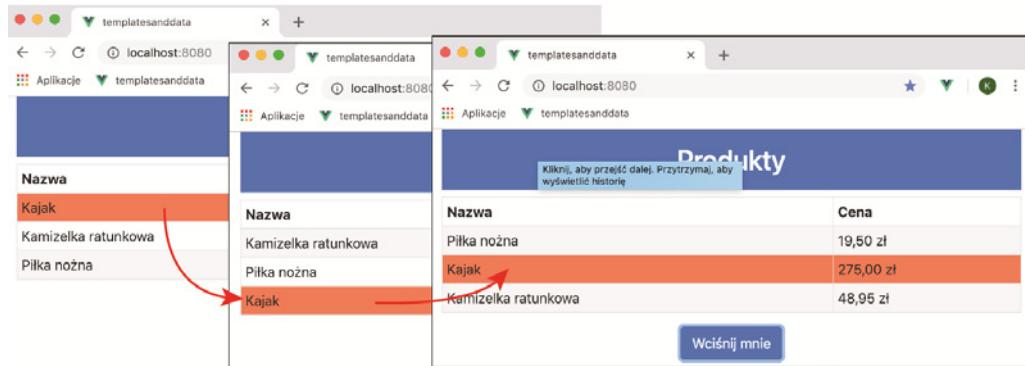
Zastosowanie dyrektywy, która powiąże każdy obiekt z odpowiadającym mu elementem, wymaga określenia właściwości pełniącej rolę unikatowego klucza, używanego w dyrektywie *v-bind* (listing 13.7).

Listing 13.7. Wybór klucza w pliku src/App.vue

```
...
<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered table-striped text-left">
      <tr><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="p in products" v-bind:key="p.name">
          <td>{{ p.name }}</td>
          <td>{{ p.price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
...
```

Dyrektywa `v-bind` jest używana do ustawienia atrybutu `key`, którego wartość jest wyrażona za pomocą aliasu zdefiniowanego w wyrażeniu dyrektywy `v-for`. W tym przypadku alias to `p`, a ja chcę skorzystać z właściwości `name` jako klucza dla obiektów danych, dlatego podałem nazwę `p.name` w wyrażeniu `v-bind`.

Zapisz zmiany w komponencie, odśwież aplikację i wykonaj instrukcje z listingu 13.6 raz jeszcze, aby oznaczyć pierwszy element tr w ciele tabeli. Po zmianie koloru tła kliknij przycisk, aby zmienić kolejność obiektów w tablicy `products`. Zwróć uwagę, że teraz dyrektywa `v-for` „wie”, jak obiekty są powiązane z elementami, i jest w stanie zareagować na zmiany w kolejności elementów tablicy (rysunek 13.5).



Rysunek 13.5. Zmiana kolejności elementów stanowi reakcję na zmianę stanu

- **Wskazówka** Możesz wyłączyć regułę lintera, która wymaga podania klucza, jeśli domyślnie zachowanie Ci odpowiada — jest ono szybsze dla niewielkiej liczby obiektów (por. listing 13.14).

Pobieranie indeksu elementu

Dyrektywa `v-for` obsługuje dodatkową zmienną, do której jest przypisywany indeks aktualnie przetwarzanego obiektu w tablicy. Taki indeks może posłużyć do różnych celów — wyświetlania numeru wiersza w tablicy lub oznaczania elementów stylami. W listingu 13.8 dodaję do tabeli nową kolumnę, która wyświetla indeks wiersza, a także stosuję dyrektywę `v-bind`, aby ustawić atrybut na wierszu tabeli, do którego stosuję styl.

Listing 13.8. Zastosowanie indeksu w dyrektywie v-for w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered text-left">
      <tr><th>Indeks</th><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="(p, i) in products"
          v-bind:key="p.name" v-bind:odd="i % 2 == 0">
          <td>{{ i + 1 }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
```

```

        </div>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                products: [
                    { name: "Kajak", price: 275 },
                    { name: "Kamizelka ratunkowa", price: 48.95 },
                    { name: "Piłka nożna", price: 19.50 },
                    { name: "Chorągiewki narożne", price: 39.95 },
                    { name: "Stadion", price: 79500 },
                    { name: "Myśląca czapeczka", price: 16 }
                ]
            }
        },
        filters: {
            currency(value) {
                return new Intl.NumberFormat("pl-PL",
                    { style: "currency", currency: "PLN" }).format(value);
            },
        },
        methods: {
            handleClick() {
                this.products.push(this.products.shift());
            }
        }
    }
</script>
<style>
    [odd]{ background-color: lightblue; }
</style>

```

Aby skorzystać z indeksu, musimy wprowadzić dodatkową zmienną przy definiowaniu wyrażenia `v-for`, np.:

```

...
<tr v-for="(p, i) in products" v-bind:key="p.name" v-bind:odd="i % 2 == 0">
...

```

Zmienna indeksu jest oddzielona od aliasu przecinkiem, a obie zmienne są otoczone nawiasami. W trakcie przetwarzania tablicy następuje przypisanie obiektu do zmiennej `p`, a położenia w tablicy — do zmiennej `i` (zaczynając od 0). Zmienna indeksu może być używana w wiązaniach danych, tak jak każda inna wartość danych. W tym listingu korzystam z wiązania interpolacji tekstu, aby ustawić zawartość nowej kolumny. Wyrażenie jest skonstruowane tak, aby użytkownik widział sekwencję liczb zaczynającą się od 1, a nie od 0:

```

...
<td>{{ i + 1 }}</td>
...

```

Z indeksu korzystam także, aby ustawić własny atrybut o nazwie `odd` na wszystkich elementach `tr` o parzystych indeksach (parzystość stwierdzam za pomocą operatora modulo):

```

...
<tr v-for="(p, i) in products" v-bind:key="p.name" v-bind:odd="i % 2 == 0">
...

```

W efekcie parzyste wiersze otrzymują klasę `odd`, która dopasowuje selektor w elemencie `style`, co pozwala uzyskać efekt w postaci pokolorowania tabeli w paski (rysunek 13.6). W listingu 13.8 dodałem więcej elementów, aby lepiej przedstawić powstały efekt.

Indeks	Nazwa	Cena
1	Kajak	275,00 zł
2	Kamizelka ratunkowa	48,95 zł
3	Piłka nożna	19,50 zł
4	Chorągiewki narożne	39,95 zł
5	Stadion	79 500,00 zł
6	Myśląca czapeczka	16,00 zł

Rysunek 13.6. Zastosowanie indeksu w dyrektywie `v-for`

Rozszerzona postać dyrektywy `v-for`

Może wydawać się dziwne, że jestem w stanie skorzystać ze zmiennej indeksu w listingu 13.8 w ramach dyrektywy `v-bind`, która jest stosowana w tym samym elemencie co `v-for`. Takie podejście działa, ponieważ korzystam ze zwięzlej postaci dyrektywy `v-for`, w której jest ona stosowana względem najwyższego w hierarchii elementu duplikowanego dla każdego obiektu. Jest to równoważne z zastosowaniem elementu `template`:

```
...
<tbody>
    <template v-for="(p, i) in products" >
        <tr v-bind:key="p.name" v-bind:odd="i % 2 == 0">
            <td>{{ i + 1 }}</td>
            <td>{{ p.name }}</td>
            <td>{{ p.price | currency }}</td>
        </tr>
    </template>
</tbody>
...
```

Taki kod jest równoważny z dyrektywą zastosowaną w listingu 13.8, ale w tym przypadku lepiej widać, że dyrektywa i jej zawartość wchodzą z sobą w interakcję. Nie musisz określać elementu `template` w powiązaniu z dyrektywą `v-for`, o ile nie musisz powielać wielu równorzędnych elementów dla każdego obiektu danych. Jeśli korzystasz z elementu `template`, pamiętaj o zastosowaniu atrybutu `key` dyrektywy `v-bind` wobec powielanych elementów — a nie szablonu!

Stosowanie stylów CSS do naprzemiennego kolorowania wierszy tabeli

Naprzemienne kolory w tabeli czy też w układzie siatki to częste wymaganie, ale można je spełnić jeszcze prościej niż w przypadku zastosowania indeksu dyrektywy v-for. Pozwala na to większość frameworków CSS, w tym Bootstrap, przy czym efekt ten można osiągnąć nawet z zastosowaniem własnych stylów:

```
...
<style>
    tbody > tr:nth-child(even) { background-color: coral; }
    tbody > tr:nth-child(odd) { background-color: lightblue; }
</style>
...
```

Selektory tych stylów dopasowują nieparzyste i parzyste elementy tr, będące dziećmi elementu tbody. Najważniejszymi elementami stylu są :nth-child(odd) i :nth-child(even), które mogą być używane do wybierania dowolnych nieparzystych i parzystych elementów.

Wykrywanie zmian w tablicy

Vue.js wykrywa zmiany w tablicy dokonane za pomocą metod: push, pop, shift, unshift, splice, sort i reverse. Metody te zmieniają treść tablicy, dzięki czemu Vue.js jest w stanie śledzić obiekt tablicy i obserwować zmiany. To właśnie zmian tego rodzaju dokonałem w listingu 13.8, aby przedstawić jego najważniejszy aspekt:

```
...
handleClick() {
    this.products.push(this.products.shift());
}
...
```

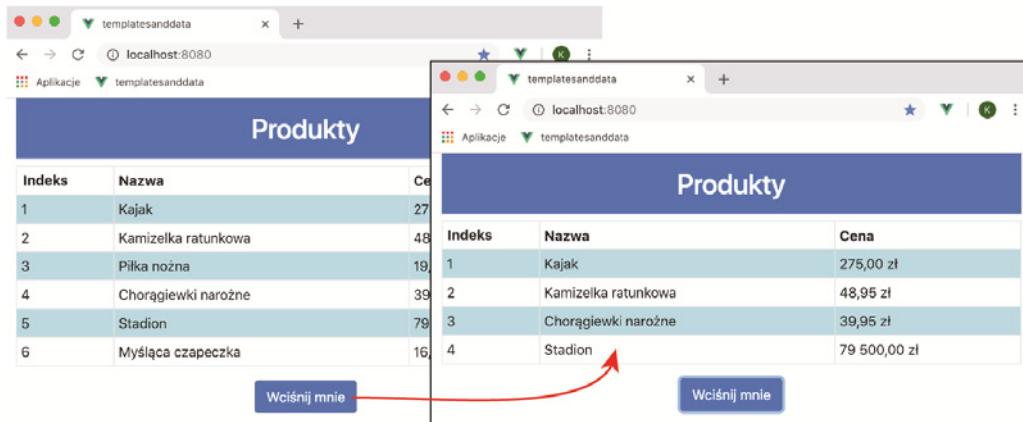
Skorzystałem z metody shift, aby usunąć element z tablicy, a następnie dodałem go ponownie za pomocą metody push. Choć zawartość tablicy się zmieniła, referencja do obiektu tablicy, dostępna we właściwości danych products, pozostała bez zmian.

Niektoře operacje powodują powstanie nowego obiektu tablicy. Do takich operacji należą metody filter i slice. W związku z tym do właściwości data zostaje przypisany nowy obiekt tablicy (listing 13.9).

Listing 13.9. Tworzenie nowej tablicy w pliku src/App.vue

```
...
handleClick() {
    this.products = this.products.filter(p => p.price > 20);
}
...
```

Zmieniłem instrukcję w metodzie handleClick na taką, w której korzystam z metody filter, w celu utworzenia nowej tablicy, zawierającej tylko ceny wyższe niż 20. Następnie przypisałem nową tablicę do właściwości products. Na szczęście jest to operacja, na którą Vue.js jest przygotowane. Jeśli niektóre elementy starej i nowej tablicy powtarzają się, istniejące treści zostaną użyte ponownie w celu zwiększenia wydajności. Niezależnie od ilości ponownie użytych treści zamiana tablicy ze starej na nową spowoduje odświeżenie przedstawianej treści (rysunek 13.7).



Rysunek 13.7. Przypisanie nowej tablicy

Problemy z aktualizacją treści

Istnieją dwojakiego rodzaju zmiany w zakresie tablicy, z którymi Vue.js sobie nie poradzi. Pierwsza z nich dotyczy sytuacji, gdy zamieniany jest element tablicy (listing 13.10).

Listing 13.10. Zamiana elementu tablicy w pliku src/App.vue

```
...
handleClick() {
    this.products[1] = { name: "Buty do biegania", price: 100 };
}
...
```

Metoda korzysta z indeksu tablicy, aby przypisać nowy obiekt pod indeksem o numerze 1 (drugi element). Vue.js nie wykryje takiej zmiany. Vue.js nie wykryje też zmian we właściwościach nowego obiektu.

Aby poradzić sobie z tym problemem, musimy skorzystać z metody, którą Vue.js udostępnia w celu zamiany obiektu w tablicy. Metoda przed użyciem musi być zaimportowana z modułu vue (listing 13.11).

Listing 13.11. Bezpieczna zamiana elementu tablicy w pliku src/App.vue

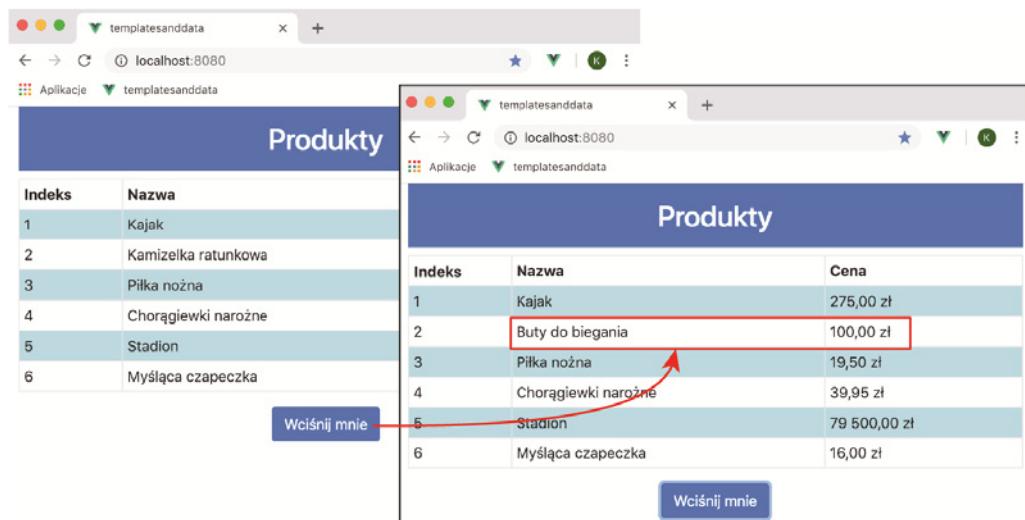
```
...
<script>
import Vue from "vue";
export default {
    name: "MyComponent",
    data: function () {
        return {
            products: [
                { name: "Kajak", price: 275 },
                { name: "Kamizelka ratunkowa", price: 48.95 },
                { name: "Piłka nożna", price: 19.50 },
                { name: "Chorągiewki narożne", price: 39.95 },
                { name: "Stadion", price: 79500 },
                { name: "Mysiąca czapeczka", price: 16 }
            ]
        },
        filters: {
            currency(value) {
                return new Intl.NumberFormat("pl-PL",
                    { style: "currency", currency: "PLN", }).format(value);
            }
        }
    }
},
```

```

        },
    },
    methods: {
        handleClick() {
            Vue.set(this.products, 1, { name: "Buty do biegania", price: 100 });
        }
    }
</script>
...

```

Metoda `Vue.set` przyjmuje trzy argumenty: tablicę do zmodyfikowania, indeks obiektu do zamiany i nowy obiekt. Odśwież aplikację i wyzwól proces wykrycia zmian — przekonasz się, że treść widoczna dla użytkownika zostanie zaktualizowana (rysunek 13.8).



Rysunek 13.8. Bezpieczna zamiana obiektu w tablicy

- **Wskazówka** Metoda `Vue.set` jest również dostępna w komponentie pod nazwą `this.$set`. Wolę tę pierwszą postać, ponieważ nie wszystkie tablice są obsługiwane w komponentach, a preferuję spójną metodę aktualizacji danych w całej aplikacji.

Drugą ważną zmianą, której Vue.js nie jest w stanie wykryć, jest skrócenie tablicy za pomocą zmiany właściwości `length`. Ten problem można rozwiązać przez użycie metody `splice` w celu pozbycia się niechcianych elementów tablicy.

Wyliczanie właściwości obiektu

Choć najczęstszym przypadkiem użycia dyrektywy `v-for` jest wyliczenie elementów tablicy, istnieje możliwość wyliczenia właściwości obiektu, co jest znacznie bardziej użyteczne, niż mogłoby się na początku wydawać. W listingu 13.12 zamieniam tablicę obiektów na pojedynczy obiekt, którego nazwy i wartości właściwości są mi potrzebne.

Listing 13.12. Zastosowanie obiektu do wyliczenia w pliku src/App.vue

```

<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered text-left">
      <tr><th>Indeks</th><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="(p, key, i) in products" v-bind:key="p.name">
          <td>{{ i + 1 }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
<script>
  import Vue from "vue";
  export default {
    name: "MyComponent",
    data: function () {
      return {
        products: [
          1: { name: "Kajak", price: 275 },
          2: { name: "Kamizelka ratunkowa", price: 48.95 },
          3: { name: "Piłka nożna", price: 19.50 },
          4: { name: "Chorągiewki narożne", price: 39.95 }
        ]
      }
    },
    filters: {
      currency(value) {
        return new Intl.NumberFormat("pl-PL",
          { style: "currency", currency: "PLN", }).format(value);
      },
    },
    methods: {
      handleClick() {
        Vue.set(this.products, 5, { name: "Buty do biegania", price: 100});
      }
    }
  }
</script>

```

W tym listingu zmieniłem zawartość właściwości `products`, przez co zwraca ona obiekt, a nie tablicę. Obiekt zawiera właściwości, których wartościami są również obiekty (o właściwościach `name` i `price`). Wyliczenie właściwości obiektu `products` osiągamy w następujący sposób:

```

...
<tr v-for="(p, key, i) in products" v-bind:key="p.name">
...

```

W momencie przetwarzania obiektu dyrektywa definiuje alias, nową zmienną zawierającą klucz i zmienną indeksu. Vue.js wykrywa, gdy właściwość ulega zmianie, ale nie jest w stanie wykryć dodania nowej właściwości — stąd użycie metody `Vue.set` w metodzie `handleClick`:

```
...
Vue.set(this.products, 5, { name: "Buty do biegania", price: 100});
...
```

Dyrektywa `v-for` wyliczy właściwości obiektu i zduplikuje treść dla każdej z nich (rysunek 13.9).

Indeks	Nazwa	Cena
1	Kajak	275,00 zł
2	Kamizelka ratunkowa	48,95 zł
3	Piłka nożna	19,50 zł
4	Chorągiewki narożne	39,95 zł

Wciśnij mnie

Rysunek 13.9. Wyliczanie właściwości obiektu

Właściwości obiektu a kwestia kolejności

Właściwości są wyliczane w kolejności zwróconej przez metodę `Object.keys`, która z reguły działa następująco:

- Jeżeli klucze stanowią wartości liczbowe (w tym wartości, które da się skonwertować na liczby), klucze są porządkowane rosnąco według wartości.
- Jeżeli klucze stanowią wartości tekstowe, są one uporządkowane w kolejności dodawania.
- Wszystkie inne klucze są uporządkowane w kolejności dodawania.

■ **Ostrzeżenie** Przedstawiony powyżej porządek jest najczęściej spotykany, ale mogą też pojawić się w nim różnice, wynikające z różnic w implementacjach języka JavaScript. Skorzystaj z właściwości `Object.keys`, aby posortować obiekty i tym samym zapewnić spójność. Omawiam to zagadnienie w dalszej części rozdziału.

Aby zademonstrować przykład porządkowania właściwości, zmieniam klucze w obiekcie `products` i dodaję kolumnę do tabeli w celu wyświetlenia wartości klucza (listing 13.13).

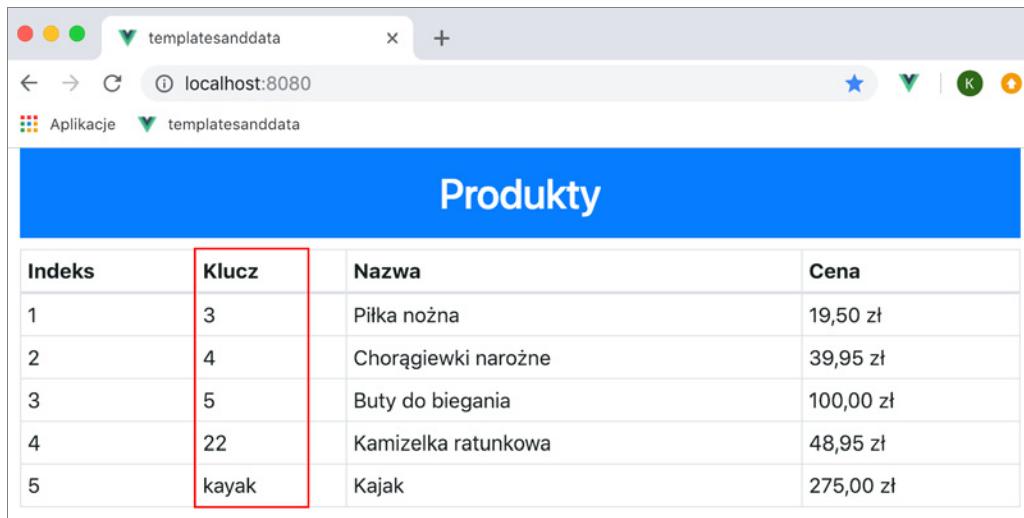
Listing 13.13. Obsługa kluczów obiektów w pliku src/App.vue

```

<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered text-left">
      <tr><th>Indeks</th><th>Klucz</th><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="(p, key, i) in products" v-bind:key="p.name">
          <td>{{ i + 1 }}</td>
          <td>{{ key }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button v-on:click="handleClick" class="btn btn-primary">
        Wciśnij mnie
      </button>
    </div>
  </div>
</template>
<script>
  import Vue from "vue";
  export default {
    name: "MyComponent",
    data: function () {
      return {
        products: [
          "kayak": { name: "Kajak", price: 275 },
          22: { name: "Kamizelka ratunkowa", price: 48.95 },
          3: { name: "Piłka nożna", price: 19.50 },
          "4": { name: "Chorągiewki narożne", price: 39.95 }
        ]
      }
    },
    filters: {
      currency(value) {
        return new Intl.NumberFormat("pl-PL",
          { style: "currency", currency: "PLN", }).format(value);
      },
    },
    methods: {
      handleClick() {
        Vue.set(this.products, 5, { name: "Buty do biegania", price: 100 });
      }
    }
  }
</script>

```

Gdy dyrektywa `v-for` wylicza właściwości obiektu `products`, zostaną one przetworzone w następującej kolejności: 3, 4, 22, *kayak*. Po kliknięciu przycisku i dodaniu nowej właściwości do obiektu, z kluczem o wartości 5, nowy wiersz tabeli zostanie uwzględniony pomiędzy kluczami 4 i 22 (rysunek 13.10).



Indeks	Klucz	Nazwa	Cena
1	3	Piłka nożna	19,50 zł
2	4	Chorągiewki narożne	39,95 zł
3	5	Buty do biegania	100,00 zł
4	22	Kamizelka ratunkowa	48,95 zł
5	kayak	Kajak	275,00 zł

Rysunek 13.10. Efekt uporządkowania właściwości obiektu

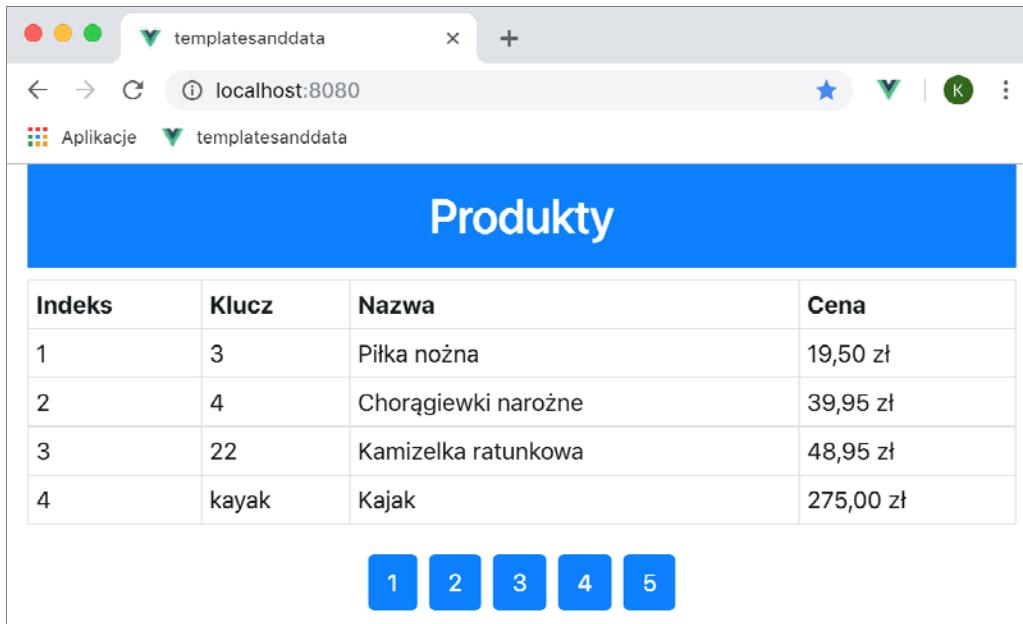
Powtarzanie elementów HTML bez źródła danych

Dyrektywa `v-for` może być używana do powtarzania elementów HTML określona liczbę razy, bez potrzeby zastosowania źródła danych takiego jak tablica czy obiekt. Ta funkcja jest przydatna w tworzeniu treści, która nie jest związana z określonym elementem (np. zestaw przycisków do stronicowania, jak w listingu 13.14).

Listing 13.14. Powtarzanie treści w pliku `src/App.vue`

```
...
<template>
  <div class="container-fluid">
    <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
    <table class="table table-sm table-bordered text-left">
      <tr><th>Indeks</th><th>Klucz</th><th>Nazwa</th><th>Cena</th></tr>
      <tbody>
        <tr v-for="(p, key, i) in products" v-bind:key="p.name">
          <td>{{ i + 1 }}</td>
          <td>{{ key }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p. price | currency }}</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <!-- eslint-disable-next-line vue/require-v-for-key -->
      <button v-for="i in 5" v-on:click="handleClick(i)"
              class="btn btn-primary m-1">
        {{ i }}
      </button>
    </div>
  </div>
</template>
...
```

Gdy źródłem wyrażenia v-for jest liczba całkowita, dyrektywa powtórzy elementy HTML określona liczbę razy, a do aliasu zostanie przypisana aktualna wartość iteratora. W tym listingu podałem liczbę 5, a z aliasu skorzystałem do wyświetlenia treści. Pierwszą wartością przypisaną do aliasu jest 1, co daje efekt jak na rysunku 13.11.



Rysunek 13.11. Powtarzanie elementów bez źródła danych

- **Wskazówka** Zwróć uwagę, że wyłącznie regułą lintera, która wymaga klucza do identyfikacji. Skoro sekwencja wartości jest generowana przez dyrektywę, nie trzeba martwić się kwestią zmiany kolejności elementów tablicy.

Stosowanie właściwości obliczanych z dyrektywą v-for

Wszystkie dotychczasowe przykłady bazowały na użyciu właściwości data, jednak dyrektywa v-for zadziała także z metodami i właściwościami obliczanymi. Dzięki temu możliwe jest użycie języka JavaScript do filtrowania lub sortowania powielonych obiektów. W kolejnych rozdziałach przedstawię inne metody, za pomocą których możesz zarządzać danymi przetwarzanymi przy użyciu dyrektywy v-for.

Stronicowanie danych

Większość aplikacji musi przedstawiać użytkownikowi jedynie fragment wszystkich dostępnych danych — z reguły korzystając z mechanizmu stronicowania. Połączenie właściwości obliczanej z funkcją powtarzania bez źródła danych może być łatwo zaimplementowane przez stronicowanie (listing 13.15).

Listing 13.15. Stronicowanie danych w pliku src/App.vue

```

<template>
    <div class="container-fluid">
        <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
        <table class="table table-sm table-bordered text-left">
            <tr><th>Nazwa</th><th>Cena</th></tr>
            <tbody>
                <tr v-for="p in pageItems" v-bind:key="p.name">
                    <td>{{ p.name }}</td>
                    <td>{{ p.price | currency }}</td>
                </tr>
            </tbody>
        </table>
        <div class="text-center">
            <!-- eslint-disable-next-line vue/require-v-for-key -->
            <button v-for="i in pageCount" v-on:click="selectPage(i)"
                    class="btn btn-secondary m-1"
                    v-bind:class="{ 'bg-primary': currentPage == i }">
                {{ i }}
            </button>
        </div>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                pageSize: 3,
                currentPage: 1,
                products: [
                    { name: "Kajak", price: 275 },
                    { name: "Kamizelka ratunkowa", price: 48.95 },
                    { name: "Piłka nożna", price: 19.50 },
                    { name: "Chorągiewki narożne", price: 39.95 },
                    { name: "Stadion", price: 79500 },
                    { name: "Myśląca czapeczka", price: 16 },
                    { name: "Chwiejne krzesło", price: 29.95 },
                    { name: "Szachownica", price: 75 },
                    { name: "Król(u) złoty", price: 1200 }
                ]
            }
        },
        computed: {
            pageCount() {
                return Math.ceil(this.products.length / this.pageSize);
            },
            pageItems() {
                let start = (this.currentPage - 1) * this.pageSize;
                return this.products.slice(start, start + this.pageSize);
            }
        },
        filters: {
            currency(value) {
                return new Intl.NumberFormat("pl-PL",
                    { style: "currency", currency: "PLN", }).format(value);
            },
        }
    }
</script>

```

```

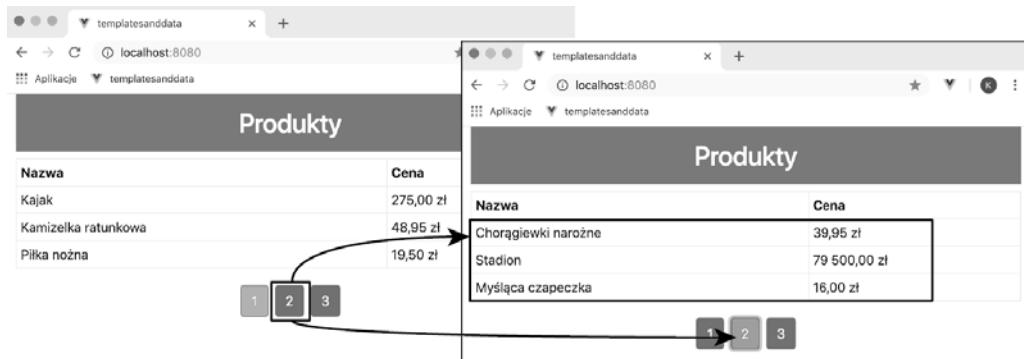
        },
        methods: {
            selectPage(page) {
                this.currentPage = page;
            }
        }
    }
</script>

```

- **Wskazówka** Bardziej skomplikowany przykład paginacji znajdziesz w aplikacji *Sklep sportowy* opisanej w pierwszej części tej książki.

Powróciłem do stosowania tablicy, a także dodałem kilka obiektów, dzięki czemu łatwiej będzie testować ten mechanizm. Zostały wprowadzone dwie właściwości danych: `pageSize`, która określa liczbę elementów na stronie, a także `currentPage`, która przechowuje aktualny numer strony. Pojawiły się także dwie nowe właściwości obliczone — `pageCount`, określająca liczbę wymaganych przez aplikację stron, i `pageItems`, w której znajdują się aktualnie wyświetlane na stronie elementy.

Aby wygenerować przyciski stronicowania, korzystam z właściwości `pageItems` w wyrażeniu dyrektywy `v-for`, dzięki czemu mogę powtórzyć element `button` tyle razy, ile stron jest dostępnych. Podkreślam także przycisk, który reprezentuje aktualnie przeglądającą stronę. W tym celu stosuję dyrektywę `v-bind`, aby przypisać klasę do elementu na podstawie wartości aliasu `v-for`. Na zakończenie stosuję dyrektywę `v-on`, opisaną w rozdziale 14., aby wywołać metodę `selectPage` po kliknięciu przycisku. W ten sposób zmieniam wartość właściwości `currentPage`, pozwalając użytkownikowi na przechodzenie pomiędzy stronami (rysunek 13.12).



Rysunek 13.12. Stronicowanie danych

Filtrowanie i sortowanie danych

Właściwości obliczane mogą być także używane do filtrowania i sortowania danych, jeszcze zanim te trafią do dyrektywy `v-for`. W listingu 13.16 dodaję element `select`, którego wartości są używane do zmiany danych wyświetlanych użytkownikowi.

Listing 13.16. Filtrowanie i sortowanie danych w pliku src/App.vue

```

<template>
    <div class="container-fluid">
        <h2 class="bg-primary text-white text-center p-3">Produkty</h2>
        <table class="table table-sm table-bordered text-left">
            <tr><th>Nazwa</th><th>Cena</th></tr>

```

```

<tbody>
    <tr v-for="p in pageItems" v-bind:key="p.name">
        <td>{{ p.name }}</td>
        <td>{{ p.price | currency }}</td>
    </tr>
</tbody>
</table>
<div class="text-center">
    <button class="btn btn-secondary m-1" v-on:click="toggleSort"
           v-bind:class="{ 'bg-primary': sort }">
        Włącz sortowanie
    </button>
    <button class="btn btn-secondary m-1" v-on:click="toggleFilter"
           v-bind:class="{ 'bg-primary': filter }">
        Włącz filtrowanie
    </button>
    <!-- eslint-disable-next-line vue/require-v-for-key -->
    <button v-for="i in pageCount" v-on:click="selectPage(i)"
            class="btn btn-secondary m-1"
            v-bind:class="{ 'bg-primary': currentPage == i }">
        {{ i }}
    </button>
</div>
</div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                pageSize: 3,
                currentPage: 1,
                filter: false,
                sort: false,
                products: [
                    { name: "Kajak", price: 275 },
                    { name: "Kamizelka ratunkowa", price: 48.95 },
                    { name: "Piłka nożna", price: 19.50 },
                    { name: "Chorągiewki narożne", price: 39.95 },
                    { name: "Stadion", price: 79500 },
                    { name: "Myśląca czapeczka", price: 16 },
                    { name: "Chwiejne krzesło", price: 29.95 },
                    { name: "Szachownica", price: 75 },
                    { name: "Król(u) złoty", price: 1200 }
                ]
            }
        },
        computed: {
            pageCount() {
                return Math.ceil(this.dataItems.length / this.pageSize);
            },
            pageItems() {
                let start = (this.currentPage - 1) * this.pageSize;
                return this.dataItems.slice(start, start + this.pageSize);
            },
            dataItems() {
                let data = this.filter
                ? this.products.filter(p => p.price > 100) : this.products;

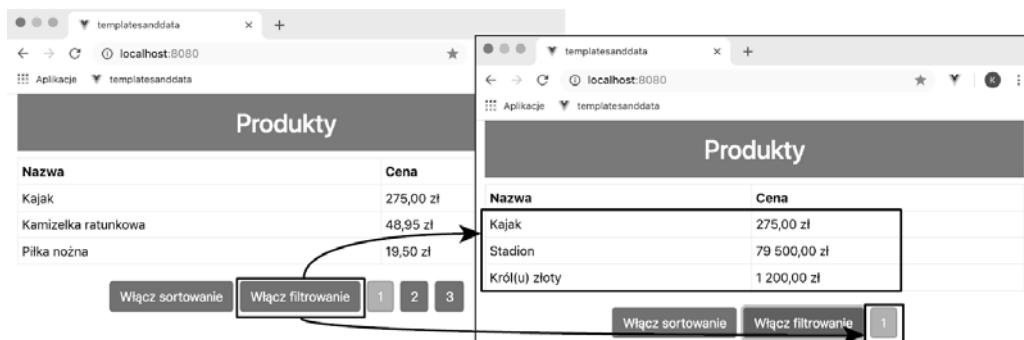
```

```

        return this.sort
            ? data.concat().sort((p1, p2) => p2.price - p1.price) : data;
    }
},
filters: {
    currency(value) {
        return new Intl.NumberFormat("pl-PL",
            { style: "currency", currency: "PLN", }).format(value);
    },
},
methods: {
    selectPage(page) {
        this.currentPage = page;
    },
    toggleFilter() {
        this.filter = !this.filter
        this.currentPage = 1;
    },
    toggleSort() {
        this.sort = !this.sort;
        this.currentPage = 1;
    }
}
}
</script>

```

W tym przykładzie korzystamy z nowej właściwości obliczanej `dataItems`, aby przygotować dane do wyświetlenia na podstawie właściwości `filter` i `sort`. Wartości właściwości danych są włączane przez kliknięcie nowych przycisków, a liczba przycisków stronicowania jest aktualizowana na podstawie wyborów użytkownika (rysunek 13.13).



Rysunek 13.13. Stronicowanie i sortowanie danych

Podsumowanie

W tym rozdziale objaśniłem zastosowanie dyrektywy `v-for` i pokazałem, jak można za jej pomocą przeglądać elementy tablic, właściwości obiektu i sekwencje liczb. Pokazałem także, jak skorzystać z aliasu, jak wydajnie zmieniać dane za pomocą klucza, a także jak pobrać indeks przetwarzanego elementu. Omówiłem również zastosowanie dyrektywy `v-for` w połączeniu z właściwościami obliczonymi, dzięki czemu możliwe jest stronicowanie, sortowanie i filtrowanie danych. W kolejnym rozdziale przedstawię obsługę zdarzeń za pomocą dyrektywy `Vue.js`.

ROZDZIAŁ 14.



Obsługa zdarzeń

W tym rozdziale kontynuuję pracę z wbudowanymi dyrektywami Vue.js, koncentrując się na dyrektywie `v-on`, która jest używana do obsługi zdarzeń (tabela 14.1 umiejscawia ją w szerszym kontekście).

Tabela 14.1. Umiejscowienie dyrektywy `v-on` w szerszym kontekście

Pytanie	Odpowiedź
Czym jest dyrektywa <code>v-on</code> ?	Dyrektyna <code>v-on</code> jest używana do nasłuchiwanie zdarzeń i reagowania na nie.
Dlaczego jest użyteczna?	Ta dyrektywa ułatwia dostęp do danych komponentu lub wykonywania metod w odpowiedzi na zdarzenia. Dzięki tej dyrektywie obsługa zdarzeń staje się integralną częścią tworzenia aplikacji w Vue.js.
Jak się z niej korzysta?	Dyrektyna <code>v-on</code> jest stosowana w elemencie HTML, którego zdarzenia Cię interesują. Wyrażenie jest ewaluowane w momencie powstania wybranych zdarzeń.
Czy są jakieś pułapki lub ograniczenia?	Dyrektyna <code>v-on</code> jest prosta w użyciu, o ile tylko pamiętasz, jak działa model propagacji zdarzeń DOM (por. punkt „Zarządzanie propagacją zdarzeń”).
Czy są jakieś rozwiązania alternatywne?	Jeśli interesują Cię zdarzenia wyzwalane przez elementy formularzy, rozsądniejszym wyborem może okazać się <code>v-model</code> .

Tabela 14.2 podsumowuje rozdział.

Przygotowania do tego rozdziału

W tym rozdziale kontynuujemy pracę z projektem `templatesanddata` z rozdziału 13. Aby przygotować się do tego rozdziału, uproszcilem komponent główny aplikacji (listing 14.1).

Tabela 14.2. Podsumowanie rozdziału

Problem	Rozwiązanie	Listing
Obsłuż zdarzenie wyemitowane przez element.	Skorzystaj z dyrektywy <code>v-on</code> .	14.3, 14.7
Pobierz szczegółowe informacje na temat zdarzenia.	Skorzystaj z obiektu zdarzenia.	14.4
Zareaguj na zdarzenie poza wyrażeniem dyrektywy.	Obsłuż zdarzenie za pomocą metody i otrzymaj obiekt zdarzenia jako parametr.	14.5 – 14.6
Obsłuż wiele zdarzeń z tego samego elementu.	Zastosuj dyrektywę <code>v-on</code> dla każdego zdarzenia, które chcesz otrzymać, lub wykryj rodzaj zdarzenia za pomocą obiektu zdarzenia.	14.8 – 14.9
Zarządzaj propagacją zdarzeń.	Skorzystaj z modyfikatorów propagacji zdarzeń.	14.10 – 14.14
Filtruj zdarzenia na podstawie aktywności myszy lub klawiatury.	Skorzystaj z modyfikatorów myszy lub klawiatury.	14.15 – 14.17

Listing 14.1. Uproszczenie zawartości pliku `src/App.vue`

```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3>{{ name }}</h3>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    }
  }
</script>
```

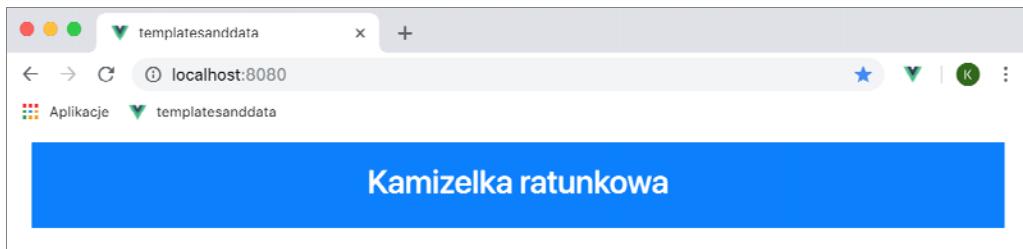
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.

Zapisz zmiany w pliku `src/App.vue` i uruchom polecenia z listingu 14.2 w katalogu `templatesanddata`, aby uruchomić narzędzia deweloperskie.

Listing 14.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, a zobaczysz efekt jak na rysunku 14.1.



Rysunek 14.1. Uruchomienie przykładowej aplikacji

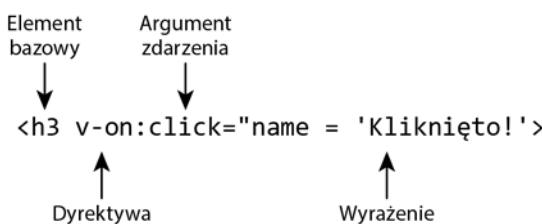
Obsługa zdarzeń

Vue.js dostarcza dyrektywę `v-on`, która jest używana do tworzenia wiązań do zdarzeń. Zdarzenia są wyzwalane w wyniku interakcji użytkownika z elementami HTML. Niektóre zdarzenia są obsługiwane przez wszystkie elementy, ale jest też cała paleta zdarzeń obsługiwanych przez wybrane elementy. W listingu 14.3 korzystam z dyrektywy `v-on`, aby poinformować Vue.js o konieczności zareagowania na kliknięcie elementu `h3` w szablonie komponentu.

Listing 14.3. Obsługa zdarzenia w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3 v-on:click="name = 'Kliknięto!'>{{ name }}</h3>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    }
  }
</script>
```

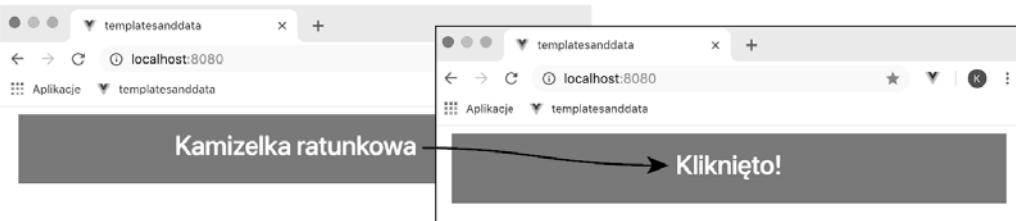
Zastosowanie dyrektywy `v-on` jest zgodne ze schematem wprowadzonym we wcześniejszych rozdziałach, który analizuję na rysunku 14.2.



Rysunek 14.2. Struktura dyrektywy `v-on`

Po nazwie dyrektywy następuje dwukropka, a następnie argument, który określa nazwę zdarzenia. Wyrażenie jest wykonywane w momencie wywołania zdarzenia. W tym przypadku wyrażeniem jest

fragment kodu JavaScript, który zmienia wartość właściwości name. Aby zapoznać się z wynikiem, zapisz zmiany i kliknij element h3 w oknie przeglądarki (rysunek 14.3).



Rysunek 14.3. Obsługa zdarzenia

Kliknięcie elementu wywala zdarzenie click, na które Vue.js reaguje, ewaluując wyrażenie dyrektywy. Po pierwszym kliknięciu nastąpi widoczna zmiana. Aby przywrócić aplikację do oryginalnego stanu, musisz odświeżyć przeglądarkę.

Omówienie zdarzeń i obiektów zdarzeń

W swojej pracy możesz spotkać się z wieloma rodzajami zdarzeń. Te, które omawiam w tym rozdziale, są opisane w tabeli 14.3.

Tabela 14.3. Zdarzenia omówione w tym rozdziale

Zdarzenie	Opis
click	To zdarzenie jest wyzwalane tuż po wciśnięciu i zwolnieniu przycisku myszy w obrębie elementu.
mousedown	To zdarzenie jest wyzwalane w momencie wciśnięcia przycisku myszy w obrębie elementu.
mousemove	To zdarzenie jest wyzwalane w momencie przesunięcia kurSORA w obrębie elementu.
mouseleave	To zdarzenie jest wyzwalane w momencie opuszczenia przez kurSOR elementu.
keydown	To zdarzenie jest wyzwalane w momencie wciśnięcia klawisza.

- **Uwaga** Szczegółowe informacje na temat dostępnych zdarzeń znajdziesz na stronie <https://developer.mozilla.org/en-US/docs/Web/Events>.

Gdy przeglądarka wywala zdarzenie, powstaje obiekt, który opisuje to zdarzenie — tzw. **obiekt zdarzenia**. Obiekt zdarzenia zawiera właściwości i metody, które dostarczają informacji na temat zdarzenia. Oprócz tego można zastosować go do kontroli sposobu przetwarzania zdarzenia. W tabeli 14.4 opisuję najbardziej użyteczne właściwości obiektów zdarzeń. Jeśli znasz się na tworzeniu aplikacji webowych, możesz zastanawiać się, co się stało z metodami i innymi właściwościami zdefiniowanymi w obiektach zdarzeń. Jak dowiesz się niebawem, nie musisz korzystać z nich bezpośrednio w przypadku użycia dyrektywy v-on, ponieważ zajmuje się ona sporą częścią procesu obsługi zdarzeń.

Dyrektyna v-on udostępnia obiekt zdarzeń za pomocą zmiennej o nazwie \$event. W listingu 14.4 zmieniłem wyrażenie dyrektywy, dzięki czemu wartość właściwości type jest wyświetlona w momencie wyzwolenia zdarzenia.

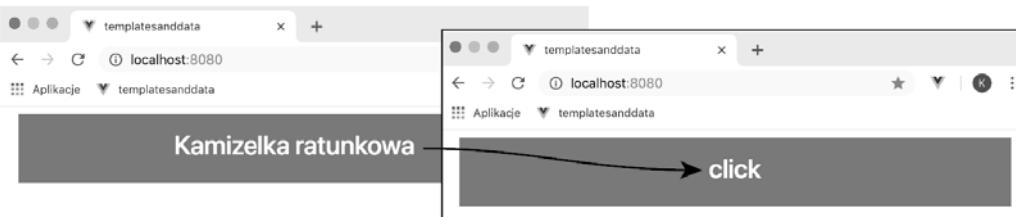
Tabela 14.4. Użyteczne właściwości obiektów zdarzeń

Właściwość	Opis
target	Ta właściwość zwraca obiekt modelu DOM reprezentujący element HTML, który wyzwoił zdarzenie.
currentTarget	Ta właściwość zwraca obiekt modelu DOM reprezentujący element HTML, który obsługuje zdarzenie. Różnicę między właściwością currentTarget i target wyjaśniono w punkcie „Zarządzanie propagacją zdarzeń”.
type	Ta właściwość zwraca rodzaj zdarzenia.
key	W przypadku zdarzeń klawiatury ta właściwość zwraca klawisz, do którego odnosi się zdarzenie.

Listing 14.4. Zastosowanie obiektu zdarzenia w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3 v-on:click="name = $event.type">{{ name }}</h3>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    }
  }
</script>
```

Dyrektywa v-on obsługuje zdarzenie click, wywołane z powodu kliknięcia elementu h3. Obiekt zdarzenia zostaje przypisany do zmiennej \$event przed ewaluacją wyrażenia, co generuje wynik jak na rysunek 14.4.

**Rysunek 14.4.** Wykorzystanie obiektu zdarzenia

Stosowanie metody do obsługi zdarzeń

Dyrektywa v-on ewaluje fragmenty kodu JavaScript w momencie wyzwolenia zdarzenia (co zaobserwowaliśmy w poprzednich przykładach), jednak typowym podejściem w tej sytuacji jest wywołanie metody. Zastosowanie metod minimalizuje ilość kodu w szablonie i pozwala na spójną obsługę zdarzeń. W listingu 14.5 dodaje metodę do komponentu i zmieniam wiązanie dyrektywy v-on, dzięki czemu metoda zostanie wywołana w momencie wyzwolenia zdarzenia click elementu.

Listing 14.5. Zastosowanie metody w pliku *src/App.vue*

```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3 v-on:click="handleEvent">{{ name }}</h3>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Kamizelka ratunkowa"
      }
    },
    methods: {
      handleEvent($event) {
        this.name = $event.type;
      }
    }
  }
</script>
```

Wyrażenie dyrektywy ustawilem na wartość `handleEvent`, co oznacza, że powinna ona wykonać metodę i przekazać do niej obiekt `$event` w momencie wygenerowania zdarzenia `click` (rysunek 14.5). Nie musisz przekazywać parametru `$event` jawnie — zdecydowałem się tak zrobić, aby pokazać zastosowanie parametru.

- **Wskazówka** Vue.js ma stosunkowo liberalne podejście do nazewnictwa metod wykonywanych w momencie wyzwolenia zdarzenia. Błąd zostanie jednak wygenerowany, jeśli podasz nazwę metody, która stanowi słowo kluczowe języka JavaScript (np. `delete`).

Określenie samej nazwy metody jest użyteczne, ale prawdziwą korzyść z zastosowania metod odniesiesz w przypadku dysponowania wieloma źródłami tego samego typu, które generują różne wyniki. W takiej sytuacji dyrektywa `v-for` może wywołać metodę z argumentami, dzięki którym możliwe jest określenie właściwego sposobu obsługi zdarzenia (listing 14.6).

Listing 14.6. Zastosowanie argumentów metod w pliku *src/App.vue*

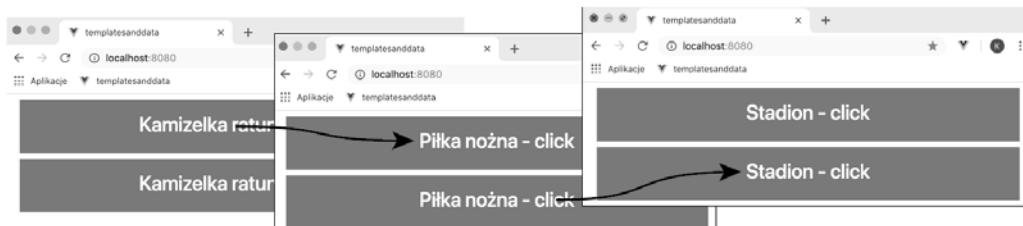
```
<template>
  <div class="container-fluid">
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3 v-on:click="handleEvent('Piłka nożna', $event)">{{ name }}</h3>
    </div>
    <div class="bg-primary text-white m-2 p-3 text-center">
      <h3 v-on:click="handleEvent('Stadion', $event)">{{ name }}</h3>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
```

```

        name: "Kamizelka ratunkowa"
    },
},
methods: {
    handleEvent(name, $event) {
        this.name = `${name} - ${$event.type}`;
    }
}
</script>

```

Dodałem kolejny element h3 z wiązaniem v-on. Oba wyrażenia wiązań wywołują metodę handleEvent, jednak różnica tkwi w wartości przekazanej w pierwszym argumentie. W rezultacie otrzymujemy różny komunikat w zależności od elementu klikniętego przez użytkownika (rysunek 14.5).



Rysunek 14.5. Wywołanie metody z argumentem

Stosowanie dyrektywy w formie skróconej

Dyrektyna v-on przyjmuje dwie postacie. Dłuższa, przedstawiona na rysunku 14.2, zawiera nazwę dyrektywy, dwukropek i nazwę zdarzenia. Krótsza zawiera jedynie znak @ połączony z nazwą zdarzenia. Dłuższa nazwa, v-on:click, jest równoznaczna z krótszą @click. Oznacza to, że dyrektywa taka jak ta:

```
...
<h3 v-on:click="handleEvent('Piłka nożna', $event)">{{ name }}</h3>
...
```

może być wyrażona następująco:

```
...
<h3 @click="handleEvent('Piłka nożna', $event)">{{ name }}</h3>
...
```

Wybór między dłuższą a krótszą formą nie wpływa na zmianę sposobu zachowania dyrektywy.

Połączenie zdarzeń, metod i elementów powtarzanych

Możliwość przekazywania argumentów w momencie wykonywania dyrektywy v-on staje się jeszcze bardziej użyteczna w połączeniu z dyrektywą v-for. W listingu 14.7 korzystam z dyrektywy v-for w celu powtórzenia zbioru elementów dla obiektów w tablicy, podczas gdy dyrektywa v-on służy do skonfigurowania mechanizmów obsługi zdarzenia dla każdego obiektu z osobna.

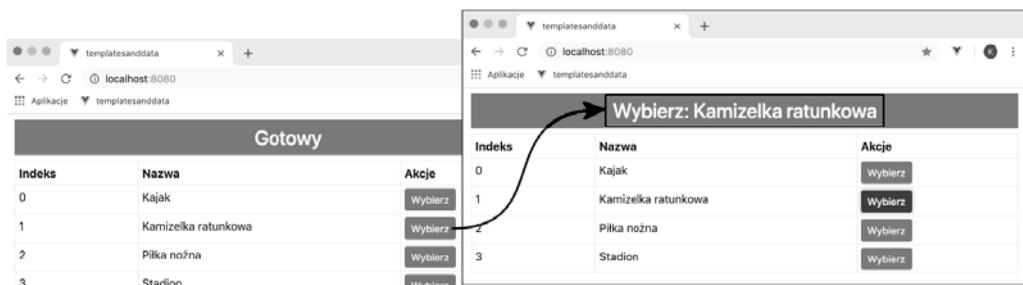
Listing 14.7. Powtarzanie elementów w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <h3 class="bg-primary text-white text-center mt-2 p-2">{{message}}</h3>
    <table class="table table-sm table-striped table-bordered">
      <tr><th>Indeks</th><th>Nazwa</th><th>Akcje</th></tr>
      <tr v-for="(name, index) in names" v-bind:key="name">
        <td>{{index}}</td>
        <td>{{name}}</td>
        <td>
          <button class="btn btn-sm bg-primary text-white"
                 v-on:click="handleClick(name)">
            Wybierz
          </button>
        </td>
      </tr>
    </table>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        message: "Gotowy",
        names: ["Kajak", "Kamizelka ratunkowa", "Piłka nożna", "Stadion"]
      }
    },
    methods: {
      handleClick(name) {
        this.message = `Wybierz: ${name}`;
      }
    }
  }
</script>
```

Dyrektywa `v-for` duplikuje wiersze w tabeli dla wszystkich obiektów z tablicy `names`. Duplikacja zawiera wszystkie wiązania danych i dyrektywy, co oznacza, że każdy wiersz tabeli zawiera przycisk z wiązaniem `v-on`:

```
...
<button class="btn btn-sm bg-primary text-white" v-on:click="handleClick(name)">
  Wybierz
</button>
...
```

Wyrażenie wiązania `v-on` jest konfigurowane za pomocą wartości aliasu `v-for`. Oznacza to, że każdy z przycisków wywoła metodę `handleClick`, przekazując obiekt przetwarzany w momencie utworzenia danego przycisku. Metoda `handleClick` korzysta z wartości parametru, aby ustawić wartość właściwości `message` wyświetlonej w elemencie `h3`. W rezultacie kliknięcie przycisku w tabeli spowoduje wyświetlenie wartości z tablicy `names` (rysunek 14.6).



Rysunek 14.6. Obsługa zdarzeń dla powtarzających się elementów

Nasłuchiwanie wielu zdarzeń z tego samego elementu

Obsługiwanie różnych zdarzeń z tego samego elementu jest możliwe na dwa sposoby. Pierwszy z nich zakłada użycie dyrektywy `v-on` oddzielnie dla każdego zdarzenia (listing 14.8).

Listing 14.8. Obsługa wielu rodzajów zdarzeń w pliku src/App.vue

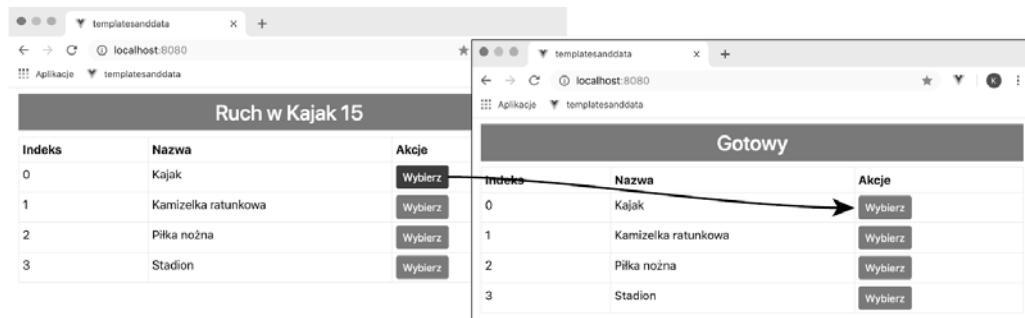
```
<template>
  <div class="container-fluid">
    <h3 class="bg-primary text-white text-center mt-2 p-2">{{message}}</h3>
    <table class="table table-sm table-striped table-bordered">
      <tr><th>Indeks</th><th>Nazwa</th><th>Akcje</th></tr>
      <tr v-for="(name, index) in names" v-bind:key="name">
        <td>{{index}}</td>
        <td>{{name}}</td>
        <td>
          <button class="btn btn-sm bg-primary text-white"
                 v-on:click="handleClick(name)"
                 v-on:mousemove="handleMouseEvent(name, $event)"
                 v-on:mouseleave="handleMouseEvent(name, $event)">
            Wybierz
          </button>
        </td>
      </tr>
    </table>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        counter: 0,
        message: "Gotowy",
        names: ["Kajak", "Kamizelka ratunkowa", "Piłka nożna", "Stadion"]
      }
    },
    methods: {
      handleClick(name) {
        this.message = `Wybierz: ${name}`;
      },
    }
  }
</script>
```

```

        handleMouseEvent(name, $event) {
            if ($event.type == "mousemove") {
                this.message = `Ruch w ${name} ${this.counter++}`;
            } else {
                this.counter = 0;
                this.message = "Gotowy";
            }
        }
    }
</script>

```

Dodałem dyrektywy `v-on`, aby obsługiwać zdarzenia `mousemove` i `mouseleave`. Zdarzenie `mousemove` jest wyzwalane w momencie przesunięcia kurSORA myszy nad elementem, a `mouseleave` — w momencie opuszczenia elementu. Oba zdarzenia są przetwarzane przez metodę `handleMouseEvent`, która korzysta ze zmiennej `$event`, aby określić rodzaj zdarzenia, i aktualizuje właściwości `message` i `counter`. Efektem jest komunikat, który pojawia się w momencie przesuwania kurSORA myszy nad przyciskami, a który znika, gdy użytkownik opuści obszar przycisku (rysunek 14.7).



Rysunek 14.7. Reagowanie na różne rodzaje zdarzeń

Drugim sposobem obsługi wielu zdarzeń jest zastosowanie dyrektywy `v-on` bez argumentu zdarzenia i ustwienie wyrażenia dyrektywy na obiekt, którego nazwy właściwości stanowią rodzaje zdarzeń, a wartości — metody do wywołania (listing 14.9).

Listing 14.9. Obsługa wielu zdarzeń w pojedynczej dyrektywie (src/App.vue)

```

<template>
    <div class="container-fluid">
        <h3 class="bg-primary text-white text-center mt-2 p-2">{{message}}</h3>
        <table class="table table-sm table-striped table-bordered">
            <tr><th>Indeks</th><th>Nazwa</th><th>Akcje </th></tr>
            <tr v-for="(name, index) in names" v-bind:key="name">
                <td>{{index}}</td>
                <td>{{name}}</td>
                <td>
                    <button class="btn btn-sm bg-primary text-white"
                           v-on="buttonEvents"
                           v-bind:data-name="name">
                        Wybierz
                    </button>
                </td>
            </tr>
        </table>
    </div>

```

```
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                buttonEvents: {
                    click: this.handleClick,
                    mousemove: this.handleMouseEvent,
                    mouseleave: this.handleMouseEvent
                },
                counter: 0,
                message: "Gotowy",
                names: ["Kajak", "Kamizelka ratunkowa", "Piłka nożna", "Stadion"]
            }
        },
        methods: {
            handleClick($event) {
                let name = $event.target.dataset.name;
                this.message = `Select: ${name}`;
            },
            handleMouseEvent($event) {
                let name = $event.target.dataset.name;
                if ($event.type == "mousemove") {
                    this.message = `Ruch w ${name} ${this.counter++}`;
                } else {
                    this.counter = 0;
                    this.message = "Gotowy";
                }
            }
        }
    }
</script>
```

Właściwość danych o nazwie buttonEvents zwraca obiekt o właściwościach click, mousemove i mouseleave, odpowiadających zdarzeniom, które chcę przetworzyć. Wartościami tych właściwości są metody, które powinny być wykonywane dla każdego zdarzenia. Ograniczeniem tego podejścia jest brak możliwości przekazania argumentów do metod — otrzymają one tylko argument \$event. Obejściem jest zastosowanie dyrektywy v-bind i przekazanie atrybutu data-name do każdego przycisku, np.:

```
...
<button class="btn btn-sm bg-primary text-white" v-on="buttonEvents"
    v-bind:data-name="name">
...

```

Obiekt \$event daje dostęp do elementu HTML, który wyzwoił zdarzenie, tak więc korzystam z właściwości dataset w celu dostępu do własnych atrybutów data-. Pobieram wartość atrybutu data-name podczas przetwarzania danych w następujący sposób:

```
...
let name = $event.target.dataset.name;
...
```

Efekt jest ten sam co w listingu 14.8, ale nie ma potrzeby definiowania wielu dyrektyw v-on w przyciskach, co często utrudnia analizę szablonu.

- **Wskazówka** Obiekt powiązania zdarzeń możesz zdefiniować, jeśli wolisz, w formie literalu tekstowego bezpośrednio w wyrażeniu dyrektywy v-on. Taki szablon będzie jednak niewątpliwie trudniejszy do zrozumienia.

Stosowanie modyfikatorów obsługi zdarzeń

Aby uprościć metody przetwarzania zdarzeń, dyrektywa `v-on` obsługuje zbiór modyfikatorów używanych do wykonywania typowych zadań, które z reguły wiążą się z wykonaniem instrukcji języka JavaScript. Tabela 14.5 przedstawia zbiór modyfikatorów obsługi zdarzeń dla dyrektywy `v-on`.

Tabela 14.5. Modyfikatory obsługi zdarzeń `v-on`

Modyfikator	Opis
<code>stop</code>	Ten modyfikator jest równoważny wywołaniu metody <code>stopPropagation</code> obiektu zdarzenia (por. „Zatrzymywanie propagacji zdarzeń”).
<code>prevent</code>	Ten modyfikator jest równoważny wywołaniu metody <code>preventDefault</code> obiektu zdarzenia. Szerzej opisuję go w rozdziale 15.
<code>capture</code>	Ten modyfikator włącza tryb przechwytywania dla propagacji zdarzeń (por. „Otrzymywanie zdarzeń w fazie przechwytywania”).
<code>self</code>	Ten modyfikator wywołuje metodę obsługi tylko, jeśli zdarzenie pochodzi z elementu, do którego została zastosowana dyrektywa (por. „Obsługa zdarzeń wyłącznie z fazy celu”).
<code>once</code>	Ten modyfikator uniemożliwia wywołanie metody obsługi przez kolejne zdarzenia tego samego typu (por. „Zapobieganie duplikacji zdarzeń”).
<code>passive</code>	Ten modyfikator umożliwia nasłuchiwanie zdarzeń pasywnych, co pozwala usprawnić wydajność zdarzeń dotyku i jest użyteczne na urządzeniach mobilnych (por. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener).

Zarządzanie propagacją zdarzeń

Modyfikatory `stop`, `capture` i `self` są używane do zarządzania propagacją zdarzenia w obrębie hierarchii elementów HTML. W momencie wyzwolenia zdarzenia przeglądarka przechodzi przez trzy fazy — przechwytywania (ang. *capture*), celu (ang. *target*) i pęcherzykową (ang. *bubble*) — aby znaleźć metody obsługi zdarzeń. Listing 14.10 przedstawia, jak działają te fazy i jak można kontrolować je za pomocą modyfikatorów `v-on`.

Listing 14.10. Obsługa zdarzeń w pliku `src/App.vue`

```
<template>
  <div class="container-fluid">
    <div id="outer-element" class="bg-primary p-4 text-white h3"
      v-on:click="handleClick">
      Element zewnętrzny
      <div id="middle-element" class="bg-secondary p-4"
        v-on:click="handleClick">
        Element pośredni
        <div id="inner-element" class="bg-info p-4"
          v-on:click="handleClick">
          Element wewnętrzny
        </div>
      </div>
    </div>
  </template>
<script>
  export default {
    name: "MyComponent",
```

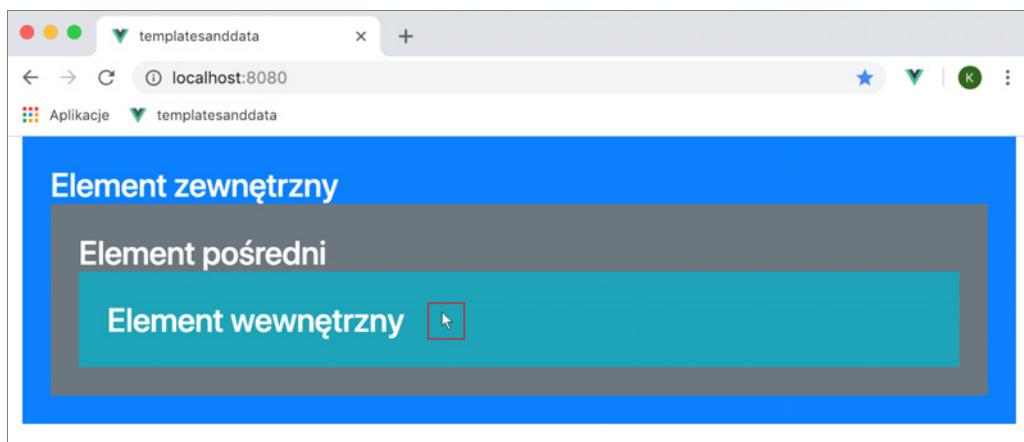
```

methods: {
    handleClick($event) {
        console.log(`handleClick target: ${$event.target.id}-
                    + ^ currentTarget: ${$event.currentTarget.id}`);
    }
}
</script>

```

Szablon zawiera trzy zagnieżdżone elementy z atrybutem id, a dyrektywa v-on jest ustawiona tak, aby przetwarzać zdarzenie click za pomocą metody handleClick. Metoda otrzymuje obiekt \$event i korzysta z właściwości target, aby pobrać szczegółowe informacje dotyczące elementu, który wyzwoił zdarzenie, oraz z właściwości currentTarget, aby pobrać informacje na temat elementu, którego dyrektywa v-on przetwarza zdarzenie.

Aby zrozumieć, dlaczego obsługa zdarzenia wymaga dwóch właściwości, zapisz zmiany w komponencie, a następnie kliknij fragment okna zajęty przez element wewnętrzny (rysunek 14.8).



Rysunek 14.8. Wyzwolenie zdarzenia w zbiorze zagnieżdżonych elementów

Domyślnie dyrektywa v-on otrzymuje zdarzenia w fazach pęcherzykowej i celu, co oznacza, że zdarzenie otrzyma najpierw dyrektywa v-on w elemencie, który wyzwoilił zdarzenie (celu). Dopiero potem otrzymają je wszyscy przodkowie, aż do głównego elementu nadrzednego w elemencie body. Komunikaty będą widoczne w konsoli JavaScript.

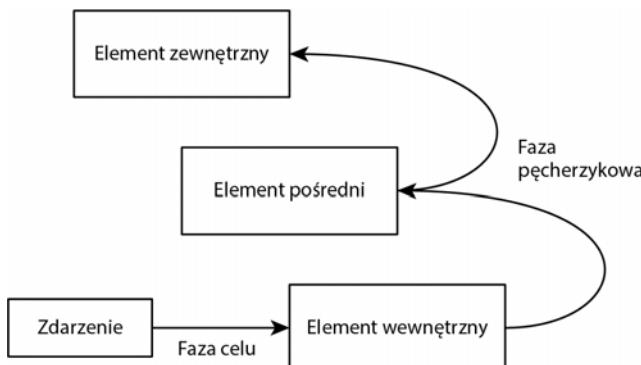
```

...
handleClick target: inner-element currentTarget: inner-element
handleClick target: inner-element currentTarget: middle-element
handleClick target: inner-element currentTarget: outer-element
...

```

Pierwszy komunikat przedstawia fazę celu zdarzenia, w której następuje wywołanie metody handleClick przez dyrektywę v-on zadeklarowaną w elemencie wywołującym zdarzenie. W fazie celu właściwości target i currentTarget obiektu \$event są identyczne.

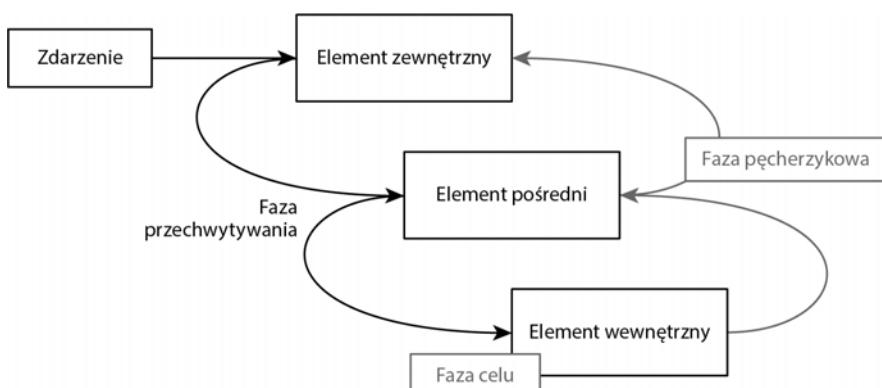
Drugi i trzeci komunikat przedstawiają fazę pęcherzykową, w której zdarzenie przechodzi w górę dokumentu HTML, przez kolejnych przodków, powodując wywołanie metody powiązanej z dyrektywą v-on. W tej fazie właściwość target obiektu \$event zwraca element, który wyzwoilił zdarzenie, a currentTarget — aktualnie przetwarzany element. Rysunek 14.9 przedstawia przepływ zdarzeń w fazach celu i pęcherzykowej.

**Rysunek 14.9.** Fazy celu i pęcherzykowa

- **Wskazówka** Nie wszystkie rodzaje zdarzeń mają fazę pęcherzykową. Z reguły zdarzenia powiązane z konkretnym elementem — np. uzyskanie i utracenie fokus — nie zaliczają się do tej fazy. Zdarzenia, które dotyczą wielu elementów — np. kliknięcie obszaru na ekranie, który jest zajmowany przez wiele elementów — będą zaliczać się do tej fazy. Aby sprawdzić, czy dane zdarzenie przechodzi przez tę fazę, należy sprawdzić właściwość bubbles obiektu \$event.

Otrzymywanie zdarzeń w fazie przechwytywania

Faza przechwytywania daje możliwość przetworzenia zdarzeń przez elementy, zanim nastąpi faza celu. Podczas fazy przechwytywania przeglądarka rozpoczyna przetwarzanie od elementu body i przechodzi w hierarchii coraz niżej, aż do elementu docelowego, podążając odwrotną ścieżką niż w fazie pęcherzykowej, co daje szansę na przetworzenie zdarzenia przez każdy element (rysunek 14.10).

**Rysunek 14.10.** Faza przechwytywania

Dyrektiva v-on nie otrzyma zdarzeń podczas fazy przechwytywania, jeżeli nie zastosujesz modyfikatora capture (listing 14.11).

Listing 14.11. Faza przechwytywania w praktyce (src/App.vue)

```
<template>
<div class="container-fluid">
  <div id="outer-element" class="bg-primary p-4 text-white h3">
```

```

    v-on:click.capture="handleClick">
Element zewnętrzny
<div id="middle-element" class="bg-secondary p-4"
    v-on:click.capture="handleClick">
Element pośredni
<div id="inner-element" class="bg-info p-4"
    v-on:click="handleClick">
Element wewnętrzny
</div>
</div>
</div>
</template>
<script>
export default {
    name: "MyComponent",
    methods: {
        handleClick($event) {
            console.log(`handleClick target: ${$event.target.id}~
+ ` currentTarget: ${$event.currentTarget.id}`);
        }
    }
}
</script>

```

Modyfikatory są włączane za pomocą nazwy modyfikatora następującej po nazwie zdarzenia i kropce:

```

...
v-on:click.capture="handleClick"
...

```

Dodanie modyfikatora capture do dyrektywy v-on oznacza, że elementy otrzymają zdarzenie w fazie przechwytywania, a nie pęcherzykowej. Efekt zobaczysz w konsoli JavaScript:

```

...
handleClick target: inner-element currentTarget: outer-element
handleClick target: inner-element currentTarget: middle-element
handleClick target: inner-element currentTarget: inner-element
...

```

Komunikaty pokazują, że zdarzenie przeszło w dół dokumentu HTML, wyzwalając dyrektywę v-on najpierw w elemencie zewnętrznym, a następnie pośrednim.

Obsługa zdarzeń wyłącznie z fazy celu

Istnieje możliwość ograniczenia dyrektywy v-on, tak aby reagowała wyłącznie na zdarzenia z fazy celu. Modyfikator self zapewnia, że będą przetwarzane tylko zdarzenia wyzwolone przez dany element (listing 14.12).

Listing 14.12. Wybór zdarzeń fazy celu (*src/App.vue*)

```

<template>
<div class="container-fluid">
    <div id="outer-element" class="bg-primary p-4 text-white h3"
        v-on:click.capture="handleClick">
Element zewnętrzny
    <div id="middle-element" class="bg-secondary p-4"
        v-on:click.self="handleClick">
Element pośredni

```

```

<div id="inner-element" class="bg-info p-4"
      v-on:click="handleClick">
    Element wewnętrzny
  </div>
</div>
</div>
</template>
<script>
  export default {
    name: "MyComponent",
    methods: {
      handleClick($event) {
        console.log(`handleClick target: ${$event.target.id}` +
                    ` currentTarget: ${$event.currentTarget.id}`);
      }
    }
  }
</script>

```

Modyfikator `self` zastosowałem wobec dyrektywy `v-on` elementu pośredniego. Jeśli klikniesz element wewnętrzny, w konsoli JavaScript zobaczyś następujące komunikaty:

```

...
handleClick target: inner-element currentTarget: outer-element
handleClick target: inner-element currentTarget: inner-element
...

```

Pierwszy komunikat otrzymujemy z elementu zewnętrznego, którego dyrektywa `v-on` zawiera modyfikator `capture`, dzięki czemu element ten uzyskuje zdarzenie w fazie przechwytywania. Drugi komunikat otrzymujemy z elementu wewnętrznego, który jest celem i otrzymuje zdarzenie w czasie fazy celu. Nie widzimy komunikatu z elementu pośredniego, ponieważ modyfikator `self` zapobiegł przekazaniu zdarzenia do fazy pęcherzykowej. Jeśli klikniesz element pośredni, zobaczysz następujące komunikaty:

```

...
handleClick target: middle-element currentTarget: outer-element
handleClick target: middle-element currentTarget: middle-element
...

```

Pierwszy komunikat pochodzi z elementu zewnętrznego, który otrzymuje zdarzenie podczas fazy przechwytywania. Drugi komunikat pochodzi z elementu pośredniego w czasie fazy celu, co jest dopuszczone przez modyfikator `self`.

Zatrzymywanie propagacji zdarzeń

Modyfikator `stop` zatrzymuje propagację zdarzenia, powstrzymując je od przetwarzania przez dyrektywy `v-on` kolejnych elementów. W listingu 14.13 zastosowałem dyrektywę `stop` do dyrektywy elementu pośredniego.

Listing 14.13. Zatrzymywanie propagacji zdarzeń (`src/App.vue`)

```

<template>
  <div class="container-fluid">
    <div id="outer-element" class="bg-primary p-4 text-white h3"
          v-on:click.capture="handleClick">
      Element zewnętrzny
    <div id="middle-element" class="bg-secondary p-4"
          v-on:click.stop="handleClick">
      Element pośredni
    </div>
  </div>
</template>

```

```

<div id="inner-element" class="bg-info p-4"
      v-on:click="handleClick">
    Element wewnętrzny
  </div>
</div>
</div>
</template>
<script>
  export default {
    name: "MyComponent",
    methods: {
      handleClick($event) {
        console.log(`handleClick target: ${$event.target.id}` +
                    ` currentTarget: ${$event.currentTarget.id}`);
      }
    }
  }
</script>

```

Modyfikator stop powstrzymuje zdarzenie przed dalszym przetwarzaniem według standardowego schematu, ale nie zatrzymuje on propagacji aż do momentu napotkania elementu, do którego modyfikator ten został zastosowany. W tym przykładzie oznacza to, że zdarzenie wyzwolone przez element pośredni przejdzie przez fazę przechwytywania przed powstrzymaniem w fazie celu. Skoro skonfigurowałem dyrektywę v-on z modyfikatorem capture, wydruk w konsoli przeglądarki będzie następujący:

```

...
handleClick target: middle-element currentTarget: outer-element
handleClick target: middle-element currentTarget: middle-element
...

```

Łączenie modyfikatorów obsługi zdarzeń

Pojedyncza dyrektywa v-on może zawierać wiele modyfikatorów. Modyfikatory te są powtarzane w kolejności zdefiniowania, co oznacza, że można osiągnąć różne efekty za pomocą tych samych modyfikatorów w różnych kombinacjach. Poniższa kombinacja przetworzy zdarzenia w fazie celu, a następnie zatrzyma przed dalszą propagacją.

```

...
v-on:click.self.stop="handleClick"
...

```

Jeśli jednak odwróciś te modyfikatory, zdarzenia zostaną zatrzymane w fazach pęcherzykowej i celu, ponieważ modyfikator stop występuje jako pierwszy.

```

...
v-on:click.stop.self="handleClick"
...

```

W związku z tym warto pamiętać o konsekwencjach łączenia modyfikatorów. Warto testować wprowadzone kombinacje bardzo dokładnie i stosować je spójnie.

Zapobieganie duplikacji zdarzeń

Modyfikator once zatrzymuje dyrektywę v-on przed wywołaniem metody więcej niż raz. Nie zatrzymujemy w ten sposób zwykłego procesu przetwarzania zdarzeń, jednak powstrzymujemy dany element od uczestniczenia w tym procesie po przetworzeniu pierwszego zdarzenia. W listingu 14.14 zastosowałem modyfikator once do elementu wewnętrznego w szablonie komponentu.

Listing 14.14. Zapobieganie duplikacji zdarzeń w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div id="outer-element" class="bg-primary p-4 text-white h3"
        v-on:click.capture="handleClick">
      Element zewnętrzny
      <div id="middle-element" class="bg-secondary p-4"
          v-on:click.stop="handleClick">
        Element pośredni
        <div id="inner-element" class="bg-info p-4"
            v-on:click.once="handleClick">
          Element wewnętrzny
        </div>
      </div>
    </div>
  </template>
<script>
  export default {
    name: "MyComponent",
    methods: {
      handleClick($event) {
        console.log(`handleClick target: ${$event.target.id}` +
                    ` currentTarget: ${$event.currentTarget.id}`);
      }
    }
  }
</script>
```

Po pierwszym kliknięciu elementu wewnętrznego zobaczysz, że wszystkie dyrektywy v-on zareagują:

```
...
handleClick target: inner-element currentTarget: outer-element
handleClick target: inner-element currentTarget: inner-element
handleClick target: inner-element currentTarget: middle-element
...
```

Element zewnętrzny można skonfigurować za pomocą modyfikatora capture, dzięki czemu otrzyma on zdarzenie podczas fazy przechwytywania. Jest to pierwsza sytuacja, w której element wewnętrzny obsługuje zdarzenie click, tak więc faza celu jest przeprowadzana jak zwykle. Na zakończenie element pośredni otrzymuje zdarzenie w czasie fazy pęcherzykowej i w tym momencie modyfikator stop uniemożliwia przejście dalej.

Po kliknięciu elementu wewnętrznego zobaczysz inny zestaw komunikatów:

```
...
handleClick target: inner-element currentTarget: outer-element
handleClick target: inner-element currentTarget: middle-element
...
```

Modyfikator once zapobiega ponownemu wywołaniu metody handleClick, ale nie przerywa propagacji do innych elementów.

Omówienie modyfikatorów zdarzeń myszy

Dyrektywa `v-on` dostarcza zestaw modyfikatorów, który upraszcza przetwarzanie zdarzeń myszy, gdy chcesz precyzyjnie określić warunki wygenerowania zdarzenia. W tabeli 14.6 opisano zbiór modyfikatorów zdarzeń myszy.

Tabela 14.6. Modyfikatory zdarzeń myszy dla dyrektywy `v-on`

Modyfikator	Opis
<code>left</code>	Ten modyfikator dopuszcza tylko zdarzenia wyzwolone przez lewy przycisk myszy.
<code>middle</code>	Ten modyfikator dopuszcza tylko zdarzenia wyzwolone przez środkowy przycisk myszy.
<code>right</code>	Ten modyfikator dopuszcza tylko zdarzenia wyzwolone przez prawy przycisk myszy.

Modyfikatory, gdy są stosowane wobec dyrektywy `v-on`, ograniczają przetwarzane zdarzenia do tych wyzwolonych przez konkretny przycisk myszy (listing 14.15).

Listing 14.15. Zastosowanie modyfikatorów myszy (`src/App.vue`)

```
<template>
  <div class="container-fluid">
    <div id="outer-element" class="bg-primary p-4 text-white h3"
        v-on:click.capture="handleClick">
      Element zewnętrzny
      <div id="middle-element" class="bg-secondary p-4"
          v-on:click.stop="handleClick">
        Element pośredni
        <div id="inner-element" class="bg-info p-4"
            v-on:mousedown.right="handleClick">
          Element wewnętrzny
        </div>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    methods: {
      handleClick($event) {
        console.log(`handleClick target: ${$event.target.id}` +
                    ` currentTarget: ${$event.currentTarget.id}`);
      }
    }
  }
</script>
```

W tym listingu zmieniłem dyrektywy `v-on`, dzięki czemu nasłuchuję one zdarzenia `mousedown`. Zdarzenie to jest wyzwalane w momencie, gdy dowolny przycisk jest wcisnięty nad elementem. Zastosowanie modyfikatora `right` ograniczy zdarzenia do tych generowanych przez prawy przycisk myszy.

Modyfikatory nie zapobiegają dalszej propagacji zdarzenia. Ich celem jest zablokowanie przetwarzania zdarzeń wyzwolonych za pomocą innych przycisków myszy. Jeśli klikniesz lewym przyciskiem myszy, zobaczysz poniższe komunikaty w konsoli przeglądarki. Elementy pośredni i zewnętrzny otrzymują zdarzenie, ale nie jest ono przetwarzane przez dyrektywę elementu wewnętrznego:

```
...
handleClick target: inner-element currentTarget: middle-element
handleClick target: inner-element currentTarget: outer-element
...
```

Jeśli klikniesz prawym przyciskiem myszy element wewnętrzny, wszystkie trzy dyrektywy v-on przetworzą zdarzenie, co da następujący efekt:

```
...
handleClick target: inner-element currentTarget: inner-element
handleClick target: inner-element currentTarget: middle-element
handleClick target: inner-element currentTarget: outer-element
...
```

Omówienie modyfikatorów zdarzeń klawiatury

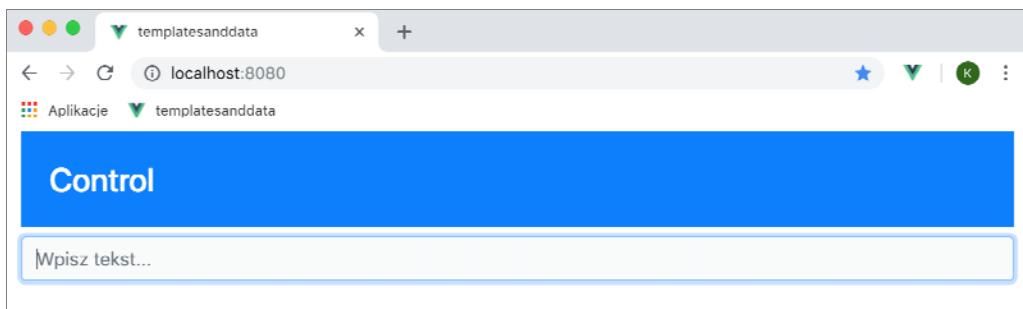
Vue.js dostarcza zbiór modyfikatorów zdarzeń klawiatury, które są używane do ograniczenia zdarzeń klawiatury przetwarzanych przez dyrektywę v-on w podobny sposób jak modyfikatory myszy opisane przed chwilą. W listingu 14.16 modyfikuję komponent, dzięki czemu zawiera on element input, za pomocą którego przechwyczę zdarzenia klawiatury.

Listing 14.16. Otrzymywanie zdarzeń klawiatury w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-primary p-4 text-white h3">
      {{message}}
    </div>
    <input class="form-control bg-light" placeholder="Wpisz tekst..." 
      ↳v-on:keydown.ctrl="handleKey" />
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        message: "Gotowy"
      }
    },
    methods: {
      handleKey($event) {
        this.message = $event.key;
      }
    }
  }
</script>
```

W tym przykładzie stosujemy wobec zdarzenia keydown modyfikator ctrl, który oznacza, że zdarzenie keydown będzie wykonane tylko, jeśli wciśnięty będzie klawisz *Ctrl*. Zachowanie to sprawdzisz, wprowadzając treść w pole input — znaki będą umieszczane w elemencie div, ale tylko jeśli wciśnięty będzie klawisz *Ctrl* (rysunek 14.11).

Tabela 14.7 przedstawia zbiór modyfikatorów zdarzeń klawiatury, dostarczonych przez Vue.js. Niektóre z tych modyfikatorów to klawisze, które mogą być łączone z innymi (np. *Shift* lub *Ctrl*), a pozostałe pozwalają na łatwe określenie często potrzebnych klawiszy, takich jak *Tab* czy spacja.



Rysunek 14.11. Zastosowanie modyfikatora do filtrowania zdarzeń klawiatury

Tabela 14.7. Modyfikatory zdarzeń klawiatury dla dyrektywy v-on

Modyfikator	Opis
enter	Ten modyfikator wybiera klawisz <i>Enter</i> .
tab	Ten modyfikator wybiera klawisz <i>Tab</i> .
delete	Ten modyfikator wybiera klawisz <i>Delete</i> .
esc	Ten modyfikator wybiera klawisz <i>Escape</i> .
space	Ten modyfikator wybiera klawisz spacji.
up	Ten modyfikator wybiera klawisz ze strzałką w góre.
down	Ten modyfikator wybiera klawisz ze strzałką w dół.
left	Ten modyfikator wybiera klawisz ze strzałką w lewo.
right	Ten modyfikator wybiera klawisz ze strzałką w prawo.
ctrl	Ten modyfikator wybiera klawisz <i>Ctrl</i> .
alt	Ten modyfikator wybiera klawisz <i>Alt</i> .
shift	Ten modyfikator wybiera klawisz <i>Shift</i> .
meta	Ten modyfikator wybiera klawisz <i>Meta</i> .
exact	Ten modyfikator wybiera określony klawisz modyfikatora (co opisuję poniżej).

Modyfikator exact jeszcze bardziej ogranicza wiązanie danych, dzięki czemu będzie ono wywołane tylko w momencie wcisnięcia określonych klawiszy. Modyfikator w listingu 14.17 wywoła metodę `handleKey` tylko w momencie wcisnięcia klawisza *Ctrl*, ale już w przypadku wcisnięcia klawiszy *Ctrl* i *Shift* — nie.

Listing 14.17. Dokładne dopasowywanie klawiszy w pliku *src/App.vue*

```
<template>
  <div class="container-fluid">
    <div class="bg-primary p-4 text-white h3">
      {{message}}
    </div>
    <input class="form-control bg-light" placeholder="Wpisz tekst...">
    <input v-on:keydown.ctrl.exact="handleKey" />
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
```

```
        return {
          message: "Gotowy"
        }
      },
      methods: {
        handleKey($event) {
          this.message = $event.key;
        }
      }
    }
  </script>
```

Podsumowanie

W tym rozdziale przedstawiłem różne sposoby użycia dyrektywy `v-on` w celu reagowania na zdarzenia. Pokazałem, jak określić jedno lub większą liczbę zdarzeń podczas stosowania dyrektywy, jak wykonywać metody i otrzymywać argumenty przy generowaniu powtórzonej treści, a także jak stosować modyfikatory w celu zmiany zachowania zdarzeń. Wiesz już także, jak obsłużyć wybrane klawisze na klawiaturze i przyciski myszy. W kolejnym rozdziale zajmę się funkcjami Vue.js związanymi z elementami formularzy.

ROZDZIAŁ 15.



Obsługa elementów formularzy

W tym rozdziale opisuję wbudowaną dyrektywę `v-model`, używaną w odniesieniu do elementów formularzy HTML. Dyrektywa `v-model` tworzy dwukierunkowe wiązanie danych pomiędzy elementem formularza a wartością danych, zapewniając, że aplikacja pozostaje w stanie spójnym niezależnie od zachodzących zmian danych. Pokażę także, jak połączyć dyrektywę `v-model` z wbudowanymi dyrektywami opisanymi w poprzednich rozdziałach, aby zwalidować dane wprowadzone do formularza przez użytkownika. Tabela 15.1 umiejscowia dyrektywę `v-model` w szerszym kontekście.

Tabela 15.1. Umiejscowienie dyrektywy `v-model` w szerszym kontekście

Pytanie	Odpowiedź
Czym jest dyrektywa <code>v-model</code> ?	Dyrektyna <code>v-model</code> tworzy wiązanie dwukierunkowe między elementem formularza HTML a właściwością danych, zapewniając spójność obu części.
Dlaczego jest użyteczna?	Praca z elementami formularzy stanowi ważny aspekt tworzenia aplikacji webowych. Dyrektywa <code>v-model</code> zapewnia wiązania danych bez konieczności pamiętania, na czym polegają różnice w obsłudze poszczególnych elementów formularzy.
Jak się z niej korzysta?	Dyrektyna <code>v-model</code> jest używana w elementach <code>input</code> , <code>select</code> i <code>text area</code> . Wyrażenie zawiera właściwość danych, z którym chce się utworzyć wiązanie.
Czy są jakieś pułapki lub ograniczenia?	Dyrektyna <code>v-model</code> nie może być używana z magazynami danych Vuex, które opisuję w rozdziale 20.
Czy są jakieś rozwiązania alternatywne?	Możesz utworzyć wiązania ręcznie — tę operację opisuję w podrozdziale „Tworzenie dwukierunkowych wiązań modeli”.

Tabela 15.2 podsumowuje rozdział.

Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę nad projektem `templatesanddata` z rozdziału 14. Zaczynam od uproszczenia głównego komponentu aplikacji (listing 15.1).

Tabela 15.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Utwórz dwukierunkowe wiązanie danych między właściwością a elementem formularza.	Skorzystaj z dyrektywy v-model.	15.1 – 15.9
Sformatuj wartości wejściowe jako liczby.	Skorzystaj z modyfikatora number.	15.10 – 15.11
Opóźnij aktualizacje wiązania.	Skorzystaj z modyfikatora lazy.	15.12
Pozbądź się białych znaków z wartości wejściowych.	Skorzystaj z modyfikatora trim.	15.13
Wypełnij tablicę wartościami podanymi przez użytkownika.	Skorzystaj z dyrektywy v-model, aby powiązać ją z tablicą.	15.14
Powiąż elementy pośrednio.	Skorzystaj z własnej wartości w dyrektywie v-model.	15.15 – 15.17
Upewnij się, że użytkownik wprowadza dopuszczalne wartości.	Zwaliduj dane formularza pobrane za pomocą dyrektywy v-model.	15.18 – 15.21

Listing 15.1. Uproszczenie treści w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      Wartość: {{ dataValue }}
    </div>
    <div class="bg-primary m-2 p-2 text-white">
      <div class="form-check">
        <label class="form-check-label">
          <input class="form-check-input" type="checkbox"
            v-on:change="handleChange" />
          Wartość danych
        </label>
      </div>
      <div class="text-center m-2">
        <button class="btn btn-secondary" v-on:click="reset">
          Reset
        </button>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        dataValue: false
      }
    },
    methods: {
      reset() {
        this.dataValue= false;
      },
      handleChange($event) {
        this.dataValue = $event.target.checked;
      }
    }
  }
</script>
```

```

        }
    }
</script>

```

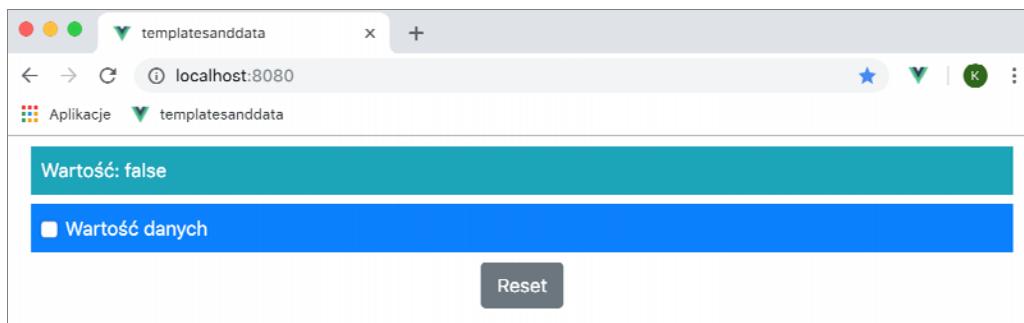
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Zapisz zmiany w pliku *App.vue* i wykonaj polecenie z listingu 15.2 w katalogu *templatesanddata*, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 15.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, a zobaczysz efekt jak na rysunku 15.1.



Rysunek 15.1. Przykładowa aplikacja po uruchomieniu

Tworzenie dwukierunkowych wiązań modeli

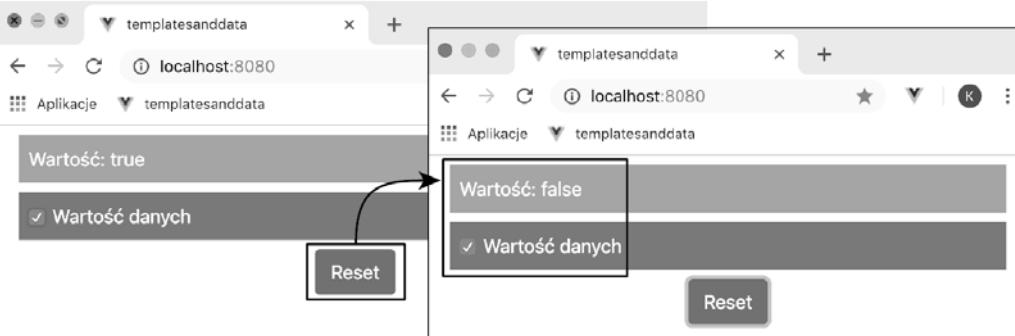
Wszystkie wiązania danych, które utworzyłem do tej pory, były jednokierunkowe, co oznacza, że przepływ danych odbywał się jedynie z elementu *script* do elementu *template* komponentu, dzięki czemu mógł być on wyświetlony użytkownikowi.

Przykładowa aplikacja przedstawia jednokierunkowy przepływ danych. Gdy stan przycisku wyboru zmienia się, dyrektywa *v-on* wywołuje metodę *handleChange*, która ustawia właściwość *dataValue*. Ta zmiana wzywała aktualizację, która jest wyświetlana za pomocą wiązania interpolacji danych (rysunek 15.2).



Rysunek 15.2. Zastosowanie jednokierunkowego wiązania danych

Jednokierunkowe wiązanie danych działa dobrze, gdy elementy formularza stanowią jedyne źródło danych w aplikacji. Sytuacja komplikuje się, gdy użytkownik może wprowadzić zmiany inaczej, np. za pomocą przycisku *Reset*. Jego dyrektywa *v-on* wywołuje metodę *reset*, która ustawia właściwość *dataValue* na *false*. Wiązanie interpolacji danych prawidłowo wyświetla nową wartość, ale element *input* nic nie wie o zmianach i następuje jego desynchronizacja względem reszty strony (rysunek 15.3).



Rysunek 15.3. Ograniczenia wiązania jednokierunkowego

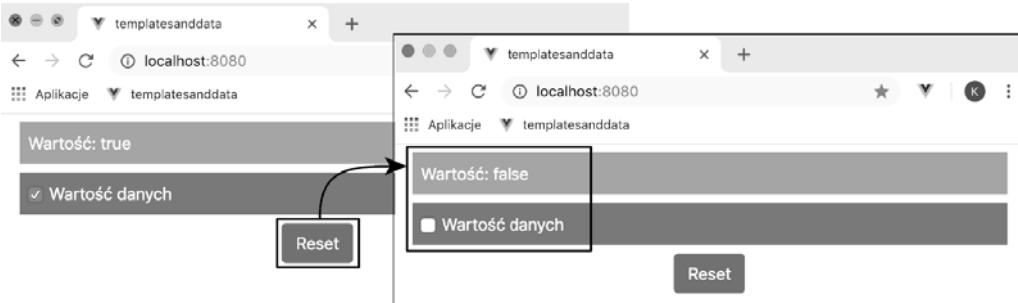
Dodawanie wiązania dwukierunkowego

Elementy formularza potrzebują danych, aby obsłużyć przepływ w obu kierunkach. Dane muszą przepływać z formularza do modelu danych w momencie zmian wprowadzonych w elemencie (np. przy wprowadzaniu tekstu w pole tekstowe lub po zaznaczeniu przycisku wyboru). Dane muszą też przepływać w drugą stronę, gdy model jest modyfikowany inaczej — np. za pomocą przycisku *Reset* — aby zapewnić, że użytkownik zawsze widzi aktualne dane. W listingu 15.3 utworzyłem wiązanie między przyciskiem wyboru a właściwością danych.

Listing 15.3. Tworzenie wiązania w pliku src/App.vue

```
...
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      Wartość: {{ dataValue }}
    </div>
    <div class="bg-primary m-2 p-2 text-white">
      <div class="form-check">
        <label class="form-check-label">
          <input class="form-check-input" type="checkbox"
            v-on:change="handleChange"
            v-bind:checked="dataValue" />
          Wartość danych
        </label>
      </div>
    </div>
    <div class="text-center m-2">
      <button class="btn btn-secondary" v-on:click="reset">
        Reset
      </button>
    </div>
  </div>
</template>
...
```

Skorzystałem z dyrektywy `v-bind`, aby ustawić właściwość `checked` elementu, która zapewnia, że po kliknięciu przycisku `Reset` przycisk wyboru zostanie odznaczony (rysunek 15.4).



Rysunek 15.4. Konsekwencja zastosowania wiązania danych

Dwukierunkowe wiązanie danych istnieje między właściwością `dataValue` a przyciskiem wyboru. Zaznaczenie i odznaczenie przycisku wyboru powoduje wyzwolenie zdarzenia `change` elementu `input`, co prowadzi do wywołania metody `handleChange` przez dyrektywę `v-on`. Finalnie doprowadza to do ustawienia wartości `dataValue`. W drugą stronę, zmiana właściwości `dataValue` wynikająca z kliknięcia przycisku `Reset` powoduje ustawienie atrybutu `checked` elementu `input` przez dyrektywę `v-bind`. Mówiąc krótko, zmienia to stan zaznaczenia przycisku wyboru.

Wiązania dwukierunkowe stanowią podstawę efektywnego użycia formularzy HTML. W aplikacji Vue.js to model danych rządzi aplikacją, dlatego zmiany w nim mogą być widoczne na różne sposoby — każda z nich musi być odzwierciedlona w elementach formularza, które widzi użytkownik.

Dodawanie kolejnego elementu wejściowego

Specyfikacje języka HTML i modelu DOM dla formularzy nie są ze sobą spójne. W związku z tym istniejące różnice muszą być odzwierciedlone w dyrektywach `v-on` i `v-bind` używanych do utworzenia wiązań dwukierunkowych. W listingu 15.4 dodaję pole typu `input`, które pokazuje różnicę między dwoma odrębnymi elementami formularza.

Listing 15.4. Rozszerzanie elementów formularzy w pliku src/App.vue

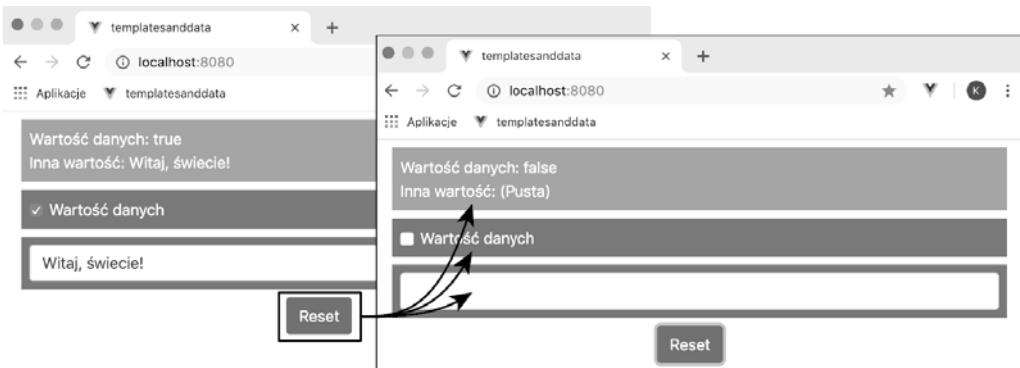
```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Wartość danych: {{ dataValue }}</div>
      <div>Inna wartość: {{ otherValue || "(Pusta)" }}</div>
    </div>
    <div class="bg-primary m-2 p-2 text-white">
      <div class="form-check">
        <label class="form-check-label">
          <input class="form-check-input" type="checkbox"
            v-on:change="handleChange"
            v-bind:checked="dataValue" />
          Wartość danych
        </label>
      </div>
    </div>
    <div class="bg-primary m-2 p-2">
      <input type="text" class="form-control"
        v-on:input="handleChange"
        v-bind:value="otherValue" />
    
```

```

</div>
<div class="text-center m-2">
    <button class="btn btn-secondary" v-on:click="reset">
        Reset
    </button>
</div>
</div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                dataValue: false,
                otherValue: ""
            }
        },
        methods: {
            reset() {
                this.dataValue = false;
                this.otherValue = "";
            },
            handleChange($event) {
                if ($event.target.type == "checkbox") {
                    this.dataValue = $event.target.checked;
                } else {
                    this.otherValue = $event.target.value;
                }
            }
        }
    }
</script>

```

Ten przykład pokazuje różnicę w tworzeniu wiązań dwukierunkowych dla różnych rodzajów elementów. Obsługując przycisk wyboru, muszę nasłuchiwać zdarzenia change i powiązać się z atrybutem checked. W przypadku pola tekstowego muszę nasłuchiwać zdarzenia input, a powiązać się z atrybutem value. Analogiczne zmiany muszę wprowadzić w funkcji handleChange, ustawiając właściwość checked dla przycisku wyboru i value dla pola tekstowego. W rezultacie dysponuję dwoma polami formularza, z których każde jest dwukierunkowo powiązane z właściwością data (rysunek 15.5).



Rysunek 15.5. Dodawanie kolejnego pola wejściowego

Upraszczanie wiązań dwukierunkowych

Różnice, o których trzeba pamiętać podczas tworzenia wiązań, komplikują proces ich konfiguracji. Łatwo pomieszać różne rodzaje elementów, co może prowadzić do powiązania niewłaściwych zdarzeń, atrybutów lub właściwości.

Vue.js udostępnia dyrektywę `v-model`, która upraszcza wiązania dwukierunkowe, automatycznie radząc sobie z różnicami między typami elementów. Ta dyrektywa może być stosowana w elementach `input`, `select` i `textarea`. W listingu 15.5 upraszczam wiązania dzięki dyrektywie `v-model`.

Listing 15.5. Uproszczenie wiązań dwukierunkowych w pliku src/App.vue

```
<template>
<div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
        <div>Wartość danych: {{ dataValue }}</div>
        <div>Inna wartość: {{ otherValue || "(Pusta)" }}</div>
    </div>
    <div class="bg-primary m-2 p-2 text-white">
        <div class="form-check">
            <label class="form-check-label">
                <input class="form-check-input" type="checkbox"
                    v-model="dataValue" />
                Wartość danych
            </label>
        </div>
        <div class="bg-primary m-2 p-2">
            <input type="text" class="form-control" v-model="otherValue" />
        </div>
        <div class="text-center m-2">
            <button class="btn btn-secondary" v-on:click="reset">
                Reset
            </button>
        </div>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                dataValue: false,
                otherValue: ""
            }
        },
        methods: {
            reset() {
                this.dataValue = false;
                this.otherValue = "";
            }
        }
    }
    // handleChange($event) {
    //     if ($event.target.type == "checkbox") {
    //         this.dataValue = $event.target.checked;
    //     } else {
    //         this.otherValue = $event.target.value;
    //     }
    // }
</script>
```

```
// }
}
</script>
```

Wyrażenie w dyrektywie `v-model` to właściwość, dla której są tworzone wiązania dwukierunkowe. Nie trzeba przechwytywać zdarzeń w metodzie, dzięki czemu można usunąć metodę `handleChange` i skupić się na danych komponentu i treści, zamiast na łączących je mechanizmach.

Wiązania z elementami formularzy

Zanim przejdziemy dalej, chciałbym pokazać, jak skorzystać z dyrektywy `v-model`, aby utworzyć wiązania do różnych rodzajów elementów formularzy. Większość różnic między elementami jest obsługiwana przez dyrektywę `v-model`, niemniej prosty zbiór wiązań pozwoli Ci kopiować i wklejać kod we własnych projektach bez konieczności znajdowania właściwego rozwiązania dla każdego z nich.

Wiązania do pól tekstowych

Najprostsze wiązania są tworzone w odniesieniu do pól typu `input`, które pozwalają wprowadzić tekst. W listingu 15.6 skorzystałem z elementów typu `input` do wprowadzenia podstawowych informacji i haseł, wraz z wiązaniem dwukierunkowym zdefiniowanym za pomocą dyrektywy `v-model`.

Listing 15.6. Wiązania do pól tekstowych w pliku src/App.vue

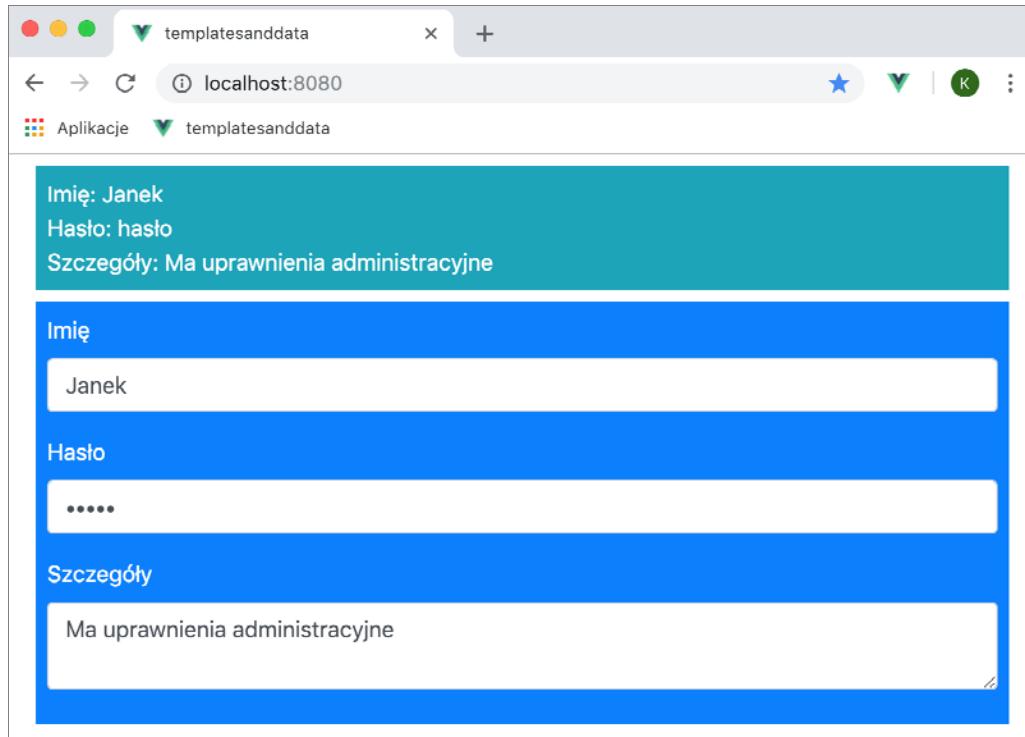
```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Imię: {{ name }}</div>
      <div>Hasło: {{ password }}</div>
      <div>Szczegóły: {{ details }}</div>
    </div>
    <div class="bg-primary m-2 p-2 text-white">
      <div class="form-group">
        <label>Imię</label>
        <input class="form-control" v-model="name" />
      </div>
      <div class="form-group">
        <label>Hasło</label>
        <input type="password" class="form-control" v-model="password" />
      </div>
      <div class="form-group">
        <label>Szczegóły</label>
        <textarea class="form-control" v-model="details" />
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Janek",
        password: "hasło",
        details: "Ma uprawnienia administracyjne"
      }
    }
  }
</script>
```

```

        }
    }
</script>

```

Dodałem dwa pola typu `input`, jedno będące zwykłym polem tekstowym, a drugie — polem hasła. Oprócz tego umieściłem element `textarea`. Wszystkie trzy elementy korzystają z dyrektywy `v-model`, aby utworzyć wiązanie dwukierunkowe z właściwościami danych zdefiniowanymi w komponencie (rysunek 15.6).



Rysunek 15.6. Wiązanie do pól tekstowych

Wiązania do przycisków opcji i wyboru

W listingu 15.7 zastąpiłem elementy z poprzedniego przykładu przyciskami opcji i wyboru, które ograniczają wybór użytkownika do kilku opcji. Również i w tym przykładzie korzystam z dyrektywy `v-model` do utworzenia wiązania dwukierunkowego z wiązaniem danych.

Listing 15.7. Wiązanie z przyciskami opcji i wyboru (src/App.vue)

```

<template>
    <div class="container-fluid">
        <div class="bg-info m-2 p-2 text-white">
            <div>Imię: {{ name }} </div>
            <div>Czy ma dostęp administracyjny: {{ hasAdminAccess }}</div>
        </div>
        <div class="bg-primary m-2 p-2 text-white">
            <div class="form-check">

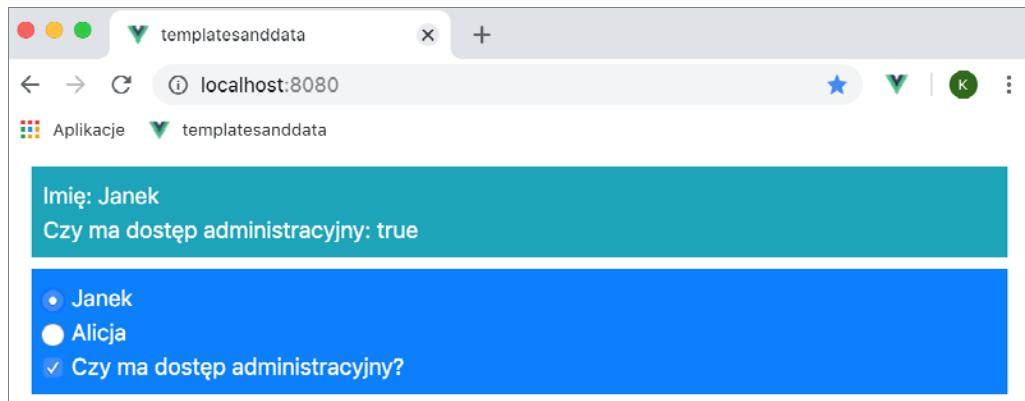
```

```

<input class="form-check-input" type="radio"
       v-model="name" value="Janek" />
<label class="form-check-label">Janek</label>
</div>
<div class="form-check">
    <input class="form-check-input" type="radio"
          v-model="name" value="Alicja" />
    <label class="form-check-label">Alicja</label>
</div>
<div class="form-check">
    <input class="form-check-input" type="checkbox"
          v-model="hasAdminAccess" />
    <label class="form-check-label">Czy ma dostęp administracyjny?</label>
</div>
</div>
</div>
</template>
<script>
export default {
    name: "MyComponent",
    data: function () {
        return {
            name: "Janek",
            hasAdminAccess: true
        }
    }
}
</script>

```

Kluczowa różnica polega na tym, że w przypadku przycisków opcji każdy element musi otrzymać atrybut value, dzięki czemu dyrektywa v-model „wie”, jak zaktualizować właściwość danych. Elementy z listingu 15.7 dają efekt jak na rysunku 15.7.



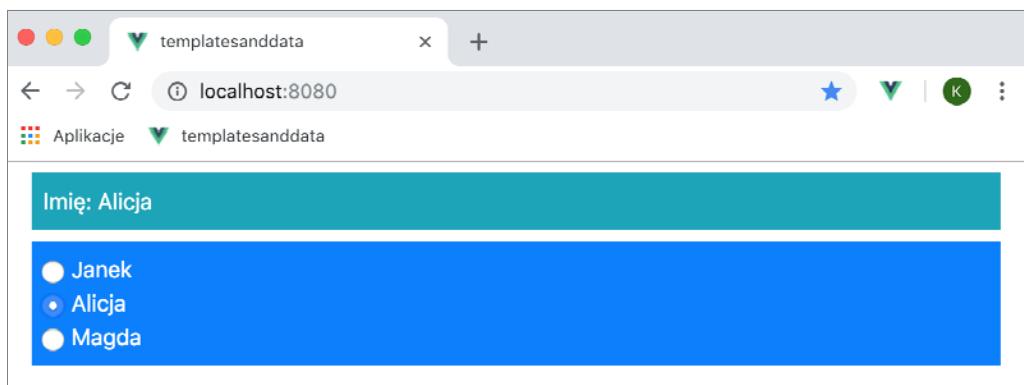
Rysunek 15.7. Wiązanie z przyciskami opcji i wyboru

Dyrektyna v-model może być połączona z dyrektywami v-for i v-bind w celu tworzenia elementów formularzy z tablic wartości (listing 15.8), co jest niezwykle przydatne, gdy opcje do wyboru są generowane w czasie działania aplikacji.

Listing 15.8. Generowanie przycisków opcji w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Imię: {{ name }} </div>
    </div>
    <div class="bg-primary m-2 p-2 text-white">
      <div class="form-check" v-for="n in allNames" v-bind:key="n">
        <input class="form-check-input" type="radio"
          v-model="name" v-bind:value="n" />
        <label class="form-check-label">>{{ n }}</label>
      </div>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        allNames: ["Janek", "Alicja", "Magda"],
        name: "Janek"
      }
    }
  }
</script>
```

Dyrektywa `v-bind` musi zostać użyta w celu ustawienia atrybutu `value` elementów `input`. W przeciwnym razie Vue.js nie potraktuje wartości atrybutu jako wyrażenia. Rysunek 15.8 przedstawia wynik działania kodu z listingu 15.8.

**Rysunek 15.8.** Generowanie elementów formularzy z wartością danych

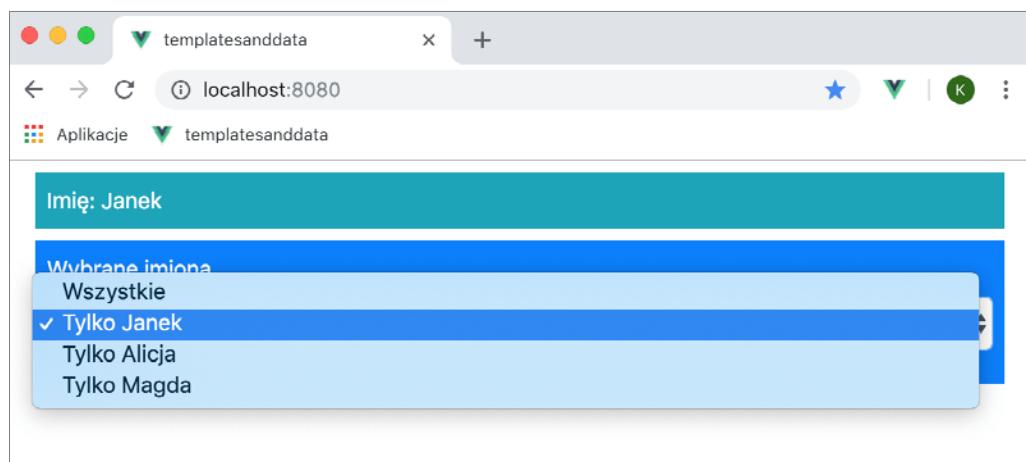
Wiązania do elementów typu select

Elementy typu `select` pozwalają na wyświetlanie ograniczonego zbioru opcji w zwięzły sposób (listing 15.9). Elementy `option` definiują dostępne opcje. Można określić je statycznie, za pomocą dyrektywy `v-for` lub — jak pokazano w listingu — łącząc oba podejścia.

Listing 15.9. Wiązanie z elementem select (src/App.vue)

```
<template>
    <div class="container-fluid">
        <div class="bg-info m-2 p-2 text-white">
            <div>Imię: {{ name }} </div>
        </div>
        <div class="bg-primary m-2 p-2 text-white">
            <div class="form-group">
                <label>Wybrane imiona</label>
                <select class="form-control" v-model="name">
                    <option value="all">Wszystkie</option>
                    <option v-for="n in allNames" v-bind:key="n"
                           v-bind:value="n">Tylko {{ n }}</option>
                </select>
            </div>
        </div>
    </div>
</template>
<script>
    export default {
        name: "MyComponent",
        data: function () {
            return {
                allNames: ["Janek", "Alicja", "Magda"],
                name: "Janek"
            }
        }
    }
</script>
```

Podobnie jak w przypadku przycisków opcji dyrektywa v-bind musi być zastosowana do ustawienia atrybutu value. Rysunek 15.9 przedstawia wynik działania listingu 15.9.

**Rysunek 15.9.** Wiązanie z elementem typu select

Stosowanie modyfikatorów dyrektywy v-model

Dyrektyna v-model dostarcza trzy modyfikatory, które zmieniają tworzone przez nią wiązania dwukierunkowe. Modyfikatory te są opisane w tabeli 15.3 i dalszej części rozdziału.

Tabela 15.3. Modyfikatory dyrektywy v-model

Modyfikator	Opis
number	Ten modyfikator parsuje wartość z pola wejściowego i przetwarza ją na liczbę przed przypisaniem do właściwości danych.
trim	Ten modyfikator usuwa wszelkie wiodące i kończące białe znaki przed przypisaniem do właściwości danych.
lazy	Ten modyfikator zmienia zdarzenie nasłuchiwanie przez dyrektywę v-model, przez co właściwość danych jest zmieniana tylko, jeśli użytkownik opuści dany element wejściowy.

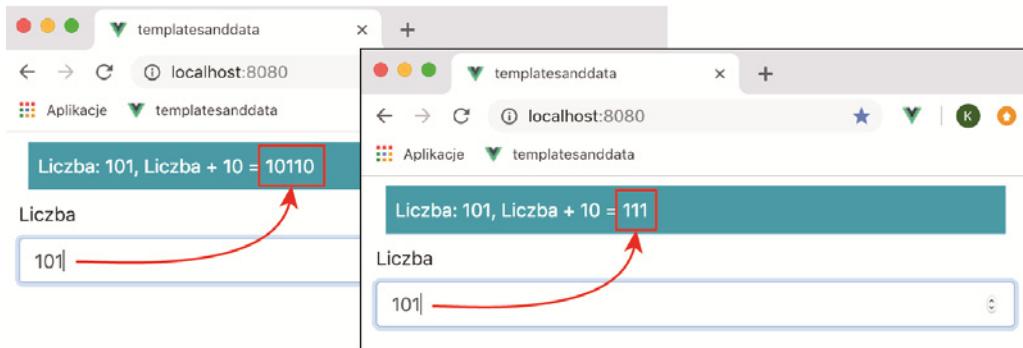
Formatowanie wartości jako liczb

Modyfikator number usprawnia zachowanie elementów typu input w sytuacji, gdy typem atrybutu jest number (listing 15.10).

Listing 15.10. Zastosowanie liczbowego elementu typu input (src/App.vue)

```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Liczba: {{ amount }}, Liczba + 10 = {{ amount + 10 }}</div>
    </div>
    <div class="form-group">
      <label>Liczba</label>
      <input type="number" class="form-control" v-model="amount" />
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        amount: 100
      }
    }
  }
</script>
```

Specyfikacja HTML5 zezwala na podanie jednej z kilku wartości dla atrybutu type, a wartość number oznacza, że przeglądarka zaakceptuje tylko cyfry i znak separatora dziesiętnego. Z drugiej strony wartość wprowadzona przez użytkownika jest łańcuchem znaków. Połączenie tego faktu z dynamicznym systemem typów w JavaScriptie spowoduje problem, który można zobaczyć, zmieniając wartość w polu *Liczba*, np. na 101. Dyrektywa v-model zareaguje, aktualizując właściwość danych o nazwie *amount* za pomocą otrzymanej wartości tekstowej (rysunek 15.10).



Rysunek 15.10. Efekt zastosowania modyfikatora number

Problem tkwi w wiązaniu interpolacji tekstu, które dodaje 10 do wartości amount. Skoro dyrektywa v-model przekazała wartość tekstową, JavaScript zinterpretuje wyrażenie jako konkatenację łańcucha znaków, łącząc ze sobą ciągi znaków 101 i 10 w 10110. W listingu 15.11 stosuję modyfikator number w dyrektywie v-model, likwidując problem.

Listing 15.11. Zastosowanie modyfikatora w pliku src/App.vue

```
...
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Liczba {{ amount }}, Liczba + 10 = {{ amount + 10 }}</div>
    </div>
    <div class="form-group">
      <label> Liczba </label>
      <input type="number" class="form-control" v-model.number="amount" />
    </div>
  </div>
</template>
...
```

Modyfikator number nakazuje dyrektywie v-model przekonwertować wartość wprowadzoną przez użytkownika na liczbę przed przekazaniem jej dalej.

-
- **Ostrzeżenie** Modyfikator number nie ogranicza zakresu znaków wprowadzanych w polu input — jeśli użytkownik wprowadzi do tego pola wartość nieliczbową, model danych zostanie zaktualizowany za pomocą wartości tekstowej. Korzystając z tego modyfikatora, musisz się upewnić, że użytkownik może wprowadzić jedynie cyfry, zmieniając atrybut type elementu input na number.
-

Opóźnianie aktualizacji

Domyślnie dyrektywa v-model aktualizuje model danych po każdym wcisnięciu klawisza w elementach takich jak input czy textarea. Modyfikator lazy zmienia zdarzenie, na które reaguje dyrektywa v-model — aktualizacja zostanie wykonana tylko wtedy, gdy zmieni się aktywny element. W listingu 15.12 stosuję modyfikator do elementu input w szablonie komponentu.

Listing 15.12. Zastosowanie modyfikatora w pliku src/App.vue

```
...
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div> Liczba: {{ amount }}, Liczba + 10 = {{ amount + 10 }}</div>
    </div>
    <div class="form-group">
      <label> Liczba </label>
      <input type="number" class="form-control" v-model.number.lazy="amount" />
    </div>
  </div>
</template>
...
```

Modyfikator `lazy` zapobiegnie aktualizacji właściwości `amount`, zanim element `input` nie straci fokusu, czyli zanim użytkownik nie przejdzie do innej kontrolki lub nie kliknie przycisku myszy poza elementem.

Usuwanie białych znaków

Modyfikator `trim` usuwa wiodące i końcowe białe znaki z tekstu wprowadzonego przez użytkownika oraz pomaga uniknąć błędów walidacji, które są trudne do znalezienia przez użytkownika. W listingu 15.13 dodaję element typu `input` do komponentu i stosuję modyfikator `trim` w dyrektywie `v-model`.

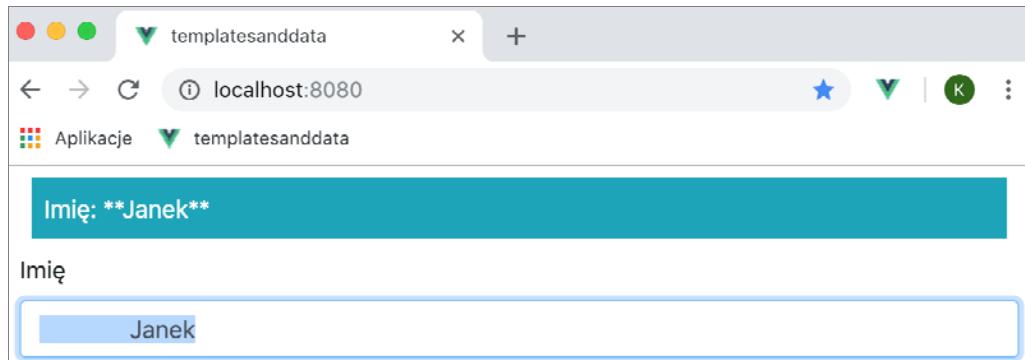
Listing 15.13. Obcinanie białych znaków (src/App.vue)

```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Imię: **{{name}}** </div>
    </div>
    <div class="form-group">
      <label>Imię</label>
      <input type="text" class="form-control" v-model.trim="name" />
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        name: "Janek"
      }
    }
  }
</script>
```

■ **Uwaga** Modyfikator `trim` dotyczy tylko wartości wprowadzonych przez użytkownika. Jeśli w innej części aplikacji ustawisz właściwość danych na tekst, który zawiera wiodące bądź końcowe białe znaki, zostaną one wyświetcone w elemencie `input`.

Wiązanie interpolacji tekstu otoczyłem znakami gwiazdki, aby łatwiej było zauważać wiodące lub końcowe białe znaki. Aby sprawdzić działanie modyfikatora, zapisz zmiany w komponencie i wprowadź

łańcuch znaków, który zaczyna się lub kończy spacjami. Znaki zostaną wycięte z tekstu przypisanego do właściwości danych, co da efekt jak na rysunku 15.11.



Rysunek 15.11. Usuwanie białych znaków

Wiązania do różnych typów danych

Dyrektywa `v-model` potrafi dostosować sposób, w jaki łączy się ona z modelem danych. Dzięki temu jesteśmy w stanie skorzystać z elementów formularzy tak, aby było to użyteczne w procesie tworzenia aplikacji webowej — opiszę to szerzej już za chwilę.

Wybór tablicy elementów

Jeśli dyrektywa `v-model` jest zastosowana wobec przycisku wyboru, a jednocześnie jest powiązana z właściwością danych będącą tablicą, zaznaczenie i odznaczenie pola spowoduje dodanie wartości do tablicy bądź usunięcie jej z tablicy. Zdecydowanie łatwiej pokazać ten mechanizm w praktyce — w listingu 15.14 zastosowałem dyrektywę `v-for` do wygenerowania zbioru przycisków wyboru, a dzięki dyrektywie `v-model` mogłem powiązać je z tablicą.

Listing 15.14. Tworzenie wiązania do tablicy (src/App.vue)

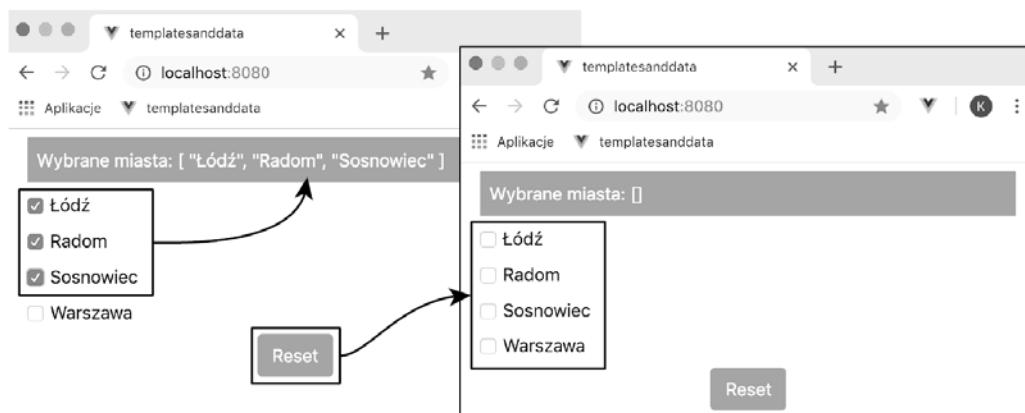
```
<template>
  <div class="container-fluid">
    <div class="bg-info m-2 p-2 text-white">
      <div>Wybrane miasta: {{ cities }}</div>
    </div>
    <div class="form-check m-2" v-for="city in cityNames" v-bind:key="city">
      <label class="form-check-label">
        <input type="checkbox" class="form-check-input"
          v-model="cities" v-bind:value="city" />
        {{city}}
      </label>
    </div>
    <div class="text-center">
      <button v-on:click="reset" class="btn btn-info">Reset</button>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data() {
      return {
        cities: [],
        cityNames: ["Kraków", "Gdańsk", "Szczecin", "Wrocław", "Poznań", "Łódź", "Katowice", "Bielsko-Biała", "Gliwice", "Częstochowa", "Dąbrowa Górnicza", "Kielce", "Rzeszów", "Tarnów", "Bielsko-Biała", "Gliwice", "Częstochowa", "Dąbrowa Górnicza", "Kielce", "Rzeszów", "Tarnów"]
      }
    },
    methods: {
      reset() {
        this.cities = []
      }
    }
  }
</script>
```

```

data: function () {
    return {
        cityNames: ["Łódź", "Radom", "Sosnowiec", "Warszawa"],
        cities: []
    }
},
methods: {
    reset() {
        this.cities = [];
    }
}
}
</script>

```

Dyrektywa `v-for` tworzy przycisk wyboru dla każdej wartości z tablicy `cityName`. To spośród wartości tablicy użytkownik może wybierać. Każdy element `input` jest konfigurowany za pomocą atrybutu `value`, który określa nazwę miasta. Zaznaczenie i odznamczenie danego pola spowoduje dodanie miasta do tablicy bądź usunięcia go z niej. Dyrektywa `v-model` tworzy wiązanie dwukierunkowe z tablicą `cities`, wypełnioną wartościami z zaznaczonych pól. Wiązanie dwukierunkowe oznacza, że zmiana w tablicy wprowadzona w innej części aplikacji — np. w metodzie `reset` — spowoduje automatyczne odznamczenie pola. Wartości z tablicy wyświetlam za pomocą wiązania interpolacji tekstu (rysunek 15.12).



Rysunek 15.12. Tworzenie wiązania danych z tablicą

Wiązanie z tablicą w elemencie select

Podobny efekt można uzyskać, stosując element `select` z włączoną opcją zaznaczenia wielu elementów (`multiple`).

```

...
<div class="form-control">
    <label>Miasto</label>
    <select multiple class="form-control" v-model="cities">
        <option v-for="city in cityNames" v-bind:key="city">
            {{city}}
        </option>
    </select>
</div>
...

```

Dyrektywa `v-for` jest stosowana wobec elementu `option` i wypełnia element `select` listą dostępnych opcji. Dyrektywa `v-model` jest powiązana z tablicą w ten sam sposób co grupy przycisków wyboru z listingu 15.14.

Moim zdaniem lepiej jest skorzystać z przycisków wyboru, ponieważ liczni użytkownicy nie wiedzą, że wybór wielu opcji z listy wymaga wcisnięcia klawisza modyfikatora (np. `Shift` lub `Ctrl` w systemie Windows), aby wykonywać ciągle lub nieciągłe zaznaczenia. Zbiór przycisków wyboru jest znacznie łatwiejszy w użyciu niż wyświetlenie elementu `select` z załączoną instrukcją.

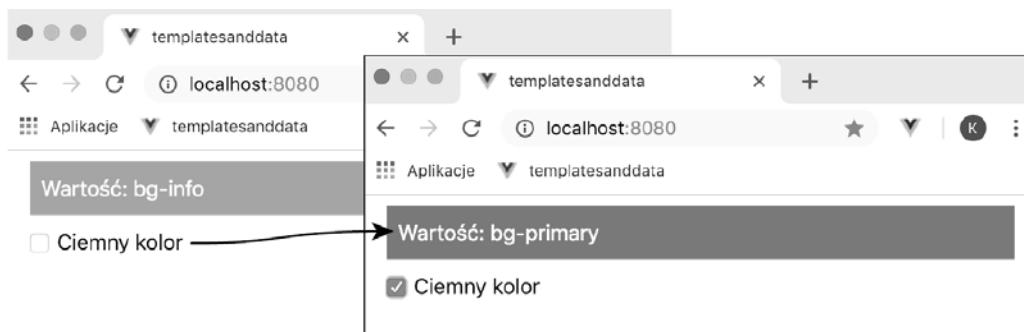
Stosowanie własnych wartości w elementach formularza

Typowym przykładem użycia przycisku wyboru jest jego pośredni wpływ na pewien aspekt aplikacji — wartość logiczna `true/false` dostarczona przez element `input` i wiązanie `v-model` jest przekształcana na inne wartości, wynikające z właściwości obliczanych lub wyrażeń wiązania danych. W listingu 15.15 przedstawiam takie zachowanie, stosując dyrektywę `v-bind` w celu przypisania bootstrapowych klas do elementu.

Listing 15.15. Pośrednie wartości w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="m-2 p-2 text-white" v-bind:class="elemClass">
      <div>Wartość: {{ elemClass }}</div>
    </div>
    <div class="form-check m-2">
      <label class="form-check-label">
        <input type="checkbox" class="form-check-input"
          v-model="dataValue" />
        Ciemny kolor
      </label>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        dataValue: false,
      }
    },
    computed: {
      elemClass() {
        return this.dataValue ? "bg-primary" : "bg-info";
      }
    }
  }
</script>
```

Dyrektywa `v-model` powiązana z przyciskiem wyboru ustawia właściwość danych `dataValue`, używaną wyłącznie we właściwości obliczanej `elemClass`. Za pomocą tej właściwości dyrektywa `v-bind` w elemencie `div` ustawia przynależność do klas. W rezultacie zmiana stanu przycisku wyboru spowoduje zmianę między klasami `bg-primary` i `bg-info`, co w konsekwencji spowoduje ustawienie koloru tła (rysunek 15.13).



Rysunek 15.13. Zastosowanie wartości pośredniej za pomocą przycisku wyboru

Ustawienie właściwości danych za pomocą przycisku wyboru pozwala na pominięcie właściwości obliczanej przez zastosowanie atrybutów `true-value` i `false-value` (listing 15.16).

Listing 15.16. Uproszczenie wiązań w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="m-2 p-2 text-white" v-bind:class="dataValue">
      <div>Wartość: {{ dataValue }}</div>
    </div>
    <div class="form-check m-2">
      <label class="form-check-label">
        <input type="checkbox" class="form-check-input"
          v-model="dataValue" true-value="bg-primary"
          false-value="bg-info" />
        Ciemny kolor
      </label>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        dataValue: "bg-info"
      }
    }
  }
</script>
```

Atrybuty `true-value` i `false-value` są używane w dyrektywie `v-model`, aby ustawić wartość właściwości `dataValue`. W ten sposób pomijamy konieczność zastosowania właściwości obliczanej — nie musimy przekształcać wyboru użytkownika na nazwę klasy. Efekt pozostaje bez zmian, ale kod staje się czytelniejszy.

Stosowanie własnych wartości dla przycisków opcji i elementu select

Dyrektyna `v-model` może być połączona z dyrektywą `v-bind`, aby obsługiwać wartości, które mogą być używane w przyciskach opcji i elementach `select`. W listingu 15.17 dodaję oba rodzaje elementów i konfiguruje je w taki sposób, aby korzystały z nazw klas wprowadzonych w poprzednim punkcie.

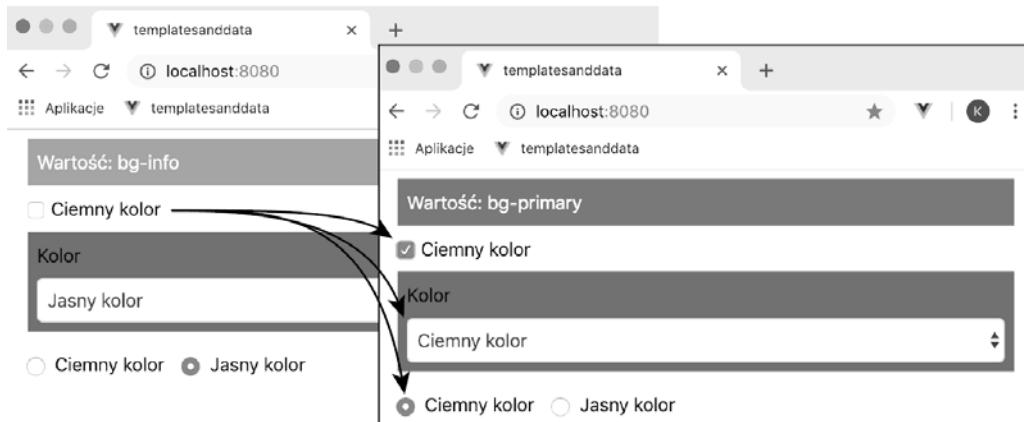
Listing 15.17. Stosowanie własnych wartości w innych elementach (src/App.vue)

```

<template>
  <div class="container-fluid">
    <div class="m-2 p-2 text-white" v-bind:class="dataValue">
      <div>Wartość: {{ dataValue }}</div>
    </div>
    <div class="form-check m-2">
      <label class="form-check-label">
        <input type="checkbox" class="form-check-input"
          v-model="dataValue" v-bind:true-value="darkColor"
          v-bind:false-value="lightColor" />
        Ciemny kolor
      </label>
    </div>
    <div class="form-group m-2 p-2 bg-secondary">
      <label>Kolor</label>
      <select v-model="dataValue" class="form-control">
        <option v-bind:value="darkColor">Ciemny kolor</option>
        <option v-bind:value="lightColor">Jasny kolor</option>
      </select>
    </div>
    <div class="form-check-inline m-2">
      <label class="form-check-label">
        <input type="radio" class="form-check-input"
          v-model="dataValue" v-bind:value="darkColor" />
        Ciemny kolor
      </label>
    </div>
    <div class="form-check-inline m-2">
      <label class="form-check-label">
        <input type="radio" class="form-check-input"
          v-model="dataValue" v-bind:value="lightColor" />
        Jasny kolor
      </label>
    </div>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
      return {
        darkColor: "bg-primary",
        lightColor: "bg-info",
        dataValue: "bg-info"
      }
    }
  }
</script>

```

Aby uniknąć zduplikowania nazw klas w elementach, korzystam z dyrektywy `v-bind` — ustawiam atrybut `value` w elementach `input` i `option` za pomocą właściwości danych `darkColor` i `lightColor`. W rezultacie otrzymuję zbiór elementów formularzy, które zarządzają przynależnością do klas CSS elementu `div` bez konieczności stosowania właściwości obliczanych do konwertowania wartości `true/false` (rysunek 15.14).



Rysunek 15.14. Zastosowanie własnych wartości dla innych elementów

Walidacja danych w formularzu

Gdy tylko zaczniesz korzystać z elementów formularzy, musisz pomyśleć również o kwestii walidacji danych. Użytkownicy wprowadzają w pola tekstowe przeróżne dane, przez co musisz się upewnić, że aplikacja otrzymuje dane w odpowiedniej postaci. W tym podrozdziale omawiamy metody walidacji danych. W związku z tym zaktualizowałem treść komponentu, przez co zawiera on jedynie prosty formularz (listing 15.18).

Listing 15.18. Tworzenie formularza w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="btn-primary text-white my-2 p-2">
      Nazwa: {{ name }}, Kategoria: {{ category }}, Cena: {{ price }}
    </div>
    <form v-on:submit.prevent="handleSubmit">
      <div class="form-group">
        <label>Nazwa</label>
        <input v-model="name" class="form-control" />
      </div>
      <div class="form-group">
        <label>Kategoria</label>
        <input v-model="category" class="form-control" />
      </div>
      <div class="form-group">
        <label>Cena</label>
        <input type="number" v-model.number="price" class="form-control" />
      </div>
      <div class="text-center">
        <button class="btn btn-primary" type="submit">Wyślij</button>
      </div>
    </form>
  </div>
</template>
<script>
  export default {
    name: "MyComponent",
    data: function () {
```

```

        return {
            name: "",
            category: "",
            price: 0
        }
    },
methods: {
    handleSubmit() {
        console.log(`WYSŁANO FORMULARZ: ${this.name} ${this.category} + ${this.price}`);
    }
}
</script>

```

- **Uwaga** W tym miejscu przedstawiam metody tworzenia własnego kodu walidacji, ponieważ uważam to za interesujące zagadnienie. Pokazuję także, jak można połączyć mechanizmy omawiane w tym i w poprzednich rozdziałach, aby uzyskać ciekawe efekty. W prawdziwych projektach gorąco zachęcam do skorzystania z jednego z wielu popularnych, otwartych pakietów przeznaczonych do walidacji formularzy, np. `vuelidate` (<https://github.com/monterail/vuelidate>), z którego korzystałem w części I, w aplikacji *Sklep sportowy*, lub `VeeValidate` (<http://vee-validate.logaretm.com>).

Komponent definiuje właściwości danych o nazwach `name`, `category` i `price`, które będą widoczne za pośrednictwem pól typu `input` (opatrzonych dyrektywą `v-model`). Szczegółowe informacje na temat wprowadzonych wartości są wyświetlane ponad elementami formularza (rysunek 15.15).

Rysunek 15.15. Widok formularza

Elementy `input` są zawarte w elemencie `form`, do którego zastosowaliśmy następującą dyrektywę `v-on`:

```

...
<form v-on:submit.prevent="handleSubmit">
...

```

Zdarzenie submit jest wyzwalane w momencie kliknięcia przez użytkownika przycisku typu submit. Modyfikator prevent jest konieczny, aby powstrzymać przeglądarkę od wysłania formularza do serwera HTTP, co stanowi domyślną akcję dla tego zdarzenia. Wyrażenie dyrektywy wywołuje metodę handleSubmit, która w prawdziwej aplikacji wysłałaby dane do usługi sieciowej, jednak tutaj tylko wypisuje komunikat w konsoli przeglądarki JavaScript. Kliknij przycisk *Wyślij*, a zobaczysz komunikat podobny do poniższego:

```
...
WYSŁANO FORMULARZ: Buty do biegania Bieganie 100
...
```

Obsługę usług sieciowych omówiono w rozdziale 19. W tym rozdziale skupiam się na danych wprowadzonych przez użytkownika. Celem w tej części rozdziału jest kontrola komunikatu wyświetlanego w metodzie handleSubmit; będzie on wyświetlany tylko, jeśli użytkownik wprowadzi poprawne dane we wszystkich polach formularza. W trakcie wykonywania validacji trzeba mieć jasno sprecyzowany zbiór wymagań do sprawdzenia. W tym przykładzie wymagania validacji dla każdej właściwości są opisane w tabeli 15.4.

Tabela 15.4. Wymagania validacji

Nazwa	Opis
name	Użytkownik musi wprowadzić tekst nie krótszy niż trzy znaki.
category	Użytkownik musi wprowadzić tekst składający się z samych liter.
price	Użytkownik musi wprowadzić wartość składającą się z samych cyfr o wartości między 1 a 1000.

Definiowanie reguł validacji

Walidując dane, przekonasz się, że te same reguły są stosowane raz po raz dla różnych danych wprowadzonych przez użytkownika. Aby zmniejszyć duplikację kodu, warto zdefiniować każdą regułę raz, a potem tylko się do niej odwoływać. W celu zdefiniowania reguł validacji opisanych w tabeli 15.4 dodaję plik JavaScript o nazwie validationRules.js do katalogu src (listing 15.19).

Listing 15.19. Zawartość pliku src/validationRules.js

```
function required(name) {
    return {
        validator: (value) => value != "" && value !== undefined && value !== null,
        message: `Pole ${name} nie może być puste`
    }
}
function minLength(name, minlength) {
    return {
        validator: (value) => String(value).length >= minlength,
        message: `Pole ${name} wymaga wprowadzenia minimum ${minlength} znaków`
    }
}
function alpha(name) {
    return {
        validator: (value) => /^[a-zA-Z]*$/.test(value),
        message: `Pole ${name} może zawierać tylko litery`
    }
}
function numeric(name) {
    return {
        validator: (value) => /^[0-9]*$/.test(value),
        message: `Pole ${name} może zawierać tylko cyfry`
    }
}
```

```

        }
    }
    function range(name, min, max) {
        return {
            validator: (value) => value >= min && value <= max,
            message: `Pole ${name} może zawierać wartość z zakresu między ${min} i ${max}`
        }
    }
    export default {
        name: [minLength("Nazwa", 3)],
        category: [required("Kategoria"), alpha("Kategoria")],
        price: [numeric("Cena"), range("Cena", 1, 1000)]
    }
}

```

Dlaczego użytkownicy wprowadzają niepoprawne dane?

Istnieje wiele powodów, dla których użytkownicy wprowadzają niepoprawne dane w aplikacji, przez co musimy stosować reguły walidacji. Przede wszystkim użytkownik może nie zrozumieć, czego się od niego oczekuje. Typowy przykład stanowi pole do wprowadzania numeru karty ze spacją między grupami cyfr lub bez spacji. Oczekiwany format powinien być jasno wyrażony. Możesz też rozwiązać problem, akceptując różne formaty i konwertując je na wybrany przez siebie.

Kolejny rodzaj błędów wynika z braku zainteresowania użytkownika procesem wprowadzania danych — jego celem jest otrzymanie wyniku. Jeśli zmusisz użytkownika do wypełniania długich i złożonych formularzy bądź do powtarzania wielokrotnie tego samego procesu, możesz otrzymać dane niemające zbytniego sensu. Aby zminimalizować ten rodzaj problemów, proś użytkownika wyłącznie o niezbędne dane, zapewniając rozsądne wartości domyślne tam, gdzie to możliwe. Nie zapominaj, że rzadko który użytkownik będzie przywiązywał do Twojej aplikacji tak, jakbyś sobie tego życzył.

Wreszcie, niektórzy użytkownicy będą chcieli obejść zabezpieczenia w Twojej aplikacji. Zawsze będą istnieli komputerowi eksperci szukający luk w Twoim oprogramowaniu, a im bardziej wartościowe i popularne będą Twoje produkty, tym więcej takich osób przyciągniesz. Walidacja danych sama w sobie w tym przypadku nie wystarczy — konieczna jest kompleksowa i całościowa analiza bezpieczeństwa, monitoring i odpowiednie działania prewencyjne.

Przedstawiony plik zawiera funkcje walidacji, z których będę korzystać w celu sprawdzania wartości wprowadzonych przez użytkowników. Zaaplikuję kombinacje tych funkcji do walidowanych przez mnie właściwości.

Stosowanie funkcji walidacji

Kolejny krok polega na dodaniu kodu, który wykona walidację. W listingu 15.20 dodaję metody i właściwości niezbędne do zastosowania reguł walidacji, które zdefiniujemy za chwilę.

Listing 15.20. Wykonywanie walidacji (*src/App.vue*)

```

<template>
    <div class="container-fluid">
        <div class="bg-danger text-white my-2 p-2" v-if="errors">
            <h5>Znaleziono następujące problemy:</h5>
            <ul>
                <template v-for="(error, index) in validationErrors" :key="index">
                    <li>${error}</li>
                </template>
            </ul>
        </div>
    </div>

```

```

        </ul>
    </div>
<form v-on:submit.prevent="handleSubmit">
    <div class="form-group">
        <label>Nazwa</label>
        <input v-model="name" class="form-control" />
    </div>
    <div class="form-group">
        <label>Kategoria</label>
        <input v-model="category" class="form-control" />
    </div>
    <div class="form-group">
        <label>Cena</label>
        <input type="number" v-model.number="price" class="form-control" />
    </div>
    <div class="text-center">
        <button class="btn btn-primary" type="submit">
            Wyślij
        </button>
    </div>
</form>
</div>
</template>
<script>
    import validation from "./validationRules";
    import Vue from "vue";
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "",
                category: "",
                price: 0,
                validationErrors: {},
            }
        },
        computed: {
            errors() {
                return Object.values(this.validationErrors).length > 0;
            }
        },
        methods: {
            validate(propertyName, value) {
                let errors = [];
                Object(validation)[propertyName].forEach(v => {
                    if (!v.validator(value)) {
                        errors.push(v.message);
                    }
                });
                if (errors.length > 0) {
                    Vue.set(this.validationErrors, propertyName, errors);
                } else {
                    Vue.delete(this.validationErrors, propertyName);
                }
            },
            validateAll() {
                this.validate("name", this.name);
                this.validate("category", this.category);
            }
        }
    }
</script>

```

```

        this.validate("price", this.price);
        return this.errors;
    },
    handleSubmit() {
        if (this.validateAll()) {
            console.log(`WYSŁANO FORMULARZ: ${this.name} ${this.category} ` +
                `+ ${this.price}`);
        }
    }
}
</script>

```

Metoda validate jest używana do walidacji pojedynczych właściwości danych za pomocą reguł zdefiniowanych w listingu 15.20, dostępnych dzięki zastosowaniu instrukcji import. Szczegóły dotyczące błędów walidacji będą przechowywane we właściwości danych o nazwie validationErrors, w której każde pole input będzie zawierać powiązaną właściwość, a której wartością będzie tablica komunikatów walidacji powiązana z danym polem.

-
- **Wskazówka** Metoda Vue.delete, którą stosuję w listingu 15.20, stanowi przeciwnieństwo metody Vue.set opisanej w rozdziale 13. Wykorzystuję ją do usunięcia właściwości z obiektu w taki sposób, aby Vue.js odnotowało tę zmianę.
-

Gdy użytkownik kliką przycisk *Wyślij*, metoda handleSubmit wywołuje metodę validateAll, co w konsekwencji prowadzi do wykonania metody validate dla każdej właściwości danych. Pożądana akcja, czyli wyświetlenie komunikatu, odbywa się tylko, jeśli nie ma problemów z walidacją.

W szablonie komponentu korzystam z dyrektywy v-if, aby kontrolować widoczność elementu div. Dalej stosuję dyrektywę v-for, aby wyliczyć właściwości obiektu validationErrors, wyświetlając element li dla każdego komunikatu.

Efektem prac jest mechanizm, który sprawdzi wprowadzone przez użytkownika informacje i wyświetli błędy w razie problemów, nie zmieniając oryginalnych wartości pól (rysunek 15.16).

Rysunek 15.16. Walidacja danych formularza

Bieżące reagowanie na zmiany

Podstawowy mechanizm walidacji działa, jednak walidacja jest wykonywana dopiero w momencie kliknięcia przycisku *Wyślij*. Zdecydowanie lepiej byłoby reagować na błędy już w trakcie wypełniania pól, dając informację zwrotną znacznie szybciej. Nie chcemy rzeczy jasna rozpoczynać od wyświetlania błędów, dlatego poczekamy do pierwszego kliknięcia przycisku *Wyślij*, jak w listingu 15.21.

Listing 15.21. Reagowanie na zmiany w pliku src/App.vue

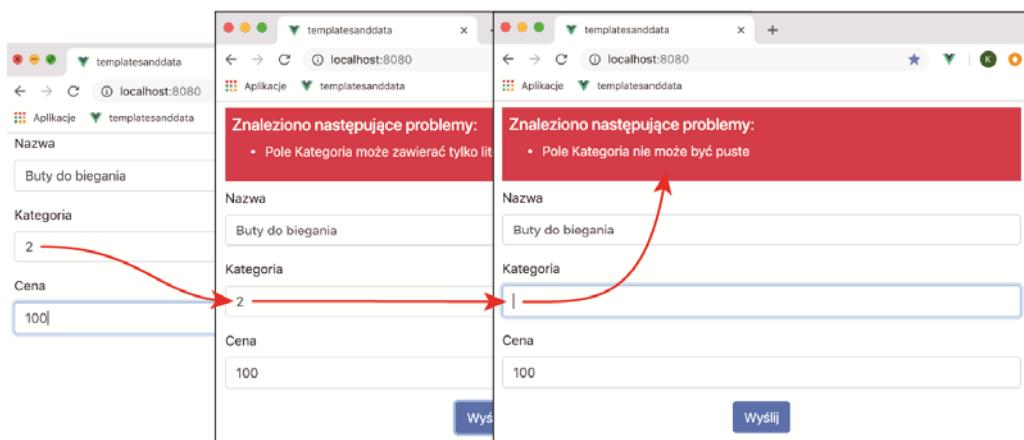
```
...
<script>
    import validation from "./validationRules";
    import Vue from "vue";
    export default {
        name: "MyComponent",
        data: function () {
            return {
                name: "",
                category: "",
                price: 0,
                validationErrors: {},
                hasSubmitted: false
            }
        },
        watch: {
            name(value) { this.validateWatch("name", value) },
            category(value) { this.validateWatch("category", value) },
            price(value) { this.validateWatch("price", value) }
        },
        computed: {
            errors() {
                return Object.values(this.validationErrors).length > 0;
            }
        },
        methods: {
            validateWatch(propertyName, value) {
                if (this.hasSubmitted) {
                    this.validate(propertyName, value);
                }
            },
            validate(propertyName, value) {
                let errors = [];
                Object(validation)[propertyName].forEach(v => {
                    if (!v.validator(value)) {
                        errors.push(v.message);
                    }
                });
                if (errors.length > 0) {
                    Vue.set(this.validationErrors, propertyName, errors);
                } else {
                    Vue.delete(this.validationErrors, propertyName);
                }
            },
            validateAll() {
                this.validate("name", this.name);
                this.validate("category", this.category);
                this.validate("price", this.price);
                return this.errors;
            }
        }
    }
</script>
```

```

        },
        handleSubmit() {
            this.hasSubmitted = true;
            if (this.validateAll()) {
                console.log(`WYSŁANO FORMULARZ: ${this.name} ${this.category} + ~
                    ${this.price}`);
            }
        }
    }
</script>
...

```

Do elementu `script` dodałem sekcję `watch`, która jest używana do definiowania obserwatorów. Obserwatorzy są opisani w rozdziale 17. — dzięki nim komponent może otrzymywać powiadomienia o zmianie jednej z wartości danych. W naszym przykładzie dodałem funkcje do sekcji `watch`, dzięki czemu otrzymam powiadomienia w przypadku zmian właściwości `name`, `category` i `price`. W takiej sytuacji wywołam metodę `validateWatch`, która zweryfikuje wartość właściwości, ale dopiero wtedy, gdy kliknięcie przycisku *Wyślij* nastąpi minimum raz. Ten warunek kontrolujemy za pomocą nowej właściwości danych o nazwie `hasSubmitted`. Dzięki nowemu podejściu komunikaty o błędach są wyświetlane zaraz po wprowadzeniu zmiany, o ile wcześniej nastąpiło kliknięcie przycisku *Wyślij* (rysunek 15.17).



Rysunek 15.17. Błędy walidacji są aktualizowane na bieżąco

Podsumowanie

W tym rozdziale objaśniłem dwukierunkowe wiązania danych i opisałem, jak można je tworzyć za pomocą dyrektywy `v-model`. Pokazałem także, jak zwalidować dane wprowadzone przez użytkownika w formularzu HTML, korzystając z dyrektywy `v-model` i innych funkcji Vue.js. W kolejnym rozdziale opowiem, jak skorzystać z możliwości komponentów, aby nadać aplikacji odpowiednią strukturę.

ROZDZIAŁ 16.



Stosowanie komponentów

W tym rozdziale omawiam komponenty pod kątem ich grupowania i budowania struktury, która ułatwia tworzenie aplikacji. Pokazuję, jak dodawać komponenty do projektu, jak ma nastąpić komunikacja między komponentami, a także jak komponenty mogą współpracować ze sobą, aby wyświetlić określone treści. Tabela 16.1 umiejscawia komponenty w szerszym kontekście.

Tabela 16.1. Umiejscowienie komponentów w szerszym kontekście

Pytanie	Odpowiedź
Czym są komponenty?	Komponenty mogą być łączone ze sobą, aby stworzyć złożone mechanizmy z prostszych „klocków”.
Dlaczego są użyteczne?	Tworzenie złożonej aplikacji za pomocą jednego komponentu utrudnia identyfikację fragmentów kodu odpowiedzialnych za poszczególne funkcje. Podział aplikacji na wiele komponentów oznacza, że każdy z nich może być utworzony i przetestowany niezależnie.
Jak się z nich korzysta?	Komponenty deklarują się za pomocą właściwości component w elemencie script. Aby skorzystać z utworzonego komponentu, należy wprowadzić własny znacznik HTML.
Czy są jakieś pułapki lub ograniczenia?	Komponenty są domyślnie izolowane od siebie — skuteczna i elegancka komunikacja między nimi bywa trudna do opanowania.
Czy są jakieś rozwiązania alternatywne?	Nie musisz tworzyć wielu komponentów, aby zbudować aplikację. Pojedynczy komponent dla całej aplikacji może być akceptowalny w przypadku prostszych projektów.

Tabela 16.2 podsumowuje rozdział.

Przygotowania do tego rozdziału

Kontynuuję pracę z projektem *templatesanddata* z rozdziału 15. Aby przygotować się do tego rozdziału, uprościłem komponent korzenia aplikacji (listing 16.1).

Tabela 16.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Zgrupuj podobne funkcje.	Dodaj komponenty do projektu.	16.1 – 16.9
Komunikuj się z komponentem dziecka.	Skorzystaj z mechanizmu propów.	16.10 – 16.12
Komunikuj się z komponentem rodzica.	Skorzystaj z mechanizmu własnych zdarzeń.	16.13 – 16.14
Połącz treści komponentów rodzica i dziecka.	Skorzystaj z mechanizmu slotów.	16.15 – 16.20

Listing 16.1. Uproszczenie treści w pliku *src/App.vue*

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    Komponent główny
  </div>
</template>
<script>
export default {
  name: 'App'
}
</script>
```

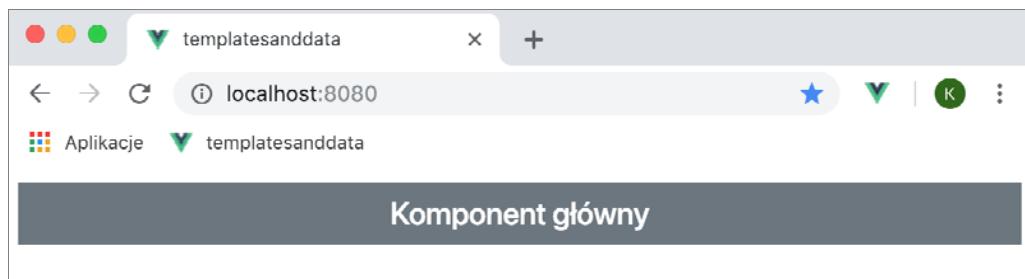
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Wykonaj polecenie z listingu 16.2 w katalogu *templatesanddata*, aby uruchomić narzędzia deweloperskie.

Listing 16.2. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Po wykonaniu inicjalizacji aplikacji otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć zastępczą treść (rysunek 16.1).

**Rysunek 16.1.** Przykładowa aplikacja po uruchomieniu

Omówienie komponentów jako podstawowych składników aplikacji

Tworzenie aplikacji w ramach jednego komponentu staje się coraz bardziej trudnym zadaniem wraz ze wzrostem złożoności aplikacji. Przykład tego widzieliśmy w rozdziale 15., gdzie formularz i logika walidacji były umieszczone obok siebie, przez co trudno było odróżnić elementy szablonu i skryptu odpowiedzialne za obsługę formularza i walidację. Komponent, który odpowiada za wiele aspektów, jest trudny do zrozumienia, testowania i zarządzania.

Komponenty stanowią esencję aplikacji Vue.js. Stosując wiele komponentów w aplikacji, możemy osiągnąć większą liczbę mniejszych, funkcjonalnych „klocków”, którymi jest łatwiej zarządzać i których łatwiej używać ponownie w aplikacji.

Z komponentową budową aplikacji nieuchronnie wiąże się powstanie relacji rodzic – dziecko, w której jeden komponent (**rodzic**) przekazuje pewną część swojej treści do innego komponentu (**dziecka**). Najlepiej pokazać to na przykładzie. Komponenty tworzy się w plikach o rozszerzeniu *.vue* w katalogu *src/components*. Najpierw dodaję plik *Child.vue* o treści z listingu 16.3.

Listing 16.3. Zawartość pliku *src/components/Child.vue*

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    Komponent-dziecko
  </div>
</template>
```

- **Uwaga** Projekt zawiera domyślnie plik *HelloWorld.vue* w katalogu *src/components*, jednak nie korzystam z niego w tym rozdziale — możesz go zignorować lub usunąć.

Komponent musi zadeklarować co najmniej element *template*, który rozpoczyna proces delegacji. Kolejnym krokiem jest tworzenie związku między rodzicem a dzieckiem (listing 16.4).

Listing 16.4. Delegacja do komponentu dziecka (*src/App.vue*)

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    Komponent-rodzic
    <ChildComponent></ChildComponent>
  </div>
</template>
<script>
import ChildComponent from "./components/Child";
export default {
  name: 'App',
  components: {
    ChildComponent
  }
}
</script>
```

Aby skonfigurować komponent-dziecko w ramach rodzica, potrzebne jest wykonanie trzech kroków. Najpierw korzystamy z instrukcji *import* dla komponentu-dziecka, np.:

```
...
import ChildComponent from "./components/Child";
...
```

Źródło zastosowane w instrukcji `import` musi rozpocząć się od kropki, co wskazuje na lokalny charakter instrukcji `import`, nie zaś na skorzystanie z pakietu instalowanego z zewnętrz. Najważniejszą częścią tej instrukcji jest nazwa, do której komponent-dziecko zostanie przypisany. Oznaczyłem ją pogrubieniem — w tym przypadku jest to `ChildComponent`.

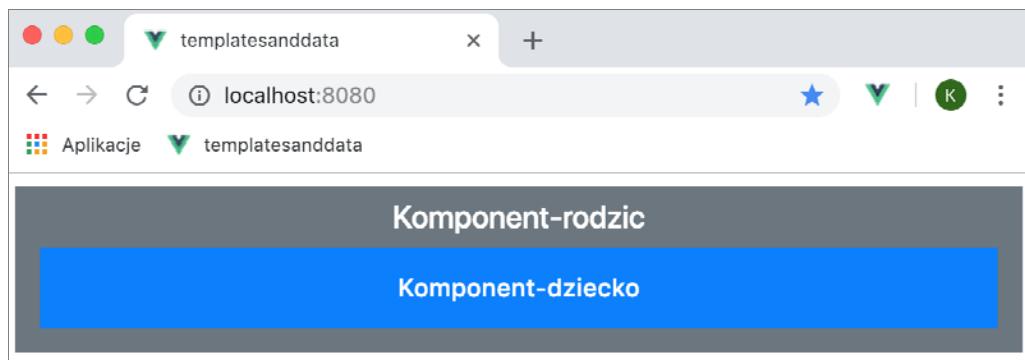
Nazwa instrukcji `import` jest używana do zarejestrowania komponentu za pomocą właściwości `components` w elemencie `script`:

```
...
export default {
  name: 'App',
  components: {
    ChildComponent
  }
}
...
```

Właściwość `components` otrzymuje obiekt, którego właściwości są używanymi przezeń komponentami. Przy deklaracji trzeba skorzystać z tych samych nazw co w instrukcji `import`. Ostatnim krokiem jest dodanie elementu HTML do szablonu komponentu-rodzica z wykorzystaniem znacznika, którego nazwa odpowiada nazwie z instrukcji `import` i `components`:

```
...
<div class="bg-secondary text-white text-center m-2 p-2 h5">
  Komponent-rodzic
  <ChildComponent></ChildComponent>
</div>
...
```

Gdy Vue.js przetwarza szablon komponentu-rodzica, własny znacznik HTML zostaje zastąpiony treścią elementu template komponentu-dziecka, co daje efekt jak na rysunku 16.2.



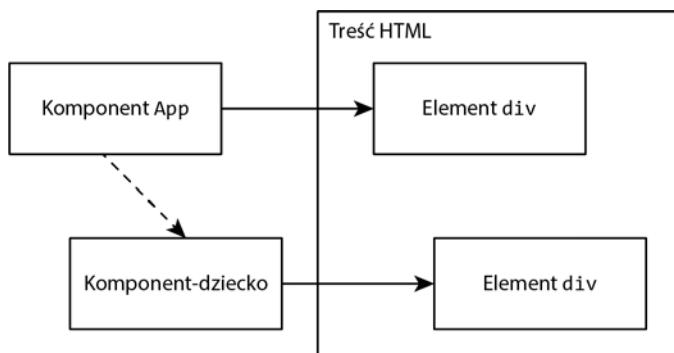
Rysunek 16.2. Dodawanie komponentu do przykładowej aplikacji

- **Uwaga** Ten przykład pokazuje ważny aspekt relacji rodzic – dziecko: to rodzic określa nazwę, pod którą występuje komponent-dziecko. Może to wydawać się dziwne, ale w ten sposób mamy możliwość stosowania nazw o specjalnym znaczeniu, pokazując dobrzej, jaki jest sens zastosowania komponentu-dziecka w danej sytuacji. Jak się niebawem przekonasz, pojedynczy komponent może być użyty na różne sposoby w różnych częściach aplikacji i dopuszczenie rodzica do ustalenia nazwy dziecka oznacza, że możemy nadać nazwę związaną z zastosowaniem komponentu-dziecka.

Jeśli skorzystasz z narzędzi F12 przeglądarki, aby sprawdzić dokument HTML, przekonasz się, że znacznik `ChildComponent` został zastąpiony przez kod HTML komponentu-dziecka.

```
...
<body>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    Komponent-rodzic
    <div class="bg-primary text-white text-center m-2 p-3 h6">
      Komponent-dziecko
    </div>
  </div>
  <script type="text/javascript" src="/app.js"></script>
</body>
...
```

Dzięki takiemu zabiegowi fragment treści przedstawianej przez komponent `App` (zadeklarowany w pliku `App.vue`) został przekazany do komponentu `Child` (plik `Child.vue`) — rysunek 16.3.



Rysunek 16.3. Relacja między komponentami rodzicem i dzieckiem

Omówienie nazw komponentów i elementów dzieci

W listingu 16.4 w instrukcji `import` podałem nazwę, aby skonfigurować komponent-dziecko, co doprowadziło do powstania następującego, nieco dziwnego kodu:

```
...
<div class="bg-secondary text-white text-center m-2 p-2 h5">
  Komponent-rodzic
  <ChildComponent></ChildComponent>
</div>
...
```

To podejście było użyteczne, ponieważ pokazałem, że to rodzic może nazywać własne dzieci. W Vue.js możemy zastosować bardziej wyszukane podejście, które koniec końców doprowadzi do bardziej eleganckiego, wynikowego kodu HTML.

Przed wszystkim Vue.js automatycznie ponownie formułuje nazwy znaczników własnych elementów HTML, szukając komponentu-dziecka do użycia (listing 16.5).

Listing 16.5. Ponowne formatowanie nazw znaczników (src/App.vue)

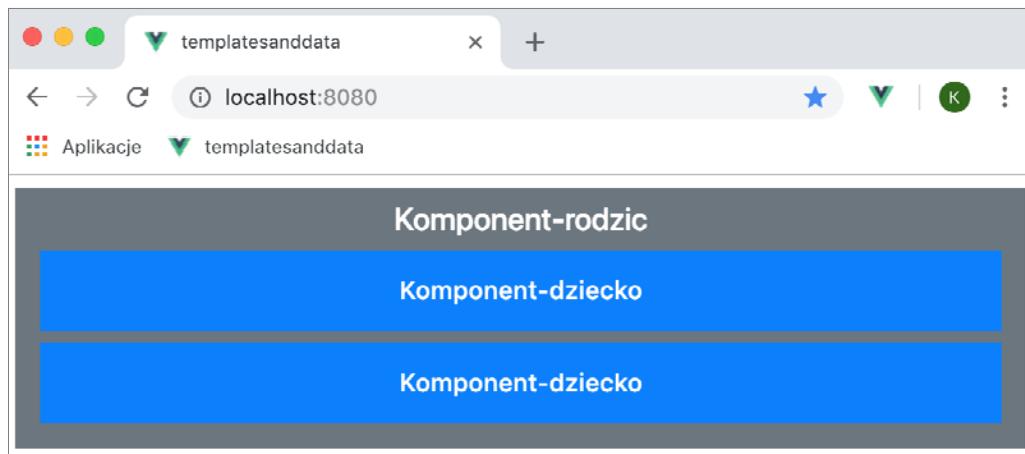
```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    Komponent-rodzic
```

```

<ChildComponent></ChildComponent>
<child-component></child-component>
</div>
</template>
<script>
import ChildComponent from "./components/Child";
export default {
  name: 'App',
  components: {
    ChildComponent
  }
}
</script>

```

Nowy własny element HTML w szablonie pokazuje, że Vue.js akceptuje również nazwy znaczników ze znakiem lącznika (-), które następnie są przekształcane zgodnie z konwencją camelCase, zwyczajowo używaną w nazewnictwie komponentów, np. child-component jest rozpoznawany jako komponent ChildComponent. Oba własne znaczniki HTML poinstruują Vue.js, aby przekazać fragment szablonu do komponentu-dziecka, co da efekt jak na rysunku 16.4.



Rysunek 16.4. Elastyczne formatowanie własnych znaczników

Właściwość components to słownik wykorzystywany przez Vue.js do przekształcania nazw własnych elementów HTML na nazwy komponentów-dzieci. Oznacza to, że możesz wymyślić kompletnie inne nazwy znaczników podczas rejestracji komponentu (listing 16.6).

Listing 16.6. Określanie nazwy znacznika w pliku src/App.vue

```

<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    Komponent-rodzic
    <MyFeature></MyFeature>
    <my-feature></my-feature>
  </div>
</template>
<script>
import ChildComponent from "./components/Child";
export default {
  name: 'App',
  components: {

```

```

    MyFeature: ChildComponent
}
}
</script>

```

Gdy określisz właściwość i wartość dla komponentu-dziecka, nazwa właściwości będzie mogła być używana jako własny element HTML. W tym przykładzie podałem nazwę MyFeature, a zatem mogę dodać komponent ChildComponent, korzystając ze znaczników MyFeature i my-feature.

Globalna rejestracja komponentów

Jeśli dysponujesz komponentem, który jest używany w całej aplikacji, możesz zarejestrować go globalnie. Zaletą takiego podejścia jest uniknięcie sytuacji, w której każdy komponent-rodzic musi być konfigurowany osobno. Z drugiej strony ten sam element HTML będzie wykorzystywany do stosowania komponentu-dziecka, przez co szablony staną się mniej czytelne. Aby zarejestrować komponent globalnie, dodaj instrukcję importu w pliku *main.js* i skorzystaj z metody *Vue.component*:

```

import Vue from 'vue'
import App from './App'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
import ChildComponent from "./components/Child";
Vue.config.productionTip = false
Vue.component("child-component", ChildComponent);
new Vue({
  render: h => h(App)
}).$mount('#app')

```

Metoda *Vue.component* musi być wywołana przed utworzeniem obiektu *Vue*. Metoda ta przyjmuje dwa argumenty — nazwę znacznika HTML, która zostanie zastosowana wobec komponentu, a także obiekt komponentu nazwany w instrukcji *import*. W rezultacie element *child-component* może być używany w całej aplikacji bez potrzeby dalszej konfiguracji.

Wykorzystywanie możliwości komponentów w komponentach-dzieciach

Komponent-dziecko, który zdefiniowałem w listingu 16.6, zawiera wyłącznie element *template*, jednak *Vue.js* obsługuje szereg przydatnych funkcji opisanych w poprzednich rozdziałach, również w komponentach-dzieciach, włączając w to wiązania jedno- i dwukierunkowe, obsługę zdarzeń, właściwości *data* i *computed*, a także metody. W listingu 16.7 dodaję element *script* do komponentu-dziecka i stosuję go do obsługi wiązań danych w szablonie.

Listing 16.7. Dodawanie nowych możliwości w pliku src/components/Child.vue

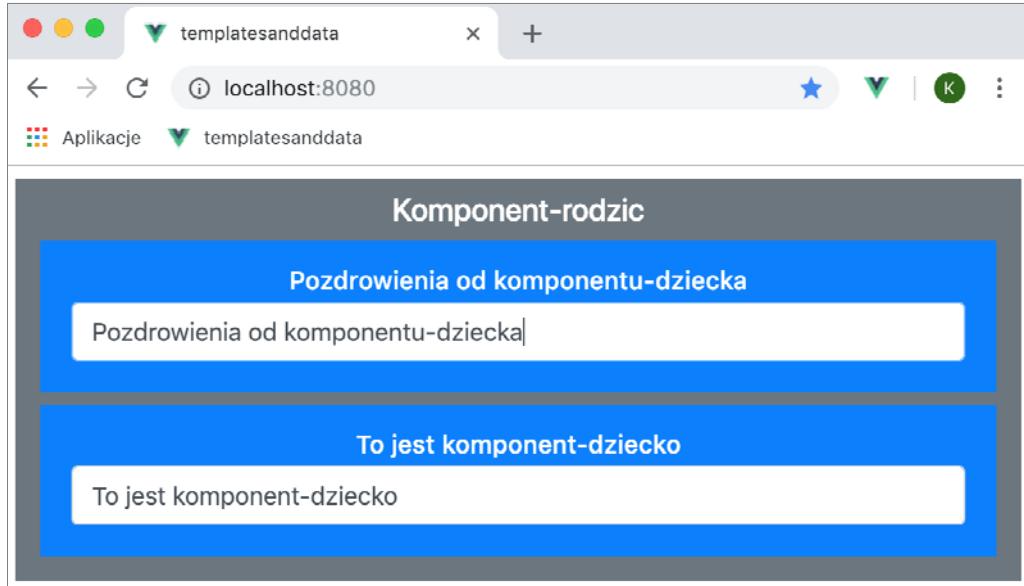
```

<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    {{ message }}
    <div class="form-group m-1">
      <input v-model="message" class="form-control" />
    </div>
  </div>
</template>
<script>

```

```
export default {
  data: function() {
    return {
      message: "To jest komponent-dziecko"
    }
  }
}</script>
```

Element script, który dodałem, definiuje właściwość danych o nazwie message, z której korzystam w elemencie template, w połączeniu z wiązaniem interpolacji tekstu i dyrektywą v-model. Efektem jest komponent-dziecko, który wyświetla element input o treści odzwierciedlającej wiązanie danych (rysunek 16.5).



Rysunek 16.5. Dodanie nowych możliwości do komponentu-dziecka

Omówienie izolacji komponentów

Komponenty są od siebie odizolowane, co oznacza, że nie musisz obawiać się popełnienia pomyłki polegającej na użyciu nazwy właściwości, metody lub wiązania do wartości, które występują już w innym komponencie.

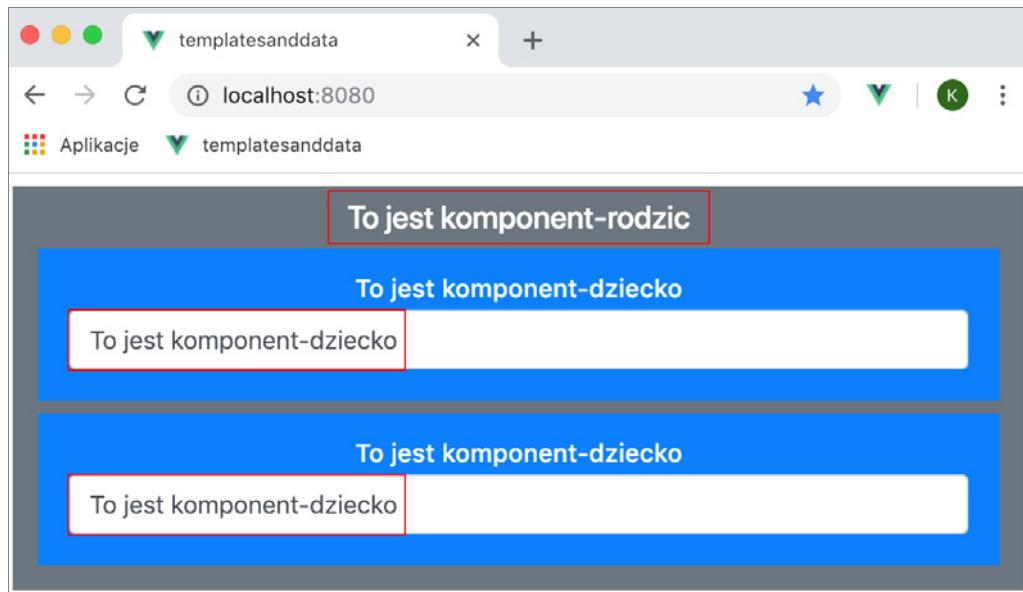
Na rysunku 16.5 widać, że zmiana zawartości jednego pola input nie ma wpływu na komponent-dziecko, mimo że oba komponenty mają właściwość message. Ta izolacja dotyczy także komponentów-rodziców i komponentów-dzieci. Możemy to udowodnić, dodając właściwość message do komponentu-rodzica w przykładowej aplikacji, co widać w listingu 16.8.

Listing 16.8. Dodawanie właściwości danych w pliku src/App.vue

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    {{ message }}
    <MyFeature></MyFeature>
    <my-feature></my-feature>
  </div>
</template>
```

```
<script>
import ChildComponent from "./components/Child";
export default {
  name: 'App',
  components: {
    MyFeature: ChildComponent
  },
  data: function() {
    return {
      message: "To jest komponent-rodzic"
    }
  }
}
</script>
```

W tym momencie mamy trzy właściwości danych o nazwie `message` w aplikacji, ale Vue.js izoluje je od siebie, dzięki czemu zmiana jednej z nich nie wpłynie na pozostałe (rysunek 16.6).



Rysunek 16.6. Izolacja komponentu-rodzica od komponentu-dziecka

Omówienie zasięgów CSS

Jeśli zdefiniujesz własne style w komponencie, zauważysz, że są one stosowane wobec elementów zdefiniowanych gdziekolwiek, np. poniższy kod:

```
...
<style>
  div { border: 5px solid red ; }
</style>
...
```

dopasuje dowolny element `div` w aplikacji i zastosuje wobec niego czerwone obramowanie. Jeśli chcesz ograniczyć własne style CSS do komponentu, który je definiuje, możesz dodać atrybut `scoped` do elementu `style`:

```
...
<style scoped>
  div { border: 5px solid red ; }
</style>
...
```

Atrybut `scoped` informuje Vue.js o konieczności ograniczenia stosowania stylów tylko do elementu template komponentu, a nie do elementów szablonów innych komponentów.

Stosowanie propów w komponentach

Izolowanie komponentów jest dobre w wielu przypadkach i na pewno jest to dobra koncepcja domyślna — unikamy w ten sposób nieoczekiwanych interakcji. Gdyby komponenty nie były od siebie izolowane, zmiana jednej właściwości `message` mogłaby wpływać na działanie wszystkich komponentów. Poza tym w większości aplikacji komponenty muszą współpracować ze sobą, aby zapewnić końcowy efekt satysfakcyjny dla użytkownika. Nie da się tego osiągnąć bez przełamania bariery między komponentami. Jedną z funkcji, która pozwala na współpracę między komponentami, jest `prop`. Prop pozwala rodzicowi na przekazanie wartości do dziecka. W listingu 16.9 dodaję prop do komponentu-dziecka.

Listing 16.9. Dodawanie propa do pliku src/Child.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    {{ message }}
    <div class="form-group m-1 text-left">
      <label>{{ labelText }}</label>
      <input v-model="message" class="form-control" />
    </div>
  </div>
</template>
<script>
  export default {
    props: ["labelText"],
    data: function () {
      return {
        message: "To jest komponent-dziecko"
      }
    }
  }
</script>
```

Prop definiuje się za pomocą tablicy tekstów, przypisanej do właściwości `props` w elemencie `script` komponentu. W naszym przypadku nazwa propa to `labelText`. Po zdefiniowaniu propa możesz korzystać z niego w komponencie, np. w wiązaniu interpolacji tekstu. Jeśli chcesz zmodyfikować wartość po otrzymaniu jej z komponentu-rodzica, musisz skorzystać z właściwości `data` lub `computed`, z której jest pobrana początkowa wartość propa (listing 16.10).

Listing 16.10. Ustawianie mutowalnej wartości z propa w pliku src/components/Child.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    {{ message }}
    <div class="form-group m-1 text-left">
      <label>{{ labelText }}</label>
      <input v-model="message" class="form-control" />
```

```

        </div>
    </div>
</template>
<script>
    export default {
        props: ["labelText", "initialValue"],
        data: function () {
            return {
                message: this.initialValue
            }
        }
    }
</script>

```

- **Uwaga** To podejście jest konieczne, ponieważ przepływ danych propa odbywa się w jednym kierunku: od rodzica do dziecka. Jeśli zmodyfikujesz wartość propa, zostanie ona najprawdopodobniej przesłonięta przez komponent-rodzica.

Komponent definiuje drugi prop, initialValue, który jest używany do ustawienia wartości właściwości message.

Stosowanie propa w komponencie-rodzicu

Gdy komponent definiuje prop, jego rodzic może wysłać wartości danych za pomocą atrybutów własnego elementu HTML (listing 16.11).

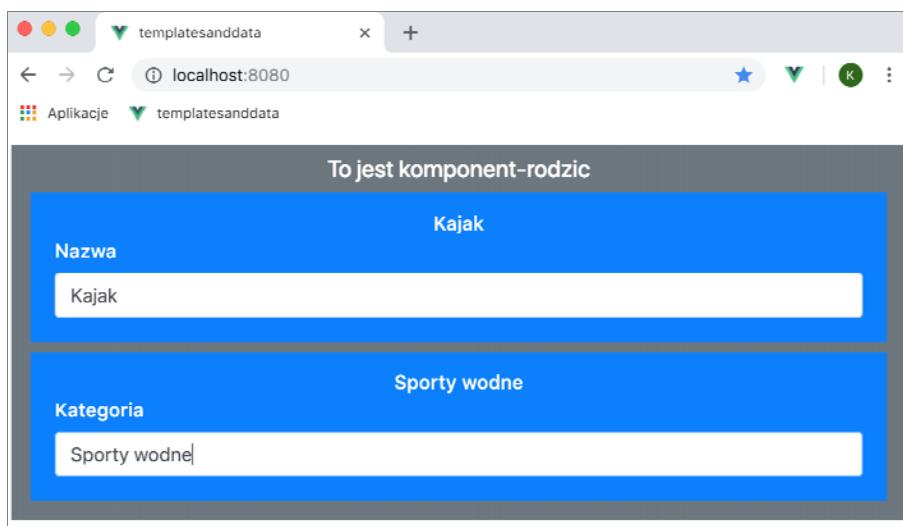
Listing 16.11. Zastosowanie propa w pliku src/App.vue

```

<template>
    <div class="bg-secondary text-white text-center m-2 p-2 h5">
        {{ message }}
        <MyFeature labelText="Nazwa" initialValue="Kajak"></MyFeature>
        <my-feature label-text="Kategoria" initial-value="Sporty wodne"></my-feature>
    </div>
</template>
<script>
    import ChildComponent from "./components/Child";
    export default {
        name: 'App',
        components: {
            MyFeature: ChildComponent
        },
        data: function () {
            return {
                message: "To jest komponent-rodzic"
            }
        }
    }
</script>

```

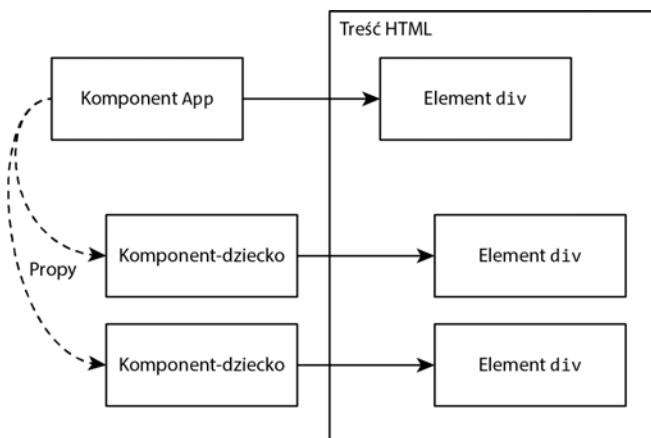
Vue.js jest równie elastyczne pod względem dopasowania nazw atrybutów do propów co w przypadku nazw komponentów i własnych znaczników HTML. Oznacza to, że mogę skorzystać z nazw labelText lub label-text, aby ustawić wartość propa. Atrybuty z listingu 16.11 konfigurują komponenty-dzieci, co pozwala uzyskać efekt jak na rysunku 16.7.



Rysunek 16.7. Zastosowanie propa do konfiguracji komponentu-dziecka

-
- **Wskazówka** Trzymanie wyżej opisanego efektu może wymagać odświeżenia przeglądarki.
 - **Wskazówka** Wartości atrybutów propów są traktowane dosłownie, czyli nie są ewaluowane jako wyrażenia. Jeśli chcesz przekazać do komponentu-dzieckałańcuch znaków, wystarczy napisać: `my-attr="Cześć"`. Nie musisz korzystać z dodatkowych cudzysłowów: `my-attr="'Cześć'"`. Jeśli chcesz dokonać ewaluacji wyrażenia będącego atrybutem propa, musisz skorzystać z dyrektywy `v-bind`. Jeśli chcesz, aby komponent-dziecko reagował na zmiany propa w wiązaniach danych, możesz skorzystać z obserwatora, opisanego w rozdziale 17.
-

Korzystając z propa, trzeba pamiętać, że przepływ danych odbywa się od rodzica do dziecka (rysunek 16.8). Otrzymasz ostrzeżenie, jeśli spróbujesz zmodyfikować wartość propa — powinieneś go wykorzystywać do inicjalizacji właściwości danych, jak w listingu 16.10.



Rysunek 16.8. Przepływ danych przy zastosowaniu propa

Ustawianie zwykłych atrybutów we własnych elementach HTML

Gdy Vue.js zastępuje własny element HTML za pomocą szablonu komponentu-dziecka, dochodzi do przeniesienia wszystkich atrybutów niebędących propami do nadrzędnego elementu szablonu. Może to doprowadzić do mylących wyników, zwłaszcza gdy element w szablonie dziecka już ma taki atrybut. Na przykład jeżeli dysponujemy następującym elementem szablonu dziecka:

```
...
<template>
  <div id="childIdValue">To jest element dziecka</div>
</template>
...

```

i następującym elementem szablonu rodzica:

```
...
<template>
  <my-feature id="parentIdValue"></my-feature>
</template>
...

```

to atrybut zastosowany w rodzicu przesłoni atrybut dziecka, generując następujący kod HTML:

```
...
<div id="parentIdValue">To jest element dziecka</div>
...
```

Zachowanie jest nieco inne dla atrybutów `class` i `style`, które są obsługiwane przez przeglądarki na zasadzie złączenia dwóch wartości atrybutów. Jeśli element dziecka jest następujący:

```
...
<template>
  <div class="bg-primary">To jest element dziecka</div>
</template>
...

```

a tak wygląda element szablonu rodzica:

```
...
<template>
  <my-feature class="text-white"></my-feature>
</template>
...

```

to przeglądarka połączy wartości atrybutu klas i wygeneruje następujący kod HTML:

```
...
<div class="bg-primary text-white">To jest element dziecka</div>
...
```

Należy szczególnie uważać, gdy rodzic i dziecko ustawiają ten sam atrybut — najlepiej w ogóle takich sytuacji unikać. Jeśli chcesz przenieść odpowiedzialność za określenie treści HTML wyświetlanej przez dziecko na rodzica, skorzystaj ze slotów, które opisuję w punkcie „[Stosowanie slotów w komponentach](#)”.

Wyrażanie wartości w propach

Wartości atrybutów w propach nie są ewaluowane jako wyrażenia, o ile nie skorzystasz z dyrektywy v-bind (listing 16.12).

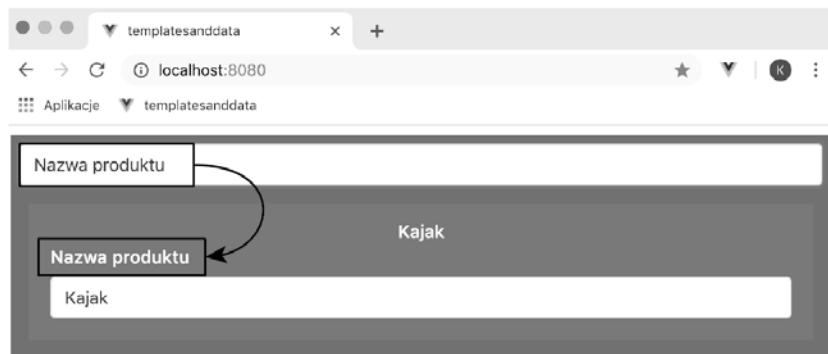
Listing 16.12. Zastosowanie wyrażenia w pliku src/App.vue

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    <div class="form-group">
      <input class="form-control" v-model="labelText" />
    </div>
    <my-feature v-bind:label-text="labelText" initial-value="Kajak"></my-feature>
  </div>
</template>
<script>
  import ChildComponent from "./components/Child";
  export default {
    name: 'App',
    components: {
      MyFeature: ChildComponent
    },
    data: function () {
      return {
        message: "To jest komponent-rodzic",
        labelText: "Nazwa"
      }
    }
  }
</script>
```

Szablon komponentu-rodzica zawiera element `input`, który wykorzystuje dyrektywę `v-model` do powiązania go z właściwością `labelText`. Ta sama właściwość jest określana we własnym elemencie dla dyrektywy dziecka, dzięki czemu tworzymy jednokierunkowe wiązanie między propem `labelText` komponentu-dziecka a właściwością `data` komponentu-rodzica.

```
...
<my-feature v-bind:label-text="labelText" initial-value="Kajak"></my-feature>
...
```

W rezultacie, gdy dochodzi do edycji elementu `input` komponentu rodzica, nową wartość otrzyma komponent-dziecko, w ramach którego jest ona wyświetlana w wiązaniu interpolacji tekstu (rysunek 16.9).



Rysunek 16.9. Wyrażenie wartości w propie

- **Ostrzeżenie** Przepływ zmian wciąż pozostaje jednokierunkowy — nawet jeśli korzystamy z dyrektywy `v-bind`. Zmiany poczynione wobec wartości propa przez komponent-dziecko nie docierają do rodzica i z pewnością zostaną anulowane, gdy właściwość podana w dyrektywie `v-bind` ulegnie zmianie. W kolejnym punkcie dowiesz się więcej na temat transferu danych od dziecka do rodzica.

Tworzenie własnych zdarzeń

Uzupełnieniem mechanizmu propów są własne zdarzenia (ang. *custom events*), które pozwalają na transfer danych od dziecka do rodzica. Aby przedstawić przykład użycia własnych zdarzeń, do komponentu dziecka dodaję nową funkcję działającą tylko w obrębie dziecka (listing 16.13).

Listing 16.13. Dodawanie funkcji do pliku src/components/Child.vue

```
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    <div class="form-group m-1 text-left">
      <label>Nazwa</label>
      <input v-model="product.name" class="form-control" />
    </div>
    <div class="form-group m-1 text-left">
      <label>Kategoria</label>
      <input v-model="product.category" class="form-control" />
    </div>
    <div class="form-group m-1 text-left">
      <label>Cena</label>
      <input v-model.number="product.price" class="form-control" />
    </div>
    <div class="mt-2">
      <button class="btn btn-info" v-on:click="doSubmit">Wyślij</button>
    </div>
  </div>
</template>
<script>
  export default {
    props: ["initialProduct"],
    data: function () {
      return {
        product: this.initialProduct || {}
      }
    },
    methods: {
      doSubmit() {
        this.$emit("productSubmit", this.product);
      }
    }
  }
</script>
```

Komponent pozwala na podstawową edycję produktów, gdzie elementy `input` odpowiadają za właściwości `name`, `category` i `price` obiektu właściwości danych `product`. Początkowe wartości danych są przekazywane przez prop `initialProduct`.

Łączenie propów i własnych zdarzeń

W pewnym momencie swojej nauki większość początkujących programistów Vue.js spróbuje utworzyć związane ze sobą komponenty rodzica i dziecka, które będą wyświetlać i zmieniać tę samą wartość danych, łącząc ze sobą własne zdarzenia, propy i dyrektywę v-model. Choć da się osiągnąć taki skutek, tego rodzaju działanie jest wbrew przeznaczeniu tych mechanizmów, a czasami potrafi ono dać efekt niezgodny z przewidywaniemi. Jeśli chcesz wyświetlać i zmieniać te same dane w wielu komponentach, korzystaj ze współdzielonego stanu aplikacji, opisanego w rozdziale 20. W prostszych aplikacjach wystarczy zastosować szynę zdarzeń (ang. *event bus*), opisaną w rozdziale 18.

Oprócz tego mamy także przycisk (button), który korzysta z dyrektywy v-on, aby zareagować na zdarzenie click poprzez wywołanie metody doSubmit. Metoda ta pozwala na komunikację komponentu ze swoim rodzicem, tak jak poniżej:

```
...
doSubmit() {
  this.$emit("productSubmit", this.product);
}
...
```

Metoda \$emit, która jest wywoływana za pomocą słowa kluczowego this, jest używana do wysyłania własnego zdarzenia. Pierwszy argument to rodzaj zdarzenia, wyrażony jako tekst, a drugi, opcjonalny argument stanowi treść, „ładunek” (ang. *payload*) zdarzenia, która to treść może być dowolna — najważniejsze, aby była przydatna dla rodzica. W tym przypadku wysyłam zdarzenie productSubmit, przekazując obiekt product w swojej treści.

Odbieranie własnego zdarzenia od komponentu-dziecka

Dyrektyna v-on jest używana przez komponent-rodzica, aby otrzymywać zdarzenia od komponentów-dzieci, podobnie jak w przypadku zwykłych zdarzeń DOM. W listingu 16.14 zmodyfikowałem plik App.vue, przez co komponent-dziecko dostarcza początkowe dane i reaguje na zdarzenie po jego wyzwoleniu.

Listing 16.14. Reagowanie na zdarzenia komponentu-dziecka w pliku src/App.vue

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    <h6>{{ message }}</h6>
    <my-feature v-bind:initial-product="product"
                v-on:productSubmit="updateProduct">
      </my-feature>
    </div>
</template>
<script>
  import ChildComponent from "./components/Child";
  export default {
    name: 'App',
    components: {
      MyFeature: ChildComponent
    },
    data: function () {
      return {
        message: "Gotowy",
        product: {
          name: "Kajak",
          category: "Sporty wodne",
        }
      }
    }
  }
</script>
```

```

        price: 275
    }
}
},
methods: {
    updateProduct(newProduct) {
        this.message = JSON.stringify(newProduct);
    }
}
}
</script>

```

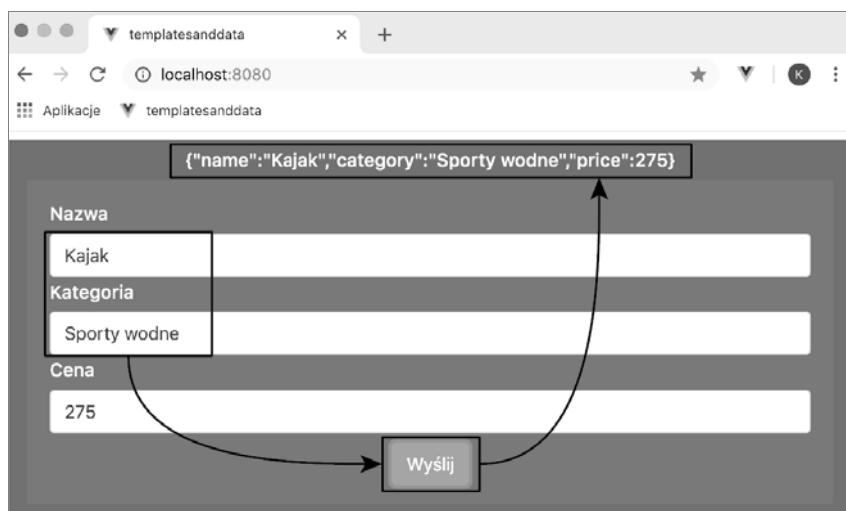
Dyrektywa `v-on` jest używana do nasłuchiwanego zdarzenia komponentu-dziecka z wykorzystaniem nazwy, która została przekazana jako pierwszy argument metody `$emit` (w tym przykładzie jest to `productSubmit`):

```

...
<my-feature v-bind:initial-product="product" v-on:productSubmit="updateProduct">
...

```

W tym przypadku wiązanie `v-on` jest używane do podjęcia działania w związku ze zdarzeniem `productSubmit`. Ma to związek z wywołaniem metody `updateProduct`. Metoda używana przez komponent-rodzica otrzymuje opcjonalną zawartość, którą komponent-dziecko wykorzystuje jako drugi argument metody `$emit`. W tym przykładzie reprezentacja zawartości w formacie JSON jest przypisywana do właściwości danych o nazwie `message`, wyświetlanej za pomocą wiązania interpolacji tekstu. Dzięki temu możesz modyfikować wartości wyświetlane przez komponent-dziecko, kliknąć przycisk *Wyślij* i wyświetlić dane otrzymane przez komponent-rodzica (rysunek 16.10).



Rysunek 16.10. Obsługa własnego zdarzenia z perspektywy komponentu-dziecka

- **Uwaga** Własne zdarzenia nie działają jak zwykłe zdarzenia DOM, mimo że w obu przypadkach stosuje się dyrektywę `v-on`. Własne zdarzenia są przekazywane tylko do komponentu-rodzica i nie są propagowane przez całą hierarchię elementów HTML w drzewie DOM. Nie mają też faz przechwytywania, celu i pęcherzykowej. Jeśli chcesz wyjść poza komunikację rodzinę – dziecko, możesz skorzystać z szyny zdarzeń, opisanej w rozdziale 18., lub współdzielonego stanu (rozdział 20.).

Stosowanie slotów komponentów

Jeśli korzystasz z danego komponentu w różnych częściach swojej aplikacji, możesz zechcieć dostosować wygląd znaczników HTML, aby dopasować się do kontekstu.

W przypadku prostych zmian treści można skorzystać z propa. Komponent-rodzic może bezpośrednio stylować własny element HTML zastosowany wobec komponentu-dziecka.

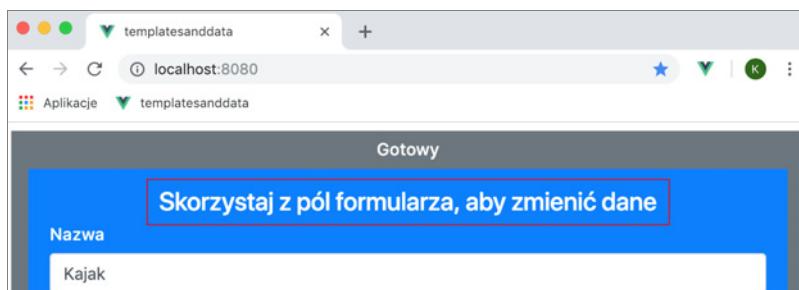
W przypadku bardziej zaawansowanych zmian Vue.js dostarcza mechanizm slotów umożliwiający przekazanie treści przez komponent-rodzica. Za pomocą tego mechanizmu możliwe jest wyświetlenie mechanizmów komponentu-dziecka. W listingu 16.15 do komponentu-dziecka dodajemy slot, który będzie użyty do wyświetlania treści dostarczonej przez rodzica.

Listing 16.15. Dodawanie slotu do pliku src/components/Child.vue

```
...
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    <slot>
      <h4>Skorzystaj z pól formularza, aby zmienić dane</h4>
    </slot>
    <div class="form-group m-1 text-left">
      <label>Nazwa</label>
      <input v-model="product.name" class="form-control" />
    </div>
    <div class="form-group m-1 text-left">
      <label>Kategoria</label>
      <input v-model="product.category" class="form-control" />
    </div>
    <div class="form-group m-1 text-left">
      <label>Cena</label>
      <input v-model.number="product.price" class="form-control" />
    </div>
    <div class="mt-2">
      <button class="btn btn-info" v-on:click="doSubmit">Wyślij</button>
    </div>
  </div>
</template>
...

```

Element slot oznacza obszar szablonu komponentu, który zostanie zastąpiony przez treść dostarczoną przez komponent-rodzica, umieszczoną pomiędzy znacznikami początkowym i końcowym własnego elementu, zastosowanymi wobec komponentu-dziecka. Jeśli komponent-rodzic nie dostarcza żadnej treści, Vue.js zignoruje element slotu, co doprowadzi do sytuacji, z której możesz się zapoznać po zapisaniu pliku Child.vue i analizie zawartości przeglądarki (rysunek 16.11).



Rysunek 16.11. Wyświetlanie domyślnej zawartości w slotie

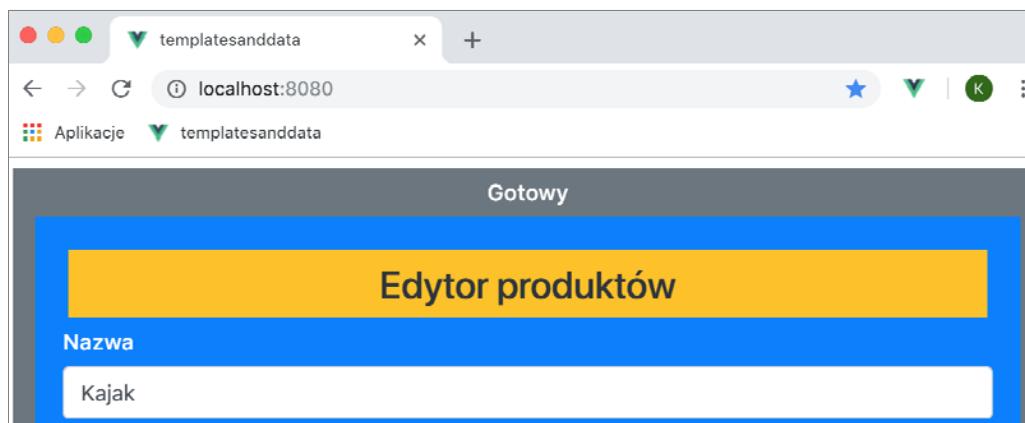
■ **Wskazówka** Do zapoznania się z tym przykładem może być konieczne odświeżenie przeglądarki.

W ten sposób umożliwiamy komponentowi-dziecku wyświetlenie użytecznej treści. Aby przesłonić domyślną zawartość, komponent-dziecko musi zatrzymać określone elementy w swoim szablonie (listing 16.16).

Listing 16.16. Dostarczanie elementów dla slotu w pliku *src/App.vue*

```
...
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    <h6>{{ message }}</h6>
    <my-feature v-bind:initial-product="product"
                v-on:productSubmit="updateProduct">
      <div class="bg-warning m-2 p-2 h3 text-dark">Edytor produktów</div>
    </my-feature>
  </div>
</template>
...
```

Element `div`, który pojawia się między początkowym a końcowym znacznikiem elementu `my-feature`, jest używany jako treść elementu slot szablonu komponentu-dziecka, co daje efekt jak na rysunku 16.12.



Rysunek 16.12. Dostarczenie treści dla slotu komponentu-dziecka

Stosowanie slotów nazwanych

Jeśli komponent-dziecko może otrzymywać różne obszary treści od swoich rodziców, możliwe jest przypisywanie nazw do slotów, jak w listingu 16.17.

Listing 16.17. Dodawanie slotów nazwanych w pliku *src/components/Child.vue*

```
...
<template>
  <div class="bg-primary text-white text-center m-2 p-3 h6">
    <slot name="header">
      <h4>Skorzystaj z pól formularza, aby zmodyfikować dane</h4>
    </slot>
    <div class="form-group m-1 text-left">
      <label>Nazwa</label>
      <input v-model="product.name" class="form-control" />
    </div>
  </div>
```

```

</div>
<div class="form-group m-1 text-left">
    <label>Kategoria</label>
    <input v-model="product.category" class="form-control" />
</div>
<div class="form-group m-1 text-left">
    <label>Cena</label>
    <input v-model.number="product.price" class="form-control" />
</div>
<slot name="footer"></slot>
<div class="mt-2">
    <button class="btn btn-info" v-on:click="doSubmit">Wyślij</button>
</div>
</div>
</template>
...

```

Atrybut name jest używany do przypisania nazwy do każdego elementu slot. W tym przykładzie do nazwy header przypisałem element, który pojawia się nad elementem input, a do nazwy footer — slot, który pojawia się pod nim. Slot footer nie zawiera żadnych elementów, co oznacza, że nie zostanie wyświetlone nic, o ile komponent-rodzic nie dostarczy treści.

Aby zastosować nazwany slot, rodzic dodaje atrybut slotu do jednego z elementów zawartego pomiędzy własnymi znacznikami początkowym i końcowym (listing 16.18).

Listing 16.18. Zastosowanie slotów nazwanych w pliku src/App.vue

```

...
<template>
    <div class="bg-secondary text-white text-center m-2 p-2 h5">
        <h6>{{ message }}</h6>
        <my-feature v-bind:initial-product="product"
                    v-on:productSubmit="updateProduct">
            <div slot="header" class="bg-warning m-2 p-2 h3 text-dark">
                Edytor produktów
            </div>
            <div slot="footer" class="bg-warning p-2 h3 text-dark">
                Sprawdź informacje przed wysłaniem
            </div>
        </my-feature>
    </div>
</template>
...

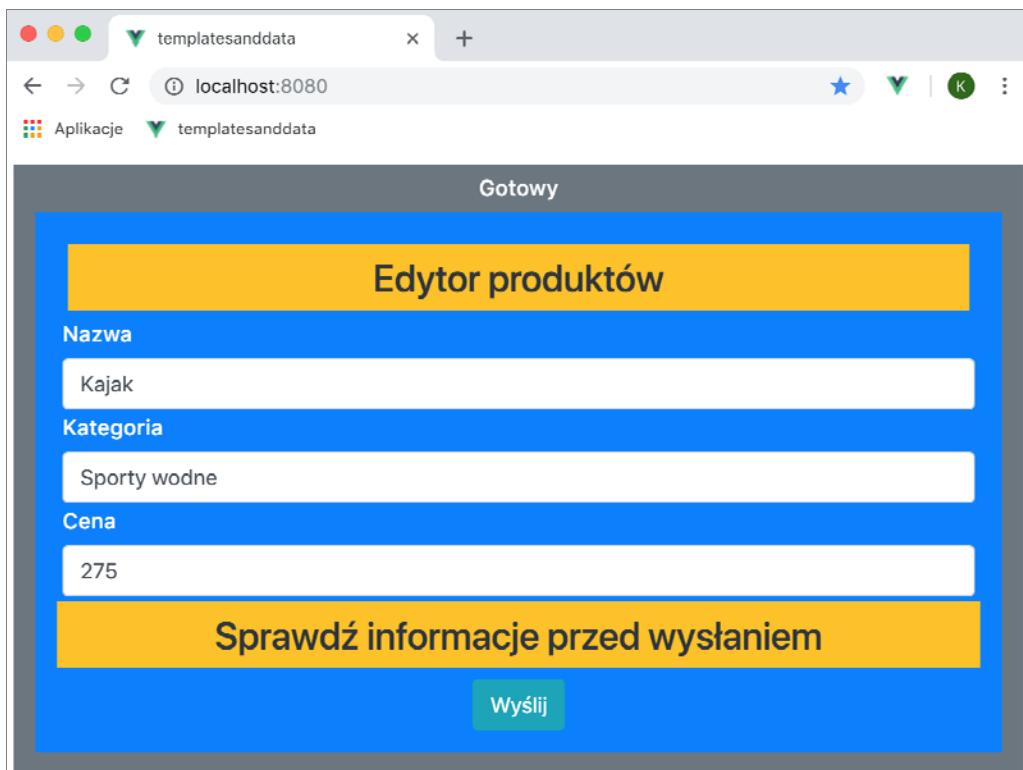
```

Rodzic dostarcza elementy div dla wszystkich nazwanych slotów, co daje efekt jak na rysunku 16.13.

-
- **Wskazówka** Jeśli komponent-dziecko definiuje nienazwany slot, zostaną wyświetlone wszystkie elementy z szablonu rodzica, które nie są przypisane do slotu z atrybutem slot. Jeśli komponent-dziecko nie definiuje nienazwanego slotu, elementy nieprzypisane zostaną anulowane.
-

Stosowanie slotów o ograniczonym zasięgu

Slot o ograniczonym zasięgu pozwala komponentowi-rodzicowi na dostarczenie szablonu, w ramach którego komponent-dziecko może wstawić dane. Jest to przydatne, gdy komponent-dziecko przekształca dane otrzymane od rodzica, a jednocześnie to rodzic kontroluje sposób formatowania danych. Aby przedstawić mechanizm slotów o ograniczonym zasięgu, dodaję w katalogu src/components plik *ProductDisplay.vue* o treści jak w listingu 16.19.



Rysunek 16.13. Zastosowanie slotów nazwanych

Listing 16.19. Zawartość pliku `src/components/ProductDisplay.vue`

```
<template>
  <ul>
    <li v-for="prop in Object.keys(product)" v-bind:key="prop">
      <slot v-bind:propname="prop" v-bind:propvalue="product[prop]">
        {{ prop }}: {{ product[prop] }}
      </slot>
    </li>
  </ul>
</template>
<script>
export default {
  props: ["product"]
}
</script>
```

Komponent otrzymuje obiekt za pomocą propa o nazwie `product`, którego właściwości i wartości są wyliczone w szablonie przy użyciu dyrektywy `v-for`. Element `slot` dostarcza rodzica z możliwością zastąpienia treści, która wyświetla nazwę i wartość, ale z ważnym dodatkiem opisany poniżej:

```
...
<slot v-bind:propname="prop" v-bind:propvalue="product[prop]">
...
...
```

Atrybuty propname i propvalue pozwalają komponentowi-rodzicowi na dołączenie przypisanych do nich wartości do zawartości slotu (listing 16.20).

Listing 16.20. Zastosowanie slotu o ograniczonym zasięgu w pliku src/App.vue

```
<template>
  <div class="bg-secondary text-white text-center m-2 p-2 h5">
    <product-display v-bind:product="product">
      <div slot-scope="data" class="bg-info text-left">
        {{data.propname}} ma wartość {{ data.propvalue }}
      </div>
    </product-display>
    <my-feature v-bind:initial-product="product"
      v-on:productSubmit="updateProduct">
      <div slot="header" class="bg-warning m-2 p-2 h3 text-dark">
        Edytor produktów
      </div>
      <div slot="footer" class="bg-warning p-2 h3 text-dark">
        Sprawdź szczegóły przed wysłaniem
      </div>
    </my-feature>
  </div>
</template>
<script>
import ChildComponent from "./components/Child";
import ProductDisplay from "./components/ProductDisplay";
export default {
  name: 'App',
  components: {
    MyFeature: ChildComponent,
    ProductDisplay
  },
  data: function () {
    return {
      message: "Gotowy",
      product: {
        name: "Kajak",
        category: "Sporty wodne",
        price: 275
      }
    }
  },
  methods: {
    updateProduct(newProduct) {
      this.message = JSON.stringify(newProduct);
    }
  }
}
</script>
```

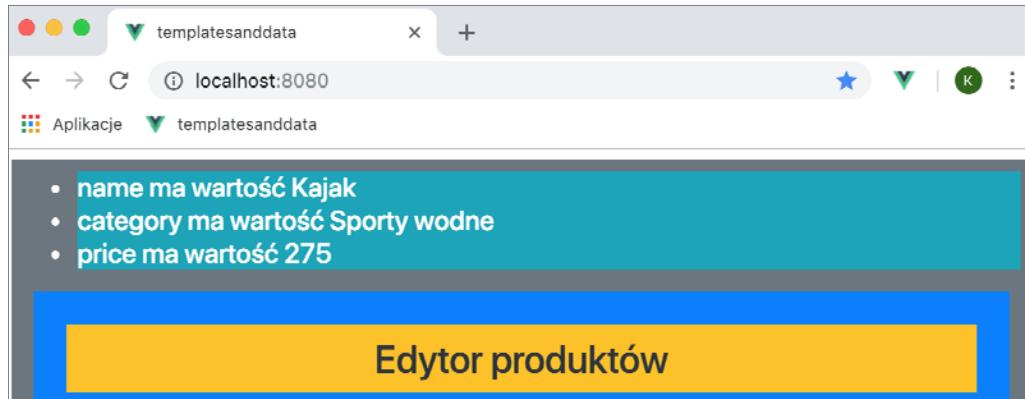
Atrybut slot-scope jest używany do wyboru nazwy dla tymczasowej zmiennej tworzonej w momencie przetworzenia szablonów. Otrzyma on właściwość dla każdego atrybutu, który został zdefiniowany przez komponent-dziecko w elemencie slot. Następnie możemy korzystać z takich elementów w wiązaniach danych i dyrektywach w treści slotu, np.:

```
...
<product-display v-bind:product="product">
  <div slot-scope="data" class="bg-info text-left">
    {{data.propname}} is {{ data.propvalue }}
```

```
</div>
</product-display>
...

```

W rezultacie otrzymujemy element języka HTML z komponentu-rodzica, który to element jest pomieszany z danymi dostarczonymi przez komponent-dziecko (rysunek 16.14).



Rysunek 16.14. Zastosowanie slotu o ograniczonym zasięgu

-
- **Wskazówka** Wyrażenia i wiązania danych w slocie o ograniczonym zasięgu są ewaluowane w kontekście rodzica, co oznacza, że Vue.js przeszuka wszelkie nieprefiksowane wartości za pomocą zmiennej nazwanej przy użyciu atrybutu s w elemencie script komponentu-rodzica.
-

Podsumowanie

W tym rozdziale opisałem zasady stosowania komponentów jako podstawowych elementów struktury w aplikacji Vue.js. Pokazałem, jak grupować powiązaną treść i kod, aby ułatwić tworzenie aplikacji i jej utrzymanie. Objąłem, jak domyślnie izolować komponenty, a także jak korzystać z propów i własnych zdarzeń, aby komponenty mogły się ze sobą komunikować. Wspomniałem także o tym, jak korzystać ze slotów, aby komponent-rodzic dostarczył treść do komponentu-dziecka. W części III tej książki opisuję zaawansowane funkcje Vue.js.

CZĘŚĆ III



Zaawansowane funkcje Vue.js



ROZDZIAŁ 17.

Omówienie cyklu życia komponentu Vue.js

Gdy Vue.js tworzy komponent, możemy mówić o początku dobrze określonego cyklu życia, w skład którego wchodzą: przygotowanie wartości danych, renderowanie treści HTML w ramach obiektowego modelu dokumentu (**DOM** — ang. *Document Object Model*), a także obsługa wszelkich aktualizacji treści. W tym rozdziale opiszę różne etapy cyklu życia komponentu i przedstawię różne metody reagowania na nie.

Cykl życia komponentu jest wart Twojej uwagi z dwóch przyczyn. Po pierwsze, im więcej wiesz na temat Vue.js, tym lepiej jesteś w stanie radzić sobie z potencjalnymi problemami. Po drugie, pewne bardziej zaawansowane zagadnienia, które omawiam w kolejnych rozdziałach, łatwiej zrozumiesz, gdy będziesz znać kontekst, w którym one funkcjonują. Tabela 17.1 umiejscowia cykl życia komponentu w szerszym kontekście.

Tabela 17.1. Umiejscowienie cyklu życia komponentu w szerszym kontekście

Pytanie	Odpowiedź
Czym jest cykl życia komponentu?	Każdy komponent ma dobrze zdefiniowany cykl życia, który zaczyna się w momencie utworzenia go przez Vue.js, a kończy się w chwili zniszczenia komponentu.
Dlaczego jest użyteczny?	Dobrze zdefiniowany cykl życia pozwala funkcjonować komponentom w przewidywalny sposób, a dzięki metodom powiadomień komponent może reagować na różne etapy cyklu życia.
Jak się z niego korzysta?	Vue.js realizuje cykl życia automatycznie — nie jest potrzebne żadne bezpośrednie działanie, aby go aktywować. Jeśli komponent ma otrzymywać powiadomienia o cyklu życia, należy zaimplementować metody opisane w tabeli 17.3.
Czy są jakieś pułapki lub ograniczenia?	Jedną z głównych przyczyn implementowania metod powiadomień cyklu życia jest możliwość bezpośredniego dostępu do treści HTML (za pomocą API DOM), bez potrzeby korzystania z dyrektyw tutejże innych funkcji Vue.js. Takie działanie może jednak negatywnie wpłynąć na projekt aplikacji, utrudniając testowanie i zarządzanie.
Czy są jakieś rozwiązania alternatywne?	Nie ma obowiązku zajmowania się cyklem życia komponentu, a w wielu projektach w ogóle nie będziesz implementować metod powiadomień.

Tabela 17.2 podsumowuje rozdział.

Tabela 17.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Chcę otrzymać powiadomienie w momencie utworzenia komponentu.	Zaimplementuj metody <code>beforeCreate</code> lub <code>created</code> .	17.8
Chcę otrzymać powiadomienie, gdy mogę skorzystać z modelu DOM	Zaimplementuj metody <code>beforeMount</code> lub <code>mounted</code> .	17.9 – 17.10
Chcę otrzymać powiadomienie, gdy zmienia się właściwość danych.	Zaimplementuj metody <code>beforeUpdate</code> lub <code>updated</code> .	17.11 – 17.12
Chcę wykonać kod po wykonaniu aktualizacji.	Skorzystaj z metody <code>Vue.nextTick</code> .	17.13
Chcę otrzymać powiadomienie dla pojedynczych właściwości danych.	Skorzystaj z obserwatora.	17.14
Chcę otrzymać powiadomienie, gdy komponent jest niszczony.	Zaimplementuj metodę <code>beforeDestroy</code> lub <code>destroyed</code> .	17.15 – 17.16
Chcę otrzymać powiadomienie, gdy dochodzi do błędu.	Zaimplementuj metodę <code>errorCaptured</code> .	17.17 – 17.18

Przygotowania do tego rozdziału

Aby uruchomić przykłady z tego rozdziału, wykonaj polecenie z listingu 17.1 w wybranej lokalizacji — zostanie utworzony nowy projekt Vue.js.

Listing 17.1. Tworzenie nowego projektu

```
vue create lifecycles --default
```

- Wskazówka Projekt zawiera domyślnie plik `HelloWorld.vue` w katalogu `src/components`, jednak nie korzystam z niego w tym rozdziale — możesz go zignorować lub usunąć.

To polecenie utworzy projekt `lifecycles`. Po utworzeniu projektu dodaj w katalogu `lifecycles` plik `vue.config.js` o treści jak w listingu 17.2. Ten plik pozwoli nam definiować szablony w formie zwykłych tekstów (por. rozdział 10.), z czego korzystam w tym rozdziale.

Listing 17.2. Zawartość pliku `lifecycles/vue.config.js`

```
module.exports = {
  runtimeCompiler: true
}
```

Dodaj instrukcję z listingu 17.3 w sekcji `lint` pliku `package.json`, aby wyłączyć regułę, która generuje ostrzeżenia podczas stosowania konsoli JavaScript. W tym rozdziale często korzystam z tej funkcji, więc wyświetlanie ostrzeżeń nie jest nam potrzebne.

Listing 17.3. Wyłączanie reguły lintera w pliku `lifecycles/package.json`

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  }
}
```

```

},
"extends": [
    "plugin:vue/essential",
    "eslint:recommended"
],
"rules": {
    "no-console": "off"
},
"parserOptions": {
    "parser": "babel-eslint"
}
},
...

```

Wykonaj polecenie z listingu 17.4, aby dodać do projektu pakiet Bootstrap CSS.

Listing 17.4. Dodawanie pakietu Bootstrap CSS

```
npm install bootstrap@4.0.0
```

Dodaj instrukcje z listingu 17.5 do pliku *main.js* w katalogu *src*, aby dołączyć Bootstrapa do aplikacji.

Listing 17.5. Dołączanie pakietu Bootstrap do pliku src/main.js

```

import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
    render: h => h(App)
}).$mount('#app')

```

Ostatnim krokiem przygotowawczym jest zamiana treści komponentu głównego (listing 17.6).

Listing 17.6. Zamiana treści pliku src/App.vue

```

<template>
    <div class="bg-primary text-white m-2 p-2">
        <div class="form-check">
            <input class="form-check-input" type="checkbox" v-model="checked" />
            <label>Pole checkbox</label>
        </div>
        Stan zaznaczenia: {{ checked }}
    </div>
</template>
<script>
export default {
    name: 'App',
    data: function () {
        return {
            checked: true
        }
    }
}
</script>

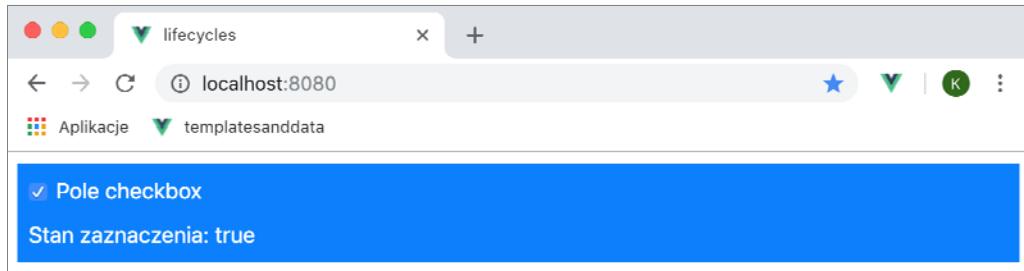
```

Wykonaj polecenie z listingu 17.7 w katalogu *lifecycles*, aby uruchomić narzędzia deweloperskie.

Listing 17.7. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Po przeprowadzeniu inicjalizacji wejdź na stronę <http://localhost:8080>, aby zapoznać się z początkowym stanem aplikacji (rysunek 17.1).



Rysunek 17.1. Przykładowa aplikacja po uruchomieniu

Omówienie cyklu życia komponentu

Cykl życia komponentu rozpoczyna się, gdy Vue.js inicjalizuje komponent. W skład cyklu życia wchodzą wiele czynności: przygotowanie właściwości danych, przetwarzanie szablonów, obsługa zmiany danych, a także — niszczenie komponentu, gdy nie jest on już potrzebny. Na każdym etapie cyklu życia Vue.js dostarcza metody, które zostaną wywołane, jeśli tylko w komponencie zostaną umieszczone ich deklaracje. W tabeli 17.3 opisuję każdą z metod cyklu życia komponentu.

Tabela 17.3. Metody cyklu życia komponentu

Nazwa	Opis
beforeCreate	Ta metoda jest wykonywana przed inicjalizacją komponentu (por. punkt „Omówienie fazy tworzenia”).
created	Ta metoda jest wykonywana po inicjalizacji komponentu (por. punkt „Omówienie fazy tworzenia”).
beforeMount	Ta metoda jest wykonywana przed przetworzeniem szablonu komponentu (por. punkt „Omówienie fazy montażu”).
mounted	Ta metoda jest wykonywana po przetworzeniu szablonu komponentu (por. punkt „Omówienie fazy montażu”).
beforeUpdate	Ta metoda jest wykonywana przed przetworzeniem aktualizacji danych komponentu (por. punkt „Omówienie fazy aktualizacji”).
updated	Ta metoda jest wykonywana po przetworzeniu aktualizacji danych komponentu (por. punkt „Omówienie fazy aktualizacji”).
activated	Ta metoda jest wykonywana w momencie aktywacji elementu, który był utrzymywany w działaniu za pomocą mechanizmu <code>keep-alive</code> (por. rozdział 22.).
deactivated	Ta metoda jest wykonywana w momencie dezaktywacji elementu, który był utrzymywany w działaniu za pomocą mechanizmu <code>keep-alive</code> (por. rozdział 22.).
beforeDestroy	Ta metoda jest wykonywana przed zniszczeniem komponentu (por. punkt „Omówienie fazy zniszczenia”).
destroyed	Ta metoda jest wykonywana po zniszczeniu komponentu (por. punkt „Omówienie fazy zniszczenia”).
errorCaptured	Ta metoda pozwala komponentowi na obsługę błędów rzuconych przez jedno z jego dzieci (por. podrozdział „Obsługa błędów komponentów”).

Omówienie fazy tworzenia

To pierwsza faza cyklu życia komponentu. Vue.js tworzy w niej instance komponentu i przygotowuje ją do użycia, włączając w to przetworzenie właściwości, takich jak `data` czy `computed`, w elemencie `script`. Po utworzeniu komponentu — ale przed przetworzeniem obiektu konfiguracji — Vue.js wywołuje metodę `beforeCreate`. Po przetworzeniu właściwości konfiguracji, w tym także właściwości danych, następuje wywołanie metody `created`. W listingu 17.8 dodaję obie metody dla tej fazy cyklu życia komponentu.

Listing 17.8. Metody cyklu życia związane z tworzeniem komponentu (src/App.vue)

```
<template>
  <div class="bg-primary text-white m-2 p-2">
    <div class="form-check">
      <input class="form-check-input" type="checkbox" v-model="checked" />
      <label>Pole checkbox</label>
    </div>
    Stan zaznaczenia: {{ checked }}
  </div>
</template>
<script>
export default {
  name: 'App',
  data: function () {
    return {
      checked: true
    }
  },
  beforeCreate() {
    console.log("Wywołanie metody beforeCreate " + this.checked);
  },
  created() {
    console.log("Wywołanie metody created" + this.checked);
  }
}
</script>
```

W tym listingu dodałem metody `beforeCreate` i `created`, które wyświetlają komunikat w konsoli przeglądarki JavaScript z informacją o stanie właściwości `checked`.

- **Uwaga** Metody cyklu życia są definiowane bezpośrednio w obiekcie, w elemencie `script`, a nie we właściwości `methods`.

W czasie pomiędzy wywołaniami metod `beforeCreate` i `created` Vue.js ustawia mechanizmy, które czynią komponent użytecznym, w tym metody i właściwości danych (co widać po wydruku w konsoli JavaScript):

```
...
Wywołanie metody beforeCreate undefined
Wywołanie metody created true
...
```

W momencie wywołania metody `beforeCreate` konfiguracja właściwości `data` nie została jeszcze wykonana, dlatego wartość właściwości `checked` to `undefined`. W momencie wywołania metody `created` czynność ta jest już za nami, dlatego właściwość `checked` ma przypisaną początkową wartość.

Omówienie procesu tworzenia obiektów komponentów

Komunikat wyświetlany w metodzie beforeCreate pokazuje, że komponent został utworzony i przypisany do specjalnej zmiennej this przed wywołaniem metody. Obiekt przypisany do this to komponent. Jeśli w aplikacji występuje wiele instancji komponentów, obiekt zostanie utworzony dla każdej z nich. Gdy Vue.js przechodzi przez proces inicjalizacji, to właśnie do tego obiektu są przypisywane właściwości data, computed i metody. Definicja metody polega na skorzystaniu z właściwości methods obiektu konfiguracji użytego podczas tworzenia komponentu. W trakcie inicjalizacji Vue.js przypisuje zdefiniowaną przez Ciebie funkcję do obiektu komponentu, dzięki czemu możesz wykonać ją za pomocą instrukcji this.myMethod() bez konieczności zagłębiania się w strukturę obiektu konfiguracji. Tworzony przez Vue.js obiekt komponentu na początku nie ma zbyt wielu użytecznych funkcji. W większości projektów zaimplementowanie metody zdarzenia beforeCreate nie będzie konieczne.

Omówienie przygotowań związanych z reaktywnością

Jedną z najważniejszych możliwości Vue.js jest reaktywność. Dzięki niej zmiana właściwości danych jest automatycznie propagowana w całej aplikacji, co wyzwala zmiany we właściwościach obliczanych, wiązaniach danych i propach — pozwala to mieć pewność, że cała aplikacja dysponuje najświeższymi danymi.

Podczas fazy tworzenia Vue.js przetwarza obiekt konfiguracji umieszczony w elemencie script. Obiekt komponentu otrzymuje właściwość dla każdej właściwości umieszczonej w sekcji data, wraz z akcesoriami dostępu (getter/setter), dzięki czemu Vue.js „wie” o każdej próbie odczytu i zapisu. To sprawia, że Vue.js „wie”, kiedy dochodzi do użycia właściwości i jest w stanie aktualizować wybrane części aplikacji. Oznacza to także, że musisz mieć pewność, iż wszystkie właściwości w sekcji data są zdefiniowane, zanim Vue.js zakończy fazę tworzenia.

-
- **Wskazówka** Jeśli aplikacja korzysta z zewnętrznych danych, takich jak REST-owe API, użytecznego sposobu pobrania danych dostarcza metoda created (por. rozdział 20.).
-

Omówienie fazy montażu

Podczas drugiej fazy cyklu życia komponentu Vue.js zajmuje się szablonem komponentu, obsługą wiązań danych, dyrektywami i innymi funkcjami aplikacji.

Dostęp do obiektowego modelu dokumentu

Jeśli komponent musi użyć obiektowego modelu dokumentu (DOM), warto skorzystać ze zdarzenia mounted, które informuje o przetworzeniu treści komponentu. Treść jest dostępna za pomocą właściwości \$el, w której Vue.js przechowuje obiekt komponentu.

W listingu 17.9 uzyskuję dostęp do modelu DOM, aby pobrać wartości danych dostarczone za pomocą atrybutu w elemencie HTML, który wdraża komponent. Jak wyjaśniłem w rozdziale 16., dowolny atrybut zastosowany w elemencie HTML, który wdraża komponent, zostanie przeniesiony do nadrzędnego elementu w szablonie komponentu.

Listing 17.9. Dostęp do modelu DOM w pliku src/App.vue

```
<template>
  <div class="bg-primary text-white m-2 p-2">
    <div class="form-check">
      <input class="form-check-input" type="checkbox" v-model="checked" />
      <label>Pole checkbox</label>
    </div>
    Stan zaznaczenia: {{ checked }}
    <div class="bg-info p-2">
      Imiona:
```

```

<ul>
    <li v-for="name in names" v-bind:key="name">
        {{ name }}
    </li>
</ul>
</div>
</template>
<script>
    export default {
        name: 'App',
        data: function () {
            return {
                checked: true,
                names: []
            }
        },
        beforeCreate() {
            console.log("Wywołanie metody beforeCreate" + this.checked);
        },
        created() {
            console.log("Wywołanie metody created" + this.checked);
        },
        mounted() {
            this.$el.dataset.names.split(",").forEach(name => this.names.push(name));
        }
    }
</script>

```

- **Ostrzeżenie** Dostęp do modelu DOM powinien być osiągany tylko, jeśli nie ma żadnej rozsądnej alternatywy. W tym przykładzie dostarczamy dane za pomocą atrybutów — jest to użyteczne zachowanie zwłaszcza w środowisku, w którym dokumenty HTML są generowane dynamicznie z poziomu kodu. Takie podejście powinno być traktowane wyłącznie jako tymczasowe, dopóki nie zostanie zaimplementowane podejście docelowe (np. API w formacie REST, o czym piszę w rozdziale 20.).

Metoda `mounted` zostanie wykonana po przetworzeniu szablonu i dodaniu jego treści do drzewa DOM. W tym samym czasie atrybuty dodane do własnego elementu HTML (powiązanego z komponentem) zostaną przeniesione do nadrzędnego elementu `div` z listingu 17.9. Wewnątrz metody `mounted` korzystam z właściwości `this.$el`, aby uzyskać dostęp do atrybutów `data-` w celu wykonania ich odczytu, utworzenia z nich tablicy, a następnie umieszczenia elementów we właściwości danych `names`. Ich wartości są wyświetlane na liście za pomocą dyrektywy `v-for`.

- **Wskazówka** Zwróć uwagę, że zdefiniowałem właściwość `names` i przypisałem do niej pustą tablicę (w listingu 17.9). Niezwykle ważne jest, aby zdefiniować wszystkie właściwości w sekcji `data`, zanim zostaną przetworzone w celu włączenia reaktywności — inaczej zmiany nie będą wykrywane.

W listingu 17.10 dodałem atrybut `data-name` do elementu HTML, który wdraża komponent.

Listing 17.10. Dodawanie atrybutu do pliku `src/main.js`

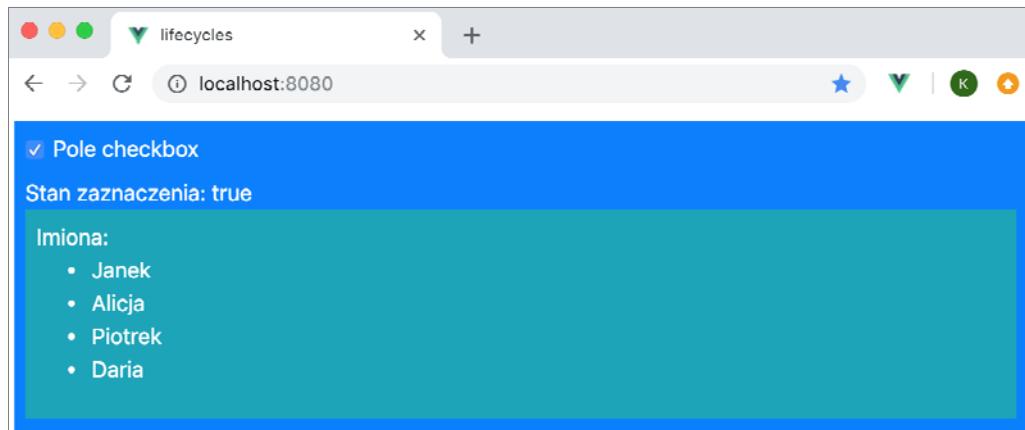
```

import Vue from 'vue'
import App from './App'
import "bootstrap/dist/css/bootstrap.min.css";

```

```
Vue.config.productionTip = false
new Vue({
  el: '#app',
  components: { App },
  template: '<App data-names="Janek, Alicja, Piotrek, Daria" />'
})
```

W rezultacie komponent skorzysta z API modelu DOM, dostarczonego przez przeglądarkę, aby odczytać atrybut wdrożony przez element. Następnie skorzysta z jego zawartości, aby ustawić właściwość danych (rysunek 17.2).



Rysunek 17.2. Dostęp do modelu DOM w czasie fazy montażu

Omówienie fazy aktualizacji

Po inicjalizacji komponentu i „podmontowaniu” jego treści rozpoczyna się faza aktualizacji. W jej trakcie Vue.js reaguje na zmiany, które zostały wykonane wobec właściwości danych. Gdy zmiana zostanie wykryta, Vue.js wywoła metodę beforeUpdate komponentu, a następnie, po zakończeniu aktualizacji i zmodyfikowaniu kodu HTML, wykona metodę updated komponentu.

W większości projektów metody beforeUpdate i updated, dające dostęp do modelu DOM przed wprowadzeniem do niego zmian i po ich wprowadzeniu, nie będą Ci potrzebne. Podobnie jak w przypadku bezpośredniego dostępu do modelu DOM, korzystanie z możliwości oferowanych przez te metody powinno być ostatecznością — stosowaną tylko, jeśli standardowe funkcje Vue.js okazują się niewystarczające. W listingu 17.11 implementuję obie metody tej fazy.

Listing 17.11. Implementacja metod fazy aktualizacji (src/App.vue)

```
...
<script>
  export default {
    name: 'App',
    data: function () {
      return {
        checked: true,
        names: []
      }
    },
    beforeCreate() {
      console.log("Wywołano metodę beforeCreate" + this.checked);
    }
  }
}
```

```

    },
    created() {
      console.log("Wywołano metodę created" + this.checked);
    },
    mounted() {
      this.$el.dataset.names.split(",").forEach(name => this.names.push(name));
    },
    beforeUpdate() {
      console.log(`Wywołano metodę beforeUpdate. Stan zaznaczenia: ${this.checked}` +
        ` Imię: ${this.names[0]} Elementy listy: ` +
        ` ${this.$el.getElementsByTagName("li").length}`);
    },
    updated() {
      console.log(`Wywołano metodę updated. Stan zaznaczenia: ${this.checked}` +
        ` Imię: ${this.names[0]} Elementy listy: ` +
        ` ${this.$el.getElementsByTagName("li").length}`);
    }
  }
</script>
...

```

Metody `beforeUpdate` i `updated` wypisują w konsoli JavaScript komunikat, który zawiera wartość właściwości `checked`. Właściwość ta zawiera pierwszą wartość tablicy `names` i pewną liczbę elementów `li`, do których mam dostęp z poziomu modelu DOM. Po zapisaniu zmian w komponencie w konsoli JavaScript przeglądarki zobaczysz następujące komunikaty:

```

...
Wywołano metodę beforeCreate undefined
Wywołano metodę created true
Wywołano metodę beforeUpdate. Stan zaznaczenia: true Imię: Janek Elementy listy: 0
Wywołano metodę updated. Stan zaznaczenia: true Imię: Janek Elementy listy: 4
...

```

Ta sekwencja komunikatów demonstruje proces inicjalizacji komponentu Vue.js. W metodzie `created` dodaje wartości do właściwości `data` komponentu. W rezultacie zmianie ulega kod HTML wyświetlany użytkownikowi. W momencie wywołania metody `beforeUpdate` nie ma dostępnych elementów `li`, ale już w momencie wywołania metody `updated` Vue.js wieńczy proces aktualizacji i mamy dostęp do 4 elementów `li`.

Omówienie konsolidacji aktualizacji

Zwróć uwagę, że mimo dodania czterech elementów do tablicy `names` metody `beforeUpdate` i `updated` są wykonywane tylko raz. Vue.js nie reaguje na pojedyncze zmiany danych, które, powodując szereg zmian w modelu DOM, byłyby niezwykle kosztowne do wykonania. Alternatywnie Vue.js zarządza kolejką oczekujących zmian. Dzięki temu kolejka jest w stanie usunąć duplikaty zmian i pozwala na grupowanie podobnych operacji. Aby pokazać to działanie w praktyce, dodajemy element `button` do komponentu, co prowadzi do zmian obu właściwości `data` (listing 17.12).

Listing 17.12. Dokonywanie wielu zmian (src/App.vue)

```

<template>
  <div class="bg-primary text-white m-2 p-2">
    <div class="form-check">
      <input class="form-check-input" type="checkbox" v-model="checked" />
      <label>Pole checkbox</label>
    </div>
    Stan zaznaczenia: {{ checked }}
    <div class="bg-info p-2">
      Imiona:
      <ul>

```

```

<li v-for="name in names" v-bind:key="name">
    {{ name }}
</li>
</ul>
</div>
<div class="text-white center my-2">
    <button class="btn btn-light" v-on:click="doChange">
        Zmień
    </button>
</div>
</div>
</template>
<script>
    export default {
        name: 'App',
        data: function () {
            return {
                checked: true,
                names: []
            }
        },
        beforeCreate() {
            console.log("Wywołano metodę beforeCreate" + this.checked);
        },
        created() {
            console.log("Wywołano metodę created" + this.checked);
        },
        mounted() {
            this.$el.dataset.names.split(",").forEach(name => this.names.push(name));
        },
        beforeUpdate() {
            console.log(`Wywołano metodę beforeUpdate. Stan zaznaczenia: ${this.checked}` +
                ` Imię: ${this.names[0]} Elementy listy: ` +
                ` ${this.$el.getElementsByTagName("li").length}`);
        },
        updated() {
            console.log(`Wywołano metodę updated. Stan zaznaczenia: ${this.checked}` +
                ` Imię: ${this.names[0]} Elementy listy: ` +
                ` ${this.$el.getElementsByTagName("li").length}`);
        },
        methods: {
            doChange() {
                this.checked = !this.checked;
                this.names.reverse();
            }
        }
    }
</script>

```

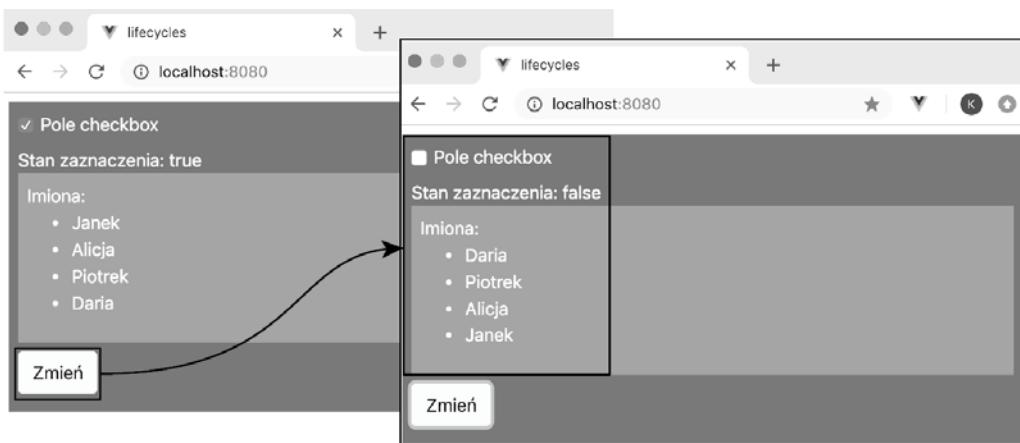
Kliknięcie przycisku wywołuje metodę doChange, która przełącza wartość właściwości checked i odwraca kolejność elementów w tablicy names (rysunek 17.3).

Mimo dokonania zmian w obrębie dwóch właściwości danych komunikaty wyświetlane w konsoli JavaScript pokazują, że dokonano tylko jednej zmiany w treści HTML komponentu:

```

...
Wykonano metodę beforeUpdate. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
Wywołano metodę updated. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
...

```



Rysunek 17.3. Dokonywanie wielu zmian

- **Wskazówka** Metody beforeUpdate i updated są wywoływanne tylko wtedy, gdy konieczna jest zmiana w kodzie HTML jednego lub większej liczby elementów. Jeśli dokonasz zmiany wobec właściwości danych, która nie wymaga zmiany w kodzie HTML, metody nie zostaną wykonane.

Stosowanie wywołania zwrotnego po aktualizacji

Jedną z konsekwencji sposobu działania Vue.js jest fakt, że nie możesz oczekiwac, iż drzewo DOM będzie odzwierciedlać efekty Twoich zmian wprowadzonych we właściwościach danych. Vue.js dostarcza metodę `Vue.nextTick`, która może być używana do wykonywania pewnych czynności po zastosowaniu wszystkich oczekujących zadań. W listingu 17.13 korzystam z funkcji `nextTick` i metody `doChange`, aby zademonstrować kolejność akcji i wywołań metod.

Listing 17.13. Zastosowanie wywołania zwrotnego po aktualizacji (src/App.vue)

```
...
<script>
    import Vue from "vue";
    export default {
        name: 'App',
        data: function () {
            return {
                checked: true,
                names: []
            }
        },
        beforeCreate() {
            console.log("Wywołano metodę beforeCreate" + this.checked);
        },
        created() {
            console.log("Wywołano metodę created" + this.checked);
        },
        mounted() {
            this.$el.dataset.names.split(",").forEach(name => this.names.push(name));
        },
        beforeUpdate() {
            console.log(`Wywołano metodę beforeUpdate. Stan zaznaczenia: ${this.checked}`);
        }
    }
</script>
```

```

        + ` Imię: ${this.names[0]} Elementy listy: `
        + this.$el.getElementsByTagName("li").length);
    },
    updated() {
        console.log(`Wywołano metodę updated. Stan zaznaczenia: ${this.checked}`
        + ` Imię: ${this.names[0]} Elementy listy: `
        + this.$el.getElementsByTagName("li").length);
    },
    methods: {
        doChange() {
            this.checked = !this.checked;
            this.names.reverse();
            Vue.nextTick(() => console.log("Wykonano wywołanie zwrotne"));
        }
    }
</script>
...

```

Metoda `nextTick` akceptuje funkcję i opcjonalnie obiekt kontekstu. Oprócz tego możemy przekazać funkcję, która zostanie wykonana na końcu następnego cyklu aktualizacji. Zapisz zmiany i zapoznaj się z sekwencją komunikatów wyświetlanych w konsoli przeglądarki JavaScript:

```

...
Wykonano metodę beforeUpdated. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
Wykonano metodę updated. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
Wykonano wywołanie zwrotne
...

```

Zmiana, która doprowadziła do aktualizacji, jest wyzwalana przez metodę `doChange`. Wywołanie metody `nextTick` zapewnia, że wywołanie zwrotne zostanie wykonane po wdrożeniu zmian przez model DOM.

Obserwowanie zmian danych za pomocą obserwatorów

Metody `beforeUpdate` i `update` informują o momencie aktualizacji elementów HTML, jednak nie dostajemy informacji o charakterze zmiany. Metody te nie zostaną wykonane, jeśli zmiana dotyczy właściwości `data`, która nie jest używana w wyrażeniu dyrektywy lub wiązaniu danych.

Jeśli chcesz otrzymywać powiadomienia o zmianie właściwości danych, skorzystaj z obserwatora. W listingu 17.14 dodaj obserwatora do obsługi powiadomień dotyczących zmiany właściwości `checked` w obrębie sekcji `data`.

Listing 17.14. Zastosowanie obserwatora w pliku `src/App.vue`

```

...
<script>
    import Vue from "vue";
    export default {
        name: 'App',
        data: function () {
            return {
                checked: true,
                names: []
            }
        },
        name: 'App',

```

```

data: function () {
    return {
        checked: true,
        names: []
    }
},
beforeCreate() {
    console.log("Wywołano metodę beforeCreate" + this.checked);
},
created() {
    console.log("Wywołano metodę created" + this.checked);
},
mounted() {
    this.$el.dataset.names.split(",").forEach(name => this.names.push(name));
},
beforeUpdate() {
    console.log(`Wywołano metodę beforeUpdate. Stan zaznaczenia: ${this.checked}` +
        ` Imię: ${this.names[0]} Elementy listy: ` +
        ` ${this.$el.getElementsByTagName("li").length}`);
},
updated() {
    console.log(`Wywołano metodę updated. Stan zaznaczenia: ${this.checked}` +
        ` Imię: ${this.names[0]} Elementy listy: ` +
        ` ${this.$el.getElementsByTagName("li").length}`);
},
methods: {
    doChange() {
        this.checked = !this.checked;
        this.names.reverse();
        Vue.nextTick(() => console.log("Wykonano wywołanie zwrotne"));
    }
},
watch: {
    checked: function (newValue, oldValue) {
        console.log(`Sprawdzono obserwatora, Stara wartość: ${oldValue}, Nowa wartość:
        ↵${newValue}`);
    }
}
}
</script>
...

```

Obserwatorów definiuje się za pomocą właściwości `watch` w obiekcie konfiguracji komponentu. Właściwość ta otrzymuje obiekt zawierający po jednej właściwości dla każdej właściwości, którą chcemy obserwować. Obserwator wymaga podania nazwy właściwości i funkcji obsługi, która zostanie wywołana z nową i starą wartością. W listingu zdefiniowałem obserwatora właściwości `checked`, który wypisuje obie wartości w konsoli przeglądarki. Zapisz zmiany i odznacz przycisk wyboru, a zobaczysz następujący ciąg komunikatów, w tym jeden z obserwatora:

```

...
Sprawdzono obserwatora, Stara wartość: true, Nowa wartość: false
Wywołano metodę beforeUpdate. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
Wywołano metodę updated. Stan zaznaczenia: false Imię: Daria Elementy listy: 4
...

```

Zwróć uwagę, że funkcja obsługi obserwatora jest wywoływana przed metodami `beforeUpdate` i `updated`.

Omówienie opcji obserwatora

Zachowanie obserwatora możemy zmienić za pomocą dwóch opcji. Ich użycie wymaga zadeklarowania obserwatora w nieco inny sposób. Pierwsza opcja to `immediate` — informuje ona Vue.js o konieczności wykonania funkcji obsługującej obserwatora tuż po jego skonfigurowaniu, podczas fazy tworzenia (pomiędzy wywołaniami metod `beforeCreate` i `created`). Aby skorzystać z tej funkcji, obserwator musi być wyrażony jako obiekt z właściwościami `handler` i `immediate`, np.:

```
...
watch: {
    checked: {
        handler: function (newValue, oldValue) {
            // w tym miejscu możesz zareagować na zmiany
        },
        immediate: true
    }
}
...
```

Druga opcja to `deep` — pozwala ona na obserwowanie wszystkich właściwości zdefiniowanych w obiekcie przypisany do właściwości `data`. Jest to lepsze rozwiązanie niż tworzenie obserwatora dla każdej właściwości. Oto przykład użycia tej opcji:

```
...
watch: {
    myObject: {
        handler: function (newValue, oldValue) {
            // w tym miejscu możesz zareagować na zmiany
        },
        deep: true
    }
}
...
```

Opcja `deep` nie może być używana do tworzenia obserwatora dla wszystkich właściwości danych — dotyczy to wyłącznie pojedynczych właściwości, do których został przypisany obiekt. Gdy stosujesz funkcję dla obserwatora, nie możesz korzystać z wyrażeń strzałkowych, ponieważ do słowa `this` nie zostanie przypisany komponent, którego wartość uległa zmianie. W związku z tym skorzystaj ze słowa `function`, jak w przykładzie.

Omówienie fazy zniszczenia

Ostatnią fazą cyklu życia komponentu jest jego zniszczenie, a następuje to w momencie wyświetlenia komponentu, gdy nie jest on już dłużej potrzebny (por. rozdział 22.). Vue.js wykona metodę `beforeDestroy`, aby umożliwić komponentowi przygotowanie się do zakończenia cyklu życia. Metoda `destroy` zostanie wykonana po zakończeniu procesu zniszczenia, czyli po usunięciu obserwatorów funkcji obsługi zdarzeń i wszelkich komponentów-dzieci.

Aby lepiej zapoznać się z tą częścią cyklu życia, dodaję plik `MessageDisplay.vue` do katalogu `src/components` (listing 17.15).

Listing 17.15. Zawartość pliku *src/components/MessageDisplay.vue*

```
<template>
  <div class="bg-dark text-light text-center p-2">
    <div>
      Stan licznika: {{ counter }}
    </div>
    <button class="btn btn-secondary" v-on:click="handleClick">
      Zwiększ
    </button>
  </div>
</template>
<script>
export default {
  data: function () {
    return {
      counter: 0
    }
  },
  created: function() {
    console.log("MessageDisplay: created");
  },
  beforeDestroy: function() {
    console.log("MessageDisplay: beforeDestroy");
  },
  destroyed: function() {
    console.log("MessageDisplay: destroyed");
  },
  methods: {
    handleClick() {
      this.counter++;
    }
  }
}
</script>
```

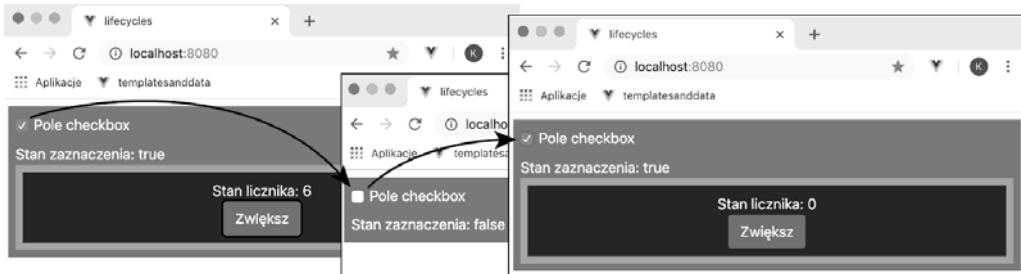
Ten komponent wyświetla licznik, który można zwiększyć przez kliknięcie przycisku. Ponadto implementuję metody `created`, `beforeDestroy` i `destroyed`, dzięki czemu możemy zaobserwować cykl życia komponentu w konsoli JavaScript. W listingu 17.16 upraszczam komponent główny aplikacji. Nowy komponent wyświetlę tylko, gdy wartość danych o nazwie `checked` jest równa `true`.

Listing 17.16. Wdrożenie komponentu-dziecka w pliku *src/App.vue*

```
<template>
  <div class="bg-primary text-white m-2 p-2">
    <div class="form-check">
      <input class="form-check-input" type="checkbox" v-model="checked" />
      <label>Pole checkbox</label>
    </div>
    Stan zaznaczenia: {{ checked }}
    <div class="bg-info p-2" v-if="checked">
      <message-display></message-display>
    </div>
  </div>
</template>
<script>
  import MessageDisplay from "./components/MessageDisplay"
```

```
export default {
  name: 'App',
  components: { MessageDisplay },
  data: function () {
    return {
      checked: true,
      names: []
    }
  }
}
</script>
```

Po zmianie stanu zaznaczenia Vue.js utworzy i usunie komponent MessageDisplay (rysunek 17.4).



Rysunek 17.4. Tworzenie i niszczenie komponentu

- **Ostrzeżenie** Nie powinno się przekazywać ważnych zadań do fazy zniszczenia, ponieważ Vue.js nie będzie w stanie zakończyć cyklu życia komponentu. Jeśli użytkownik przejdzie pod inny adres URL, aplikacja zostanie przerwana bez możliwości zakończenia cyklu życia swoich komponentów.

Po odznamieniu pola wyboru (*checkbox*) komponent-dziecko jest niszczony, a jego treść usuwana z drzewa DOM. Po zaznaczeniu pola wyboru tworzony jest nowy komponent, a cykl życia rozpoczyna się od nowa. Jeśli przeanalizujesz konsolę JavaScript przeglądarki, zobaczysz komunikaty podobne do poniższych w miarę tworzenia i niszczenia kolejnych komponentów:

```
...
MessageDisplay: beforeDestroy
MessageDisplay: destroyed
MessageDisplay: created
...
```

Kliknięcie przycisku *Zwiększ* przed przełączeniem przycisku wyboru spowoduje utratę stanu komponentu w momencie jego zniszczenia. W rozdziale 21. opisuję metodę tworzenia współdzielonego stanu aplikacji, który przetrwa zmiany wynikające z cyklu życia komponentów.

Obsługa błędów komponentów

Jedna z faz nie daje się umiejscowić w jednym miejscu standardowego cyklu życia komponentów. Mowa o obsłudze błędów pochodzących z komponentów-dzieci. Aby przedstawić tę fazę, w listingu 17.17 dodaję elementy i kod do komponentu *MessageDisplay*. Spowoduje to wygenerowanie błędu w momencie kliknięcia przycisku.

Listing 17.17. Generowanie błędu w pliku *src/components/MessageDisplay.vue*

```

<template>
  <div class="bg-dark text-light text-center p-2">
    <div>
      Stan licznika: {{ counter }}
    </div>
    <button class="btn btn-secondary" v-on:click="handleClick">
      Zwiększ
    </button>
    <button class="btn btn-secondary" v-on:click="generateError">
      Wygeneruj błąd
    </button>
  </div>
</template>
<script>
export default {
  data: function () {
    return {
      counter_base: 0,
      generate_error: false
    }
  },
  created: function() {
    console.log("MessageDisplay: created");
  },
  beforeDestroy: function() {
    console.log("MessageDisplay: beforeDestroy");
  },
  destroyed: function() {
    console.log("MessageDisplay: destroyed");
  },
  methods: {
    handleClick() {
      this.counter_base++;
    },
    generateError() {
      this.generate_error = true;
    }
  },
  computed: {
    counter() {
      if (this.generate_error) {
        throw "Mój błąd w komponencie";
      } else {
        return this.counter_base;
      }
    }
  }
}
</script>

```

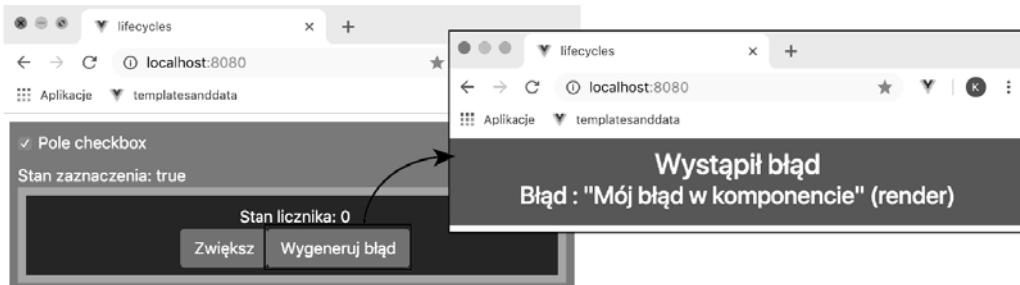
Obsługa błędów w komponencie może poradzić sobie z błędami, które powstają, gdy dane komponentu są modyfikowane, gdy wykonywana jest funkcja obserwatora lub gdy wywoływana jest jedna z metod cyklu życia. Nie zadziała natomiast, gdy do błędu dojdzie w momencie obsługi zdarzenia, dlatego kod z listingu 17.17 reaguje na kliknięcie przycisku przez ustawienie właściwości `data`, co w konsekwencji powoduje wygenerowanie błędu we właściwości obliczonej `counter`.

Aby obsłużyć błędy swoich dzieci, komponent implementuje metodę `errorCaptured`, która wprowadza trzy parametry: błąd, komponent, który go wygenerował, i tekst opisujący, co działa się w Vue.js w momencie rzucenia błędu. Jeśli metoda `errorCaptured` zwróci `false`, błąd nie będzie dalej propagowany, dzięki czemu nie musimy się obawiać, że wiele komponentów zareaguje na ten sam błąd. W listingu 17.18 implementuję metodę `errorCaptured` w komponencie głównym aplikacji, dodając elementy do szablonu, aby wyświetlić szczegółowy błąd.

Listing 17.18. Obsługa błędu w pliku src/App.vue

```
<template>
<div class="bg-danger text-white text-center h3 p-2" v-if="error.occurred">
    Wystąpił błąd
    <h4>
        Błąd : "{{ error.error }}" ({{ error.source }})
    </h4>
</div>
<div v-else class="bg-primary text-white m-2 p-2">
    <div class="form-check">
        <input class="form-check-input" type="checkbox" v-model="checked" />
        <label>Pole checkbox</label>
    </div>
    Stan zaznaczenia: {{ checked }}
    <div class="bg-info p-2" v-if="checked">
        <message-display></message-display>
    </div>
</div>
</template>
<script>
    import MessageDisplay from "./components/MessageDisplay"
    export default {
        name: 'App',
        components: { MessageDisplay },
        data: function () {
            return {
                checked: true,
                names: [],
                error: {
                    occurred: false,
                    error: "",
                    source: ""
                }
            }
        },
        errorCaptured(error, component, source) {
            this.error.occurred = true;
            this.error.error = error;
            this.error.source = source;
            return false;
        }
    }
</script>
```

Metoda `errorCaptured` reaguje na błąd, ustawiając właściwości obiektu danych o nazwie `error`, które są wyświetlane za pomocą dyrektyw `v-if` i `v-else`. Po zapisaniu zmian i kliknięciu przycisku *Wygeneruj błąd* zobaczysz efekt jak na rysunku 17.5.



Rysunek 17.5. Obsługa błędów

Definiowanie globalnej funkcji obsługi błędów

Vue.js dostarcza globalny mechanizm obsługi błędów, który jest używany do obsługi błędów pominiętych w komponentach aplikacji. Globalny mechanizm to funkcja, która otrzymuje te same argumenty co metoda `errorCaptured`, ale jest definiowana w głównym obiekcie Vue w pliku `main.js`, a nie w którymś z komponentów.

Globalna funkcja obsługi błędów umożliwia otrzymanie powiadomienia w sytuacji, gdy żaden element aplikacji nie podjął się działania. Oznacza to, że nie będziesz mógł w rozsądny sposób przywrócić poprzedniego stanu. Jeśli jednak musisz wykonać jakąś czynność, gdy błąd nie został obsłużony nigdzie indziej, zdefiniuj funkcję w pliku `main.js`.

```
...
import Vue from 'vue'
import App from './App'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
Vue.config.errorHandler = function (error, component, source) {
  console.log(`Globalna funkcja obsługi błędów: ${error}, ${component}, ${source}`);
}
new Vue({
  el: '#app',
  components: { App },
  template: `<div><App data-names="Janek, Alicja, Piotr, Daria" /></div>`
})
...
```

Ta funkcja wypisze komunikat w konsoli przeglądarki JavaScript, co jest zachowaniem podobnym do domyślnego. Istnieją usługi śledzące, które pozwalają na zebranie i analizę błędów z wykorzystaniem centralnego serwera. Niektóre rozwiązania oferują wsparcie dla aplikacji Vue.js dzięki globalnej funkcji obsługi błędów, jednak są one kosztowne w przypadku dużej liczby użytkowników aplikacji.

Podsumowanie

W tym rozdziale opisałem cykl życia komponentów Vue.js, a także metody, które pozwalają na otrzymywanie powiadomień na każdym etapie cyklu. Omówiłem przetwarzanie wartości danych, jak uzyskać dostęp do treści HTML komponentu za pomocą modelu DOM, jak obserwować wartości danych przy użyciu obserwatorów i wreszcie jak obsługiwać błędy. W kolejnym rozdziale pokażę, jak nadać właściwą strukturę w aplikacji Vue.js, korzystając z luźno powiązanych komponentów.

ROZDZIAŁ 18.



Luźno powiązane komponenty

W miarę rozwoju aplikacji tworzonej w Vue.js konieczność przekazywania komunikatów pomiędzy rodzicem a dzieckiem staje się coraz trudniejsza do zorganizowania, zwłaszcza gdy trzeba ze sobą skomunikować zupełnie różne części aplikacji. Efektem tego jest poświęcanie zbyt dużej ilości kodu na przekazywanie danych przez propy i zdarzenia, zamiast skupiania się na tym, co dany komponent ma robić. W tym rozdziale opisuję alternatywne podejście znane pod nazwą **wstrzykiwania zależności** (ang. *dependency injection*). Dzięki niemu komponenty mogą komunikować się bez ścisłych powiązań pomiędzy sobą. Pokażę różne sposoby wstrzykiwania zależności, a także jak uwalnia to strukturę aplikacji. Skorzystam również z szyny zdarzeń, która łączy wstrzykiwanie zależności z możliwością nasłuchiwanie zdarzeń z poziomu kodu. Dzięki temu komponent może wysłać dowolne zdarzenie do dowolnego słuchacza, a nie tylko do swojego rodzica. Tabela 18.1 umiejscawia wstrzykiwanie zależności i szyny zdarzeń w szerszym kontekście.

Tabela 18.1. Umiejscowienie wstrzykiwania zależności i szyn zdarzeń w szerszym kontekście

Pytanie	Odpowiedź
Czym są wstrzykiwanie zależności i szyny zdarzeń?	Wstrzykiwanie zależności pozwala na udostępnianie usług, takich jak wartości, obiekty czy funkcje, swoim potomkom. Szyny zdarzeń korzystają z wstrzykiwania zależności w celu udostępniania spójnego mechanizmu do wysyłania lub odbierania własnych zdarzeń.
Dlaczego są użyteczne?	Te mechanizmy pozwalają na tworzenie aplikacji o bardziej skomplikowanej strukturze bez konieczności nadużywania relacji pomiędzy rodzicem a dzieckiem w celu przesyłania danych pomiędzy potomkami a przodkami.
Jak się z nich korzysta?	Komponent definiuje usługi za pomocą właściwości <code>provide</code> , a deklaracja zależności odbywa się za pomocą właściwości <code>inject</code> . Szyny zdarzeń to usługi, które dystrybuują obiekt Vue, za pomocą którego można wysyłać i odbierać zdarzenia przy użyciu metod <code>\$emit</code> i <code>\$on</code> .
Czy są jakieś pułapki lub ograniczenia?	Korzystając z usług i nazw zdarzeń, należy pamiętać o spójnym nazewnictwie w całej aplikacji. Bez odpowiedniej uwagi w tym zakresie różne komponenty mogą powielać nazwy, które są stosowane w innych częściach aplikacji.
Czy są jakieś rozwiązania alternatywne?	Proste aplikacje nie muszą korzystać z funkcji opisanych w tym rozdziale — wystarczające okazują się zwykłe propy i standardowe zdarzenia własne. Złożone aplikacje mogą bazować na współdzielonym stanie aplikacji, opisanym w rozdziale 20.

Tabela 18.2 podsumowuje rozdział.

Tabela 18.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Zdefiniuj mechanizm, który może być używany przez potomków komponentu.	Skorzystaj z funkcji wstrzykiwania zależności, aby zdefiniować usługę.	18.10 – 18.12
Utwórz mechanizm, który zareaguje na zmiany.	Utwórz usługę reaktywną, definiując właściwość danych i stosując ją jako źródło dla wartości danych.	18.13 – 18.14
Zdefiniuj mechanizm, który zostanie zastosowany, jeśli usługa nie zostanie dostarczona.	Zastosuj rozwiązanie zastępcze, korzystając z obiektu z właściwością inject.	18.15 – 18.16
Przekaż własne zdarzenia poza związek pomiędzy rodzicem a dzieckiem.	Skorzystaj z szyny zdarzeń.	18.17
Wyślij zdarzenia za pomocą szyny zdarzeń.	Wywołaj metodę szyny zdarzeń \$emit.	18.18
Odbierz zdarzenia za pomocą szyny zdarzeń.	Wywołaj metodę szyny zdarzeń \$on.	18.19 – 18.20
Przekaż zdarzenie do wybranej części aplikacji.	Utwórz lokalną szynę zdarzeń.	18.21 – 18.22

Przygotowania do tego rozdziału

Aby uruchomić przykłady z tego rozdziału, wykonaj polecenia z listingu 18.1 w wybranej lokalizacji — zostanie utworzony nowy projekt Vue.js.

Listing 18.1. Tworzenie nowego projektu

```
vue create productapp --default
```

■ **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

To polecenie utworzy projekt *productapp*. Po zakończeniu procesu konfiguracji dodaj instrukcje przedstawione w listingu 18.2 do sekcji lintera pliku *package.json*. W ten sposób wyłączysz dwie reguły: pierwszą, odpowiedzialną za wyświetlanie ostrzeżeń o użyciu konsoli JavaScript, i drugą, informującą o zdefiniowaniu zmiennych, które nie zostały użyte. Z konsoli korzystam w wielu przykładach z tego rozdziału. Ponadto zdefiniuję zmienne zastępcze, które zostaną użyte dopiero w dalszej części rozdziału, po wprowadzeniu niezbędnych zagadnień.

Listing 18.2. Wyłączanie reguł lintera w pliku *productapp/package.json*

```
...
"eslintConfig": {
  "root": true,
  "env": {
    "node": true
  },
  "extends": [
    "plugin:vue/essential",
```

```

    "eslint:recommended"
],
"rules": {
  "no-console": "off",
  "no-unused-vars": "off"
},
"parserOptions": {
  "parser": "babel-eslint"
}
},
...

```

Następnie wykonaj polecenie z listingu 18.3 w katalogu *productapp*, aby do projektu dodać pakiet Bootstrap.

Listing 18.3. Instalacja pakietu Bootstrap CSS

```
npm install bootstrap@4.0.0
```

Dodaj instrukcje z listingu 18.4 do pliku *src/main.js*, aby dołączyć Bootstrapa do aplikacji.

Listing 18.4. Dołączanie pakietu Bootstrap w pliku src/main.js

```

import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
  render: h => h(App)
}).$mount('#app')

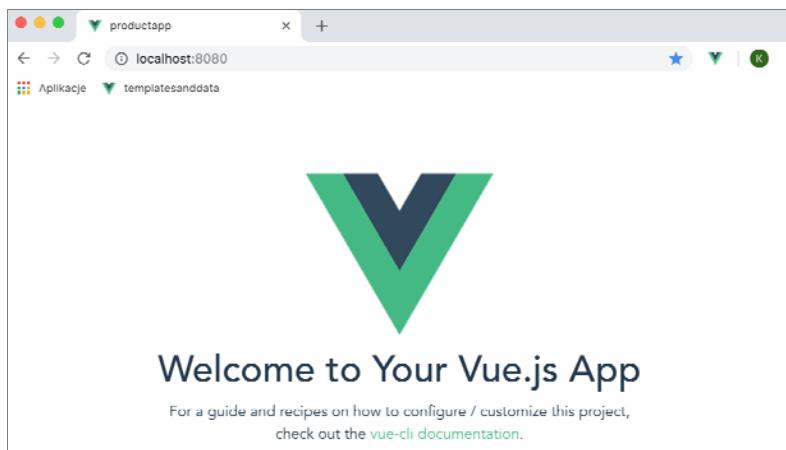
```

Wykonaj polecenie z listingu 18.5 w katalogu, aby uruchomić narzędzia deweloperskie.

Listing 18.5. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Po przeprowadzeniu inicjalizacji wejdź na stronę <http://localhost:8080>, aby zapoznać się z początkowym stanem aplikacji (rysunek 18.1).



Rysunek 18.1. Przykładowa aplikacja po uruchomieniu

Tworzenie komponentu do wyświetlania produktu

Sercem naszej przykładowej aplikacji będzie komponent odpowiedzialny za wyświetlenie tabeli zawierającej szczegółowe informacje na temat produktów wraz z przyciskami pozwalającymi na ich edycję bądź usunięcie. Do katalogu `src/components` dodaję plik `ProductDisplay.vue` o treści jak w listingu 18.6.

Listing 18.6. Zawartość pliku `src/components/ProductDisplay.vue`

```
<template>
<div>
    <table class="table table-sm table-striped table-bordered">
        <thead>
            <tr>
                <th>ID</th><th>Nazwa</th><th>Cena</th><th></th>
            </tr>
        <tbody>
            <tr v-for="p in products" v-bind:key="p.id">
                <td>{{ p.id }}</td>
                <td>{{ p.name }}</td>
                <td>{{ p.price | currency }}</td>
                <td>
                    <button class="btn btn-sm btn-primary"
                           v-on:click="editProduct(p)">
                        Edytuj
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
    <div class="text-center">
        <button class="btn btn-primary" v-on:click="createNew">
            Utwórz nowy
        </button>
    </div>
</div>
</template>
<script>
export default {
    data: function () {
        return {
            products: [
                { id: 1, name: "Kajak", price: 275 },
                { id: 2, name: "Kamizelka ratunkowa", price: 48.95 },
                { id: 3, name: "Piłka nożna", price: 19.50 },
                { id: 4, name: "Chorągiewki narożne", price: 39.95 },
                { id: 5, name: "Stadion", price: 79500 }
            ]
        },
        filters: {
            currency(value) {
                return `${value.toFixed(2)} PLN`;
            }
        },
        methods: {
            createNew() {
            },
            editProduct(product) {
            }
        }
    }
</script>
```

Ten komponent definiuje właściwość `data` o nazwie `products`, do której jest przypisana tablica obiektów wyliczanych w szablonie za pomocą dyrektywy `v-for`. Każdy obiekt z tablicy `products` generuje wiersz w tabeli, zawierający kolumny `id`, `name` i `price`, a także przycisk (`button`), który pozwoli na edycję obiektu. Dodatkowy przycisk, przeznaczony do dodania nowego produktu, jest umieszczony pod tabelą. Dzięki dyrektywie `v-on` zastosowanej wobec obu rodzajów przycisków wywołamy metody `createNew` i `editProduct`. Obie metody są w tym momencie puste.

Tworzenie komponentu edytora produktu

Teraz muszę utworzyć edytor, aby użytkownik był w stanie tworzyć nowe obiekty i edytować już istniejące. Rozpocznę od dodania do katalogu `src/components` pliku `EditorField.vue` o treści jak w listingu 18.7.

Listing 18.7. Zawartość pliku `src/components/EditorField.vue`

```
<template>
  <div class="form-group">
    <label>{{label}}</label>
    <input v-model.number="value" class="form-control" />
  </div>
</template>
<script>
export default {
  props: ["label"],
  data: function () {
    return {
      value: ""
    }
  }
}
</script>
```

Ten komponent wyświetla elementy `label` i `input`. Aby włączyć edytor w skład większej całości, dodaję do katalogu `src/components` plik `ProductEditor.vue` o treści z listingu 18.8.

Listing 18.8. Zawartość pliku `src/components/ProductEditor.vue`

```
<template>
  <div>
    <editor-field label="ID" />
    <editor-field label="Nazwa" />
    <editor-field label="Cena" />
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="save">
        {{ editing ? "Zapisz" : "Utwórz" }}
      </button>
      <button class="btn btn-secondary" v-on:click="cancel">Anuluj</button>
    </div>
  </div>
</template>
<script>
  import EditorField from "./EditorField";
  export default {
    data: function () {
      return {
        editing: false,
        product: {
          id: 0,
```

```

        name: "",
        price: 0
    }
},
components: { EditorField },
methods: {
    startEdit(product) {
        this.editing = true;
        this.product = {
            id: product.id,
            name: product.name,
            price: product.price
        }
    },
    startCreate() {
        this.editing = false;
        this.product = {
            id: 0,
            name: "",
            price: 0
        };
    },
    save() {
        // TODO - przetwórz zmieniony lub utworzony produkt
        console.log(`Zakończono edycję: ${JSON.stringify(this.product)}`);
        this.startCreate();
    },
    cancel() {
        this.product = {};
        this.editing = false;
    }
}
</script>
```

Edytor wyświetli kilka elementów, które mogą być używane do edycji lub tworzenia obiektów. Obsługiwane pola to id, name i price wraz z przyciskami do zakończenia lub anulowania operacji.

Wyświetlanie komponentów-dzieci

Aby zakończyć przygotowania do tego rozdziału, zmieniam główny komponent tak, aby wyświetlał komponenty utworzone w poprzednich punktach (listing 18.9).

Listing 18.9. Wyświetlanie komponentów w pliku src/App.vue

```
<template>
<div class="container-fluid">
    <div class="row">
        <div class="col-8 m-3">
            <product-display></product-display>
        </div>
        <div class="col m-3">
            <product-editor></product-editor>
        </div>
    </div>
</div>
```

```
</template>
<script>
    import ProductDisplay from "./components/ProductDisplay";
    import ProductEditor from "./components/ProductEditor";
    export default {
        name: 'App',
        components: { ProductDisplay, ProductEditor }
    }
</script>
```

Komponent wyświetla dzieci obok siebie (rysunek 18.2). Kliknięcie przycisków nie daje żadnego efektu, ponieważ komponenty nie są jeszcze ze sobą powiązane.

The screenshot shows a web application titled "productapp" running on "localhost:8080". The main content is a table listing five products:

ID	Nazwa	Cena	
1	Kajak	275.00 PLN	<button>Edytuj</button>
2	Kamizelka ratunkowa	48.95 PLN	<button>Edytuj</button>
3	Piłka nożna	19.50 PLN	<button>Edytuj</button>
4	Chorągiewki narożne	39.95 PLN	<button>Edytuj</button>
5	Stadion	79500.00 PLN	<button>Edytuj</button>

Below the table are three input fields for adding a new product: "ID", "Nazwa", and "Cena". At the bottom are two buttons: "Utwórz nowy" (Create new) and "Utwórz" (Create).

Rysunek 18.2. Dodawanie funkcji i komponentów do przykładowej aplikacji

Omówienie wstrzykiwania zależności

Wstrzykiwanie zależności pozwala na zdefiniowanie usługi w komponencie. Usługą może być dowolna wartość, funkcja czy też obiekt, która w ten sposób staje się dostępna dla swoich potomków. Usługi dostarczone za pomocą wstrzykiwania zależności nie są ograniczone jedynie do dzieci — nie musimy przekazywać ich za pomocą łańcucha propów, aby mieć pewność, że dotrą we właściwe miejsce.

Tworzenie usługi

W listingu 18.10 do komponentu głównego dodaję usługę przekazującą informacje dotyczące klas Bootstrapa, które powinny być używane wobec tła czy tekstu. Ta usługa jest dostępna dla wszystkich komponentów w aplikacji, ponieważ wszystkie one są potomkami komponentu głównego.

Listing 18.10. Tworzenie usługi w pliku *src/App.vue*

```
<template>
    <div class="container-fluid">
        <div class="row">
            <div class="col-8 m-3">
```

```

<product-display></product-display>
</div>
<div class="col m-3">
    <product-editor></product-editor>
</div>
</div>
</template>
</script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
export default {
    name: 'App',
    components: { ProductDisplay, ProductEditor },
    provide: function() {
        return {
            colors: {
                bg: "bg-secondary",
                text: "text-white"
            }
        }
    }
}
</script>

```

Właściwość provide działa według takiego samego wzorca jak właściwość data — zwraca funkcję dostarczającą obiekt, którego właściwości stanowią nazwy usług dostępne dla komponentów potomków. W tym przykładzie definiuję usługę o nazwie colors, której właściwości tekstowe bg i text zawierają nazwy klas stylów Bootstrapa.

Konsumowanie usługi za pomocą wstrzykiwania zależności

Gdy komponent chce skonsumować usługę dostarczoną przez swojego przodka, musi skorzystać z właściwości inject, co pokazuje listing 18.11, w którym skonfigurowałem komponent EditorField tak, aby konsumował usługę colors zdefiniowaną w listingu 18.10.

Listing 18.11. Konsumowanie usługi w pliku src/components/EditorField.vue

```

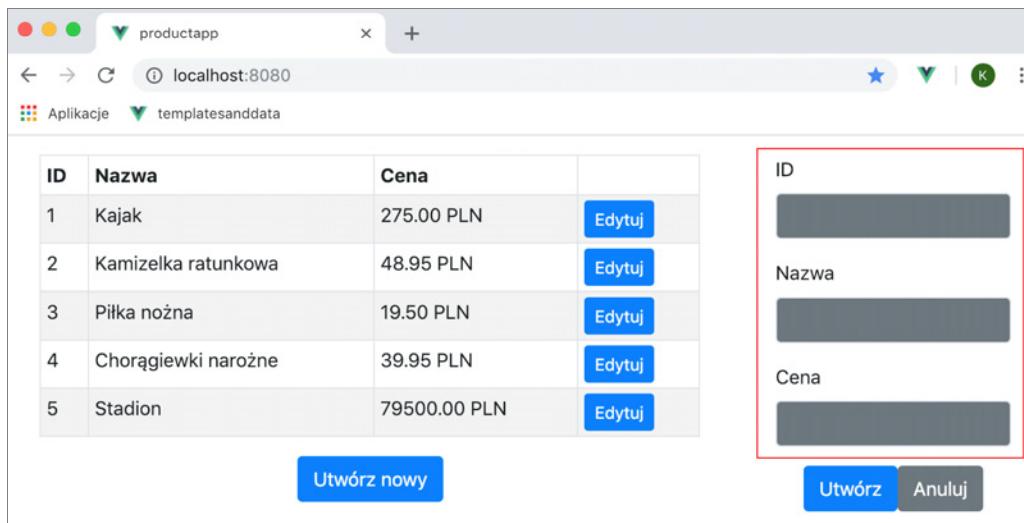
<template>
<div class="form-group">
    <label>{{label}}</label>
    <input v-model.number="value" class="form-control" v-bind:class="[colors.bg,
    ↳colors.text]" />
</div>
</template>
<script>
export default {
    props: ["label"],
    data: function () {
        return {
            value: ""
        }
    },
    inject: ["colors"]
}
</script>

```

Właściwość `inject` otrzymuje tablicę, która zawiera nazwy wszystkich usług wymaganych przez komponent. W momencie inicjalizacji komponentu Vue.js przetwarza wszystkich przodków komponentu, dopóki nie zostanie znaleziona usługa o danej nazwie. Następnie pobierana jest jej wartość, po czym dochodzi do jej przypisania do właściwości, będącej nazwą usługi, w obiekcie komponentu. W tym przykładzie Vue.js przetwarza komponent w poszukiwaniu usługi o nazwie `colors`, która znajduje się w komponencie głównym, po czym korzysta z obiektu dostarczonego przez komponent główny jako wartości komponentu o nazwie `colors` w komponencie `EditorField`. Utworzenie właściwości, która jest związana z usługą, pozwala na użycie właściwości w wyrażeniu dyrektywy:

```
...
<input v-model.number="value" class="form-control" v-bind:class="[colors.bg, colors.text]" />
...
```

Dzięki takiej konstrukcji komponent główny dostarcza nazwy klas, które powinny być zastosowane do stylowania elementów, bez konieczności przekazywania ich po kolejno przez wszystkie komponenty w łańcuchu. Nazwy klas są stosowane wobec elementów `input` komponentu `EditorField`, co daje efekt jak na rysunku 18.3 (musisz wprowadzić jakiś tekst do elementów `input`, aby zobaczyć kolor tekstu).



Rysunek 18.3. Zastosowanie wstrzykiwania zależności

■ **Wskazówka** Aby zobaczyć efekt wprowadzonych zmian, może zaistnieć konieczność odświeżenia przeglądarki.

Przesłanianie usług pochodzących od przodków

Gdy Vue.js tworzy komponent z właściwością `inject`, jego zadaniem jest znalezienie zależności od wymaganych usług, do czego niezbędne jest przejrzenie łańcucha komponentów i sprawdzenie, czy w każdym z nich nie ma przypadkiem właściwości `provide` z usługą o pasującej nazwie. W związku z tym komponent może prześlonić jedną lub więcej usług dostarczanych przez przodków, co może być użyteczną metodą tworzenia wyspecjalizowanych usług, które dotyczą tylko fragmentów aplikacji. W ramach przykładu dodaj właściwość `provide` do komponentu `ProductEditor`, który definiuje usługę `colors` (listing 18.12).

Listing 18.12. Definicja usługi w pliku *src/components/ProductEditor.vue*

```
...
<script>
  import EditorField from "./EditorField";
  export default {
    data: function () {
      return {
        editing: false,
        product: {
          id: 0,
          name: "",
          price: 0
        }
      }
    },
    components: { EditorField },
    methods: {
      //pominięto metody
    },
    provide: function () {
      return {
        colors: {
          bg: "bg-light",
          text: "text-danger"
        }
      }
    }
  }
</script>
...
```

Vue.js tworzy komponent `EditorField` i szuka usługi określonej we właściwości `inject`. Usługa ta zostanie znaleziona w komponencie `ProductEditor`, a dalsze przeszukiwanie zostanie wstrzymane. Oznacza to, że usługa `colors` zostanie znaleziona za pomocą nazw klas z listingu 18.12 (rysunek 18.4).

The screenshot shows a web application interface. At the top, there's a header bar with a title 'productapp' and a URL 'localhost:8080'. Below the header is a navigation bar with icons for applications and templates. The main content area contains a table of products:

ID	Nazwa	Cena	
1	Kajak	275.00 PLN	<button>Edytuj</button>
2	Kamizelka ratunkowa	48.95 PLN	<button>Edytuj</button>
3	Piłka nożna	19.50 PLN	<button>Edytuj</button>
4	Chorągiewki narożne	39.95 PLN	<button>Edytuj</button>
5	Stadion	79500.00 PLN	<button>Edytuj</button>

At the bottom left of the table is a blue button labeled 'Utwórz nowy'. On the right side of the table, there's a modal dialog with a red border. It contains four input fields:

- ID: An empty text input field.
- Nazwa: A text input field containing 'Kajak', which is highlighted with a blue border.
- Cena: An empty text input field.
- Buttons at the bottom right: 'Utwórz' (Create) and 'Anuluj' (Cancel).

Rysunek 18.4. Efekt przesłonięcia usługi

- **Wskazówka** Każda usługa jest rozwiązywana niezależnie od innych, co oznacza, że komponent może przesłonić tylko wybrane usługi dostarczone przez przodków.

Anonimowy charakter usług

Wstrzykiwanie zależności pozwala na tworzenie lużno powiązanych aplikacji, ponieważ usługi są dostarczane i konsumowane anonimowo. Komponent definiując usługę, nie musi wiedzieć, kto będzie ją konsumował, czy będzie ona przesłaniana przez inny komponent, a także czy w ogóle zostanie użyta.

Analogicznie komponent korzystający z usługi „nie wie”, który z przodków ją dostarczył. To podejście pozwala uniknąć wymuszania ścisłych związków pomiędzy komponentami i przekazywania danych przez kaskadę relacji rodzin – dziecko.

Tworzenie reaktywnych usług

Usługi Vue.js nie są domyślnie reaktywne, co oznacza, że wszelkie zmiany wprowadzone we właściwościach obiektu dostępnego w ramach usługi colors nie będą propagowane do reszty aplikacji. Z drugiej strony skoro Vue.js znajduje zależności usług przy każdym utworzeniu komponentu, za każdym razem będzie otrzymywał nowe wartości.

Jeśli chcesz utworzyć usługę, która propaguje zmiany, musisz przypisać obiekt do właściwości data i skorzystać z tej wartości w usłudze (listing 18.13).

Listing 18.13. Tworzenie reaktywnej usługi w pliku src/App.vue

```
<template>
  <div class="container-fluid">
    <div class="text-right m-2">
      <button class="btn btn-primary" v-on:click="toggleColors">
        Przełącz kolory
      </button>
    </div>
    <div class="row">
      <div class="col-8 m-3">
        <product-display></product-display>
      </div>
      <div class="col m-3">
        <product-editor></product-editor>
      </div>
    </div>
  </div>
</template>
<script>
  import ProductDisplay from "./components/ProductDisplay";
  import ProductEditor from "./components/ProductEditor";
  export default {
    name: 'App',
    components: { ProductDisplay, ProductEditor },
    data: function () {
      return {
        reactiveColors: {
          bg: "bg-secondary",
          text: "text-white"
        }
      }
    }
  }
</script>
```

```

        }
    },
    provide: function() {
        return {
            colors: this.reactiveColors
        }
    },
    methods: {
        toggleColors() {
            if (this.reactiveColors.bg == "bg-secondary") {
                this.reactiveColors.bg = "bg-light";
                this.reactiveColors.text = "text-danger";
            } else {
                this.reactiveColors.bg = "bg-secondary";
                this.reactiveColors.text = "text-white";
            }
        }
    }
}
</script>

```

Komponent definiuje właściwość danych o nazwie reactiveColors, która otrzymała obiekt o właściwościach bg i text. Podczas fazy tworzenia komponentu Vue.js przetwarza właściwość data przed właściwością provide, co oznacza, że obiekt reactiveColors staje się reaktywny i zostaje użyty jako wartość usługi colors. Aby zademonstrować usługę reaktywną, dodaję przycisk z dyrektywą v-on, która wywołuje metodę toggleColors zmieniającą wartości bg i text.

Aby ten przykład zadziałał, musiałem ukryć poprzez zastosowanie komentarza właściwość provide z komponentu ProductEditor (jak w listingu 18.14). W przeciwnym razie usługa w nim zadeklarowana przesłoniłaby tę zadeklarowaną w listingu 18.13.

Listing 18.14. Usuwanie usługi z pliku src/ProductEditor.vue

```

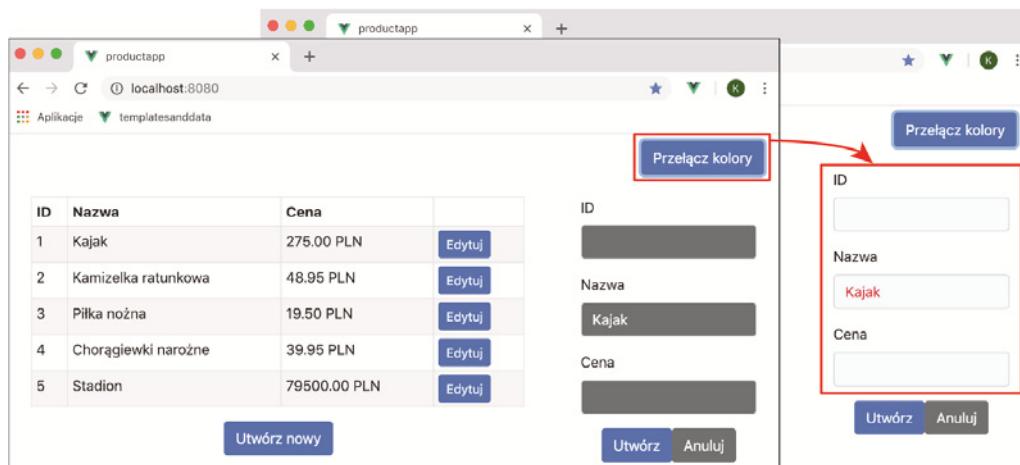
...
<script>
    import EditorField from "./EditorField";
    export default {
        data: function () {
            return {
                editing: false,
                product: {
                    id: 0,
                    name: "",
                    price: 0
                }
            }
        },
        components: { EditorField },
        methods: {
            // ...metody pominięto...
        },
        // provide: function () {
        //     return {
        //         colors: {
        //             bg: "bg-light",
        //             text: "text-danger"
        //         }
        // }

```

```
//      }
//  }
}
</script>
...
```

- **Wskazówka** Aby zobaczyć zmiany, może być konieczne odświeżenie okna przeglądarki.

W momencie kliknięcia przycisku wartości właściwości obiektu usługi są zmieniane i propagowane w aplikacji (rysunek 18.5).



Rysunek 18.5. Tworzenie reaktywnej usługi

- **Ostrzeżenie** W usłudze reaktywnej zmiany może wprowadzać dowolny komponent — nie tylko ten, który ją definiuje. Jest to niezwykle użyteczna funkcja, ale może również powodować nieoczekiwane zachowania.

Zaawansowane wstrzykiwanie zależności

Konsumowanie usług przydaje się w dwóch sytuacjach. Po pierwsze, można skorzystać z usługi w celu dostarczenia domyślnej wartości, która zostanie użyta, jeśli żaden przodek nie definiuje usługi, której komponent wymaga. Po drugie, możemy zmienić nazwę, pod którą usługa będzie używana w komponencie. Oba mechanizmy zastosowałem w listingu 18.15.

Listing 18.15. Zastosowanie zaawansowanych funkcji w pliku src/components/EditorField.vue

```
<template>
  <div class="form-group">
    <label>{{ formattedLabel }}</label>
    <input v-model.number="value" class="form-control"
           v-bind:class="[colors.bg, colors.text]" />
  </div>
</template>
<script>
  export default {
```

```

    props: ["label"],
    data: function () {
        return {
            value: "",
            formattedLabel: this.format(this.label)
        }
    },
    inject: {
        colors: "colors",
        format: {
            from: "labelFormatter",
            default: () => (value) => `Domyślnie ${value}`
        }
    }
}
</script>

```

Te funkcje oczekują, że wartość we właściwości `inject` będzie obiektem. Nazwa każdej właściwości jest nazwą, pod którą usługa będzie znana w ramach komponentu. Jeśli nie są wymagane zaawansowane funkcje, nazwą wymaganej usługi będzie wartość właściwości:

```

...
"colors": "colors",
...

```

Ta właściwość informuje Vue.js, że komponent wymaga usługi o nazwie `colors` i będzie odwoływać się do niej za pomocą tej nazwy, co da efekt identyczny jak we wcześniejszych przykładach. Druga właściwość korzysta z obu mechanizmów.

```

...
inject: {
    colors: "colors",
    format: {
        from: "labelFormatter",
        default: () => (value) => `Domyślnie ${value}`
    }
}
...

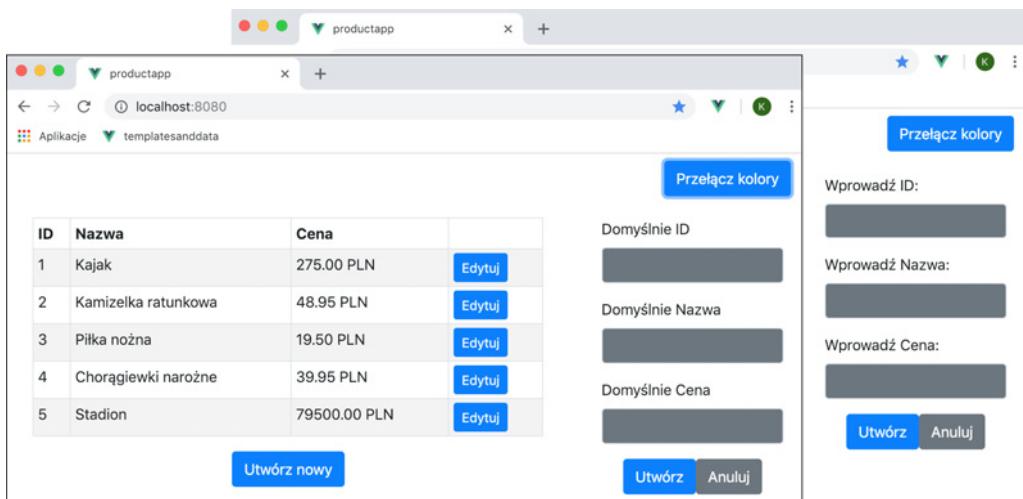
```

Właściwość `from` informuje Vue.js, że należy znaleźć usługę o nazwie `labelFormatter`, dostępną u jednego z przodków komponentu. Ta usługa będzie jednak znana pod nazwą zgodną z ustawieniami we właściwości `format`. Właściwość `default` dostarcza domyślną wartość, która zostanie użyta, jeśli żaden z przodków komponentu nie dostarcza usługi `labelFormatter`.

- **Uwaga** Funkcja fabryki jest wymagana, gdy dostarczasz wartości domyślne dla usług. Z tego względu właściwość `default` w listingu 18.15 otrzymuje funkcję, która zwraca w wyniku inną funkcję. Po utworzeniu komponentu Vue.js wywoła funkcję, aby pobrać obiekt, który będzie używany jako usługa.

Wartość właściwości `default` to funkcja, która poprzedza wartość słowem *Domyślnie*. Dzięki temu bez problemu będziemy wiedzieć, kiedy zostanie użyta domyślna usługa (rysunek 18.6).

Domyślna wartość usługi została użyta, ponieważ żaden z przodków komponentu nie dostarczył usługi o nazwie `labelFormatter`. Aby zademonstrować, że dostarczona usługa zostanie użyta, o ile tylko będzie dostępna, tworzę usługę o podanej nazwie (listing 18.16).



Rysunek 18.6. Zastosowanie domyślnej wartości dla usługi

Listing 18.16. Tworzenie usługi (src/App.vue)

```
...
<script>
    import ProductDisplay from "./components/ProductDisplay";
    import ProductEditor from "./components/ProductEditor";
    export default {
        name: 'App',
        components: { ProductDisplay, ProductEditor },
        data: function () {
            return {
                reactiveColors: {
                    bg: "bg-secondary",
                    text: "text-white"
                }
            },
            provide: function() {
                return {
                    colors: this.reactiveColors,
                    labelFormatter: (value) => `Wprowadź ${value}:`
                }
            },
            methods: {
                toggleColors() {
                    if (this.reactiveColors.bg == "bg-secondary") {
                        this.reactiveColors.bg = "bg-light";
                        this.reactiveColors.text = "text-danger";
                    } else {
                        this.reactiveColors.bg = "bg-secondary";
                        this.reactiveColors.text = "text-white";
                    }
                }
            }
        }
    </script>
    ...

```

Nowa usługa to funkcja, która przekształca przekazaną wartość, poprzedzając ją słowem *Wprowadź*, co daje efekt jak w prawej części rysunku 18.6. Skoro nie jest to domyślna wartość usługi, nie muszę definiować funkcji fabryki.

Stosowanie szyny zdarzeń

Wstrzykiwanie zależności można połączyć z inną zaawansowaną funkcją Vue.js, która pozwala na wysyłanie i odbieranie zdarzeń poza zakres relacji rodzic – dziecko. Mowa o szynie zdarzeń (ang. *event bus*). Pierwszym krokiem w tworzeniu szyny zdarzeń jest zdefiniowanie usługi w obiekcie Vue (listing 18.17).

Listing 18.17. Tworzenie usługi szyny zdarzeń w pliku *src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
  render: h => h(App),
  provide: function () {
    return {
      eventBus: new Vue()
    }
  }
}).$mount('#app')
```

Wartością usługi jest nowy obiekt Vue. Takie zachowanie może wydawać się dziwne, ale w ten sposób otrzymujemy obiekt, który może wysyłać i odbierać własne zdarzenia z poziomu kodu bez konieczności stosowania hierarchii komponentów aplikacji.

Wysyłanie zdarzeń za pomocą szyny zdarzeń

Metoda `$emit` jest używana do wysyłania zdarzeń za pomocą szyny zdarzeń. Odbywa się to według tej samej reguły, którą pokazałem w rozdziale 16. przy okazji wysyłania zdarzeń do komponentu-rodzica. W listingu 18.18 aktualizuję komponent *ProductDisplay*, dzięki czemu skorzysta on z szyny zdarzeń, aby wysłać własne zdarzenia, gdy użytkownik kliknie przycisk *Utwórz nowy* lub *Edytuj*.

Listing 18.18. Zastosowanie szyny zdarzeń w pliku *src/components/ProductDisplay.vue*

```
...
<script>
  export default {
    data: function () {
      return {
        products: [
          { id: 1, name: "Kajak", price: 275 },
          { id: 2, name: "Kamizelka ratunkowa", price: 48.95 },
          { id: 3, name: "Piłka nożna", price: 19.50 },
          { id: 4, name: "Chorągiewki narożne", price: 39.95 },
          { id: 5, name: "Stadion", price: 79500 }
        ],
        filters: {
          currency(value) {
            return `${value.toFixed(2)} PLN`;
          }
        }
      }
    }
  }
}
```

```

        },
        methods: {
            createNew() {
                this.eventBus.$emit("create");
            },
            editProduct(product) {
                this.eventBus.$emit("edit", product);
            }
        },
        inject: ["eventBus"]
    }
</script>
...

```

Właściwość `inject` jest używana do tworzenia zależności od usługi `eventBus`. Jej metoda `$emit` jest używana do wysyłania własnych zdarzeń `create` i `edit` w metodach, które są wywoływane, gdy użytkownik kliknie przyciski button zaprezentowane za pomocą komponentu.

■ **Uwaga** Jedną z wad modelu szyny zdarzeń jest konieczność zapewnienia unikatowych nazw zdarzeń w całej aplikacji, dzięki czemu zdarzenia nie będą ze sobą mylone. Jeśli aplikacja stanie się zbyt duża, aby umożliwić łatwe zarządzanie nazwami zdarzeń, powinieneś rozważyć zastosowanie stanu współdzielonego (por. rozdział 20.).

Odbieranie zdarzeń z szyny zdarzeń

Funkcją, która pozwala szynie zdarzeń na zaistnienie w Vue.js, jest możliwość rejestrowania się na zdarzenia z poziomu kodu dzięki metodzie `$on`. Można ją wykonywać po zakończeniu inicjalizacji komponentu i znalezieniu zależności od usług. W listingu 18.19 stosuję szynę zdarzeń w komponencie `ProductEditor`, aby otrzymać zdarzenia wysłane w poprzednim punkcie.

Listing 18.19. Odbieranie zdarzeń w pliku src/components/ProductEditor.vue

```

...
<script>
    import EditorField from "./EditorField";
    export default {
        data: function () {
            return {
                editing: false,
                product: {
                    id: 0,
                    name: "",
                    price: 0
                }
            }
        },
        components: { EditorField },
        methods: {
            startEdit(product) {
                this.editing = true;
                this.product = {
                    id: product.id,
                    name: product.name,
                    price: product.price
                }
            },

```

```

        startCreate() {
            this.editing = false;
            this.product = {
                id: 0,
                name: "",
                price: 0
            };
        },
        save() {
            this.eventBus.$emit("complete", this.product);
            this.startCreate();
            console.log(`Zakończono edycję: ${JSON.stringify(this.product)}`);
        },
        cancel() {
            this.product = {};
            this.editing = false;
        }
    },
    inject: ["eventBus"],
    created() {
        this.eventBus.$on("create", this.startCreate);
        this.eventBus.$on("edit", this.startEdit);
    }
}
</script>
...

```

Właściwość `inject` jest używana do zadeklarowania zależności od usługi `eventBus`. Metoda `created` rejestruje metody obsługi dla zdarzeń `create` i `edit`, które są używane do wywołania metod `startCreate` i `startEdit` zdefiniowanych na początku rozdziału.

Komponenty mogą wysyłać i odbierać zdarzenia za pomocą szyny zdarzeń. W listingu 18.19 korzystam z metody `$emit` do wysłania zdarzenia `complete` w momencie wywołania metody `save`. Ta dwukierunkowa komunikacja pozwala na łatwe konstruowanie złożonych zachowań. W listingu 18.20 modyfikuję komponent `ProductDisplay`, aby ten aktualizował wyświetlane dane w odpowiedzi na zdarzenie `complete`. Zdarzenie to informuje aplikację o zakończeniu edycji istniejącego produktu lub utworzeniu nowego.

Listing 18.20. Odbieranie zdarzeń w pliku `src/components/ProductDisplay.vue`

```

...
<script>
    import Vue from "vue";
    export default {
        data: function () {
            return {
                products: [
                    { id: 1, name: "Kajak", price: 275 },
                    { id: 2, name: "Kamizelka ratunkowa", price: 48.95 },
                    { id: 3, name: "Piłka nożna", price: 19.50 },
                    { id: 4, name: "Chorągiewki narożne", price: 39.95 },
                    { id: 5, name: "Stadion", price: 79500 }
                ]
            },
            filters: {
                currency(value) {
                    return `${value.toFixed(2)} PLN`;
                }
            },
            methods: {

```

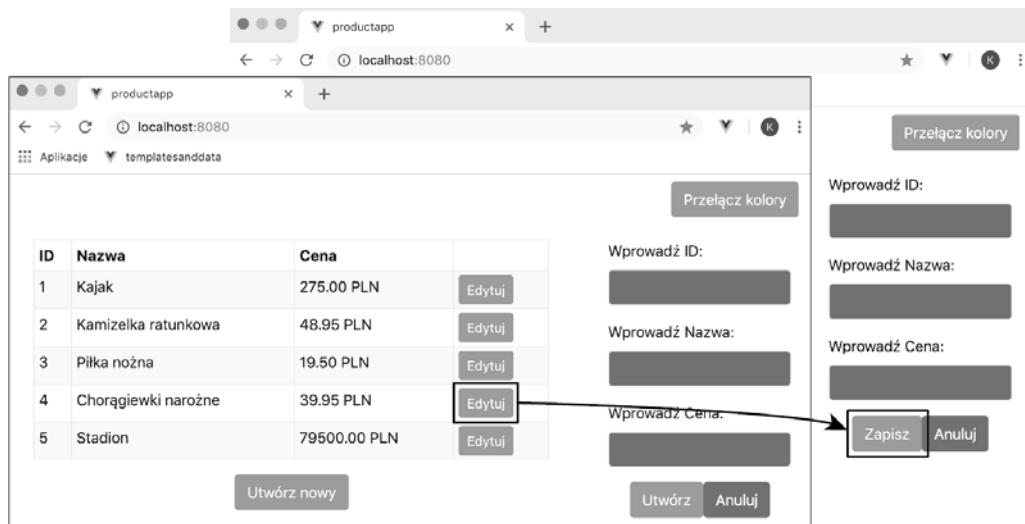
```

        createNew() {
            this.eventBus.$emit("create");
        },
        editProduct(product) {
            this.eventBus.$emit("edit", product);
        },
        processComplete(product) {
            let index = this.products.findIndex(p => p.id == product.id);
            if (index == -1) {
                this.products.push(product);
            } else {
                Vue.set(this.products, index, product);
            }
        }
    },
    inject: ["eventBus"],
    created() {
        this.eventBus.$on("complete", this.processComplete);
    }
}
</script>
...

```

Metoda created jest używana do nasłuchiwanego zdarzenia complete i w momencie jego otrzymania — wywołania metody processComplete. Metoda processComplete korzysta z właściwości id jako klucza, aby zaktualizować obiekty, które zostały zmienione, lub aby dodać nowe obiekty.

Teraz komponenty ProductDisplay i ProductEditor mogą wysyłać i odbierać zdarzenia, dzięki którym mogą reagować na swoje wzajemne działania, wykraczające poza relację rodzic – dziecko. Wciąż brakuje jeszcze trochę funkcjonalności, ale jeśli klikniesz przycisk *Edytuj* lub *Utwórz nowy*, przekonasz się, że tekst na przyciskach edytora zostanie w odpowiedni sposób zmieniony (rysunek 18.7).



Rysunek 18.7. Zastosowanie szyny zdarzeń

Tworzenie lokalnych szyn zdarzeń

Nie musisz korzystać z pojedynczej szyny zdarzeń dla całej aplikacji. Pojedyncze części Twojej aplikacji mogą mieć własne szyny i przesyłać w ich ramach własne zdarzenia bez konieczności dystrybuowania ich do kompletnie niezwiązanych komponentów. W listingu 18.21 do komponentu ProductEditor dodaję odrębną szynę zdarzeń o nazwie editingEventBus. Wykorzystuję ją do wysyłania i odbierania własnych zdarzeń, które powiążą pojedyncze pola edytora z resztą aplikacji.

Listing 18.21. Tworzenie lokalnej szyny zdarzeń w pliku src/components/ProductEditor.vue

```
<template>
  <div>
    <editor-field label="ID" editorFor="id" />
    <editor-field label="Nazwa" editorFor="name" />
    <editor-field label="Cena" editorFor="price" />
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="save">
        {{ editing ? "Zapisz" : "Utwórz" }}
      </button>
      <button class="btn btn-secondary" v-on:click="cancel">Anuluj</button>
    </div>
  </div>
</template>
<script>
  import EditorField from "./EditorField";
  import Vue from "vue";
  export default {
    data: function () {
      return {
        editing: false,
        product: {
          id: 0,
          name: "",
          price: 0
        },
        localBus: new Vue()
      }
    },
    components: { EditorField },
    methods: {
      startEdit(product) {
        this.editing = true;
        this.product = {
          id: product.id,
          name: product.name,
          price: product.price
        }
      },
      startCreate() {
        this.editing = false;
        this.product = {
          id: 0,
          name: "",
          price: 0
        };
      },
      save() {
        // ...
      }
    }
  }
</script>
```

```

        this.eventBus.$emit("complete", this.product);
        this.startCreate();
        console.log(`Zakończono edycję: ${JSON.stringify(this.product)}`);
    },
    cancel() {
        this.product = {};
        this.editing = false;
    }
},
inject: ["eventBus"],
provide: function () {
    return {
        editingEventBus: this.localBus
    }
},
created() {
    this.eventBus.$on("create", this.startCreate);
    this.eventBus.$on("edit", this.startEdit);
    this.localBus.$on("change",
        (change) => this.product[change.name] = change.value);
},
watch: {
    product(newValue, oldValue) {
        this.localBus.$emit("target", newValue);
    }
}
}
</script>
```

W tym listingu dokonaliśmy wielu zmian, ale wszystkie one mają na celu dostarczenie lokalnej szyny zdarzeń przeznaczonej wyłącznie do obsługi zdarzeń związanych z edycją obiektu produktu, co łączy mechanizmy poznane w tym rozdziale z tymi poznanymi w poprzednich rozdziałach. Gdy użytkownik rozpoczyna edycję lub tworzenie nowego obiektu, zdarzenie target jest wysyłane do lokalnej szyny zdarzeń. Po otrzymaniu w tej szynie zdarzenia change aktualizujemy obiekt produktu, odzwierciedlając zmianę dokonaną przez użytkownika. Uzupełniające zmiany są niezbędne w celu dostosowania komponentu odpowiedzialnego za wyświetlanie pól edytora (listing 18.22).

Listing 18.22. Zastosowanie lokalnej szyny zdarzeń w pliku *src/components/EditorField.vue*

```

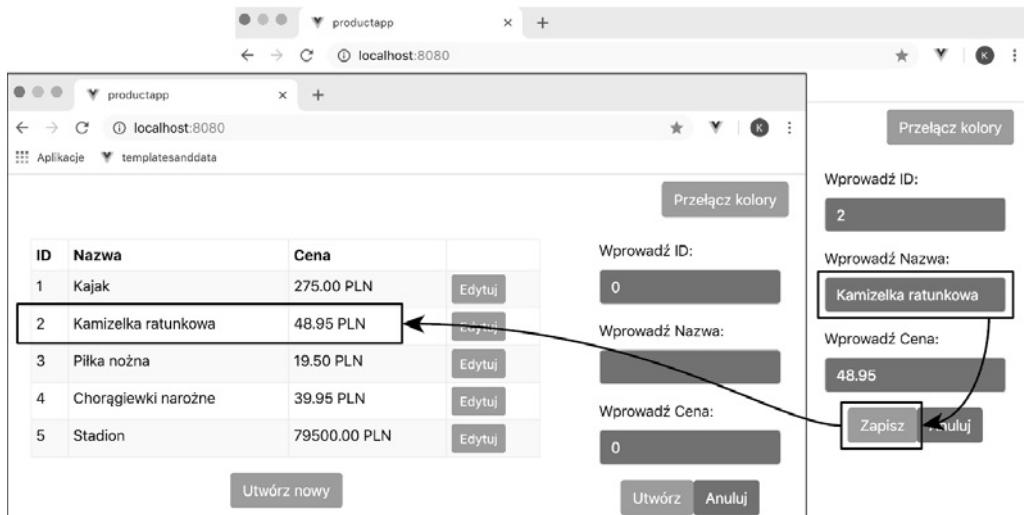
<template>
    <div class="form-group">
        <label>{{ formattedLabel }}</label>
        <input v-model.number="value" class="form-control" v-bind:class="[colors.bg,
            & colors.text]" />
    </div>
</template>
<script>
export default {
    props: ["label", "editorFor"],
    data: function () {
        return {
            value: "",
            formattedLabel: this.format(this.label)
        }
    },
    inject: {
        colors: "colors",
        format: {
```

```

        from: "labelFormatter",
        default: () => (value) => `Domyślnie ${value}`
    },
    editingEventBus: "editingEventBus"
},
watch: {
    value(newValue) {
        this.editingEventBus.$emit("change",
            { name: this.editorFor, value: this.value});
    }
},
created() {
    this.editingEventBus.$on("target",
        (p) => this.value = p[this.editorFor]);
}
}
</script>

```

Po wprowadzonych modyfikacjach kliknięcie przycisku *Edytuj* pozwoli na zmianę istniejącego obiektu, a kliknięcie przycisku *Utwórz nowy* przeniesie użytkownika do procesu tworzenia nowego obiektu. Kliknięcie przycisku *Utwórz* lub *Zapisz* spowoduje zapisanie zmian we właściwy sposób (rysunek 18.8).



Rysunek 18.8. Powiązanie komponentów edytora za pomocą lokalnej szyny zdarzeń

Podsumowanie

W tym rozdziale objaśniłem, jak skorzystać z funkcji wstrzykiwania zależności i szyny zdarzeń, aby wyjść poza ścisły schemat komunikacji rodzic – dziecko, a także jak tworzyć komponenty luźno powiązane. Pokazałem, jak definiować i konsumować usługi i czynić je reaktywnymi, a także jak utworzyć szynę zdarzeń w celu rozpowszechnienia własnych zdarzeń w aplikacji. W kolejnym rozdziale wyjaśniam, jak korzystać z REST-owych usług sieciowych w aplikacjach Vue.js.

ROZDZIAŁ 19.

Stosowanie REST-owych usług sieciowych

W tym rozdziale pokazuję, jak skonsumować REST-ową usługę sieciową w aplikacji Vue.js. Objaśniam różne sposoby wykonywania żądań HTTP. Rozszerzę także przykładową aplikację, dzięki czemu będzie ona w stanie przechowywać dane i odczytywać je z serwera. Tabela 19.1 umiejscawia usługi sieciowe w szerszym kontekście.

Tabela 19.1. Umiejscowienie REST-owych usług sieciowych w szerszym kontekście

Pytanie	Odpowiedź
Czym są REST-owe usługi sieciowe?	REST-owe usługi sieciowe dostarczają danych do aplikacji webowych za pomocą żądań HTTP.
Dlaczego są użyteczne?	Wiele aplikacji musi przechowywać dane i zarządzać nimi, korzystając z trwałego magazynu danych.
Jak się z nich korzysta?	Aplikacja webowa wysyła żądanie HTTP do serwera, korzystając z podanego rodzaju żądania i adresu URL, aby zidentyfikować, jakiego rodzaju dane są potrzebne i jaką czynność chcemy na nich wykonać.
Czy są jakieś pułapki lub ograniczenia?	REST-owe usługi sieciowe nie mają ściśle określonego standardu działania. W aplikacji Vue.js żądania HTTP są wykonywane asynchronicznie, co jest mylące dla wielu programistów. Trzeba niezwykle uważać na kwestię obsługi błędów, ponieważ Vue.js nie wykryje ich automatycznie.
Czy są jakieś rozwiązania alternatywne?	Nie musisz korzystać z żądań HTTP w aplikacji webowej, zwłaszcza jeśli pracujesz na niewielkiej ilości danych.

Tabela 19.2 podsumowuje rozdział.

Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę z projektem *productapp* z rozdziału 18. Wykonaj instrukcje z poniższych punktów, aby przygotować przykładową aplikację do pracy z żdaniami HTTP.

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.

Tabela 19.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Pobierz dane z usługi sieciowej.	Utwórz metodę Axios.get i odczytaj właściwość data obiektu odpowiedzi.	19.10 – 19.14
Skonsoliduj kod odpowiedzialny za dostęp do usługi sieciowej.	Utwórz usługę HTTP.	19.15 – 19.17
Wykonaj inne operacje protokołu HTTP.	Skorzystaj z metody Axios, która odpowiada pożądaniemu przez Ciebie typowi żądania.	19.18 – 19.19
Zareaguj na błędy.	Skonsoliduj żądania i skorzystaj z bloku try/catch, aby przechwycić i obsłużyć błąd.	19.20 – 19.23

Przygotowanie serwera HTTP

Aby obsługiwać żądania HTTP wysyłane przez naszą aplikację, musimy zainstalować dodatkowy pakiet. Wykonaj polecenie z listingu 19.1 w katalogu *productapp*, aby zainstalować pakiet *json-server*.

Listing 19.1. Instalacja pakietu

```
npm install json-server@0.12.1
```

Aby dostarczyć do serwera dane, które zostaną użyte podczas obsługi żądań HTTP, do katalogu *productapp* dodaj plik *restData.js* o treści z listingu 19.2.

Listing 19.2. Zawartość pliku *productapp/restData.js*

```
module.exports = function () {
  var data = {
    products: [
      { id: 1, name: "Kajak", category: "Sporty wodne", price: 275 },
      { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne", price: 48.95 },
      { id: 3, name: "Piłka nożna", category: "Piłka nożna", price: 19.50 },
      { id: 4, name: "Choragięwki narożne", category: "Piłka nożna", price: 34.95 },
      { id: 5, name: "Stadion", category: "Piłka nożna", price: 79500 },
      { id: 6, name: "Myśląca czapeczka", category: "Szachy", price: 16 },
      { id: 7, name: "Chwiejne krzesło", category: "Szachy", price: 29.95 },
      { id: 8, name: "Szachownica", category: "Szachy", price: 75 },
      { id: 9, name: "Król(u) złoty", category: "Szachy", price: 1200 }
    ]
  }
  return data
}
```

Aby NPM mógł uruchomić pakiet *json-server*, dodaj instrukcję z listingu 19.3 do sekcji *scripts* pliku *package.json*.

Listing 19.3. Dodawanie skryptu do pliku *productapp/package.json*

```
...
  "scripts": {
    "serve": "vue-cli-service serve", "build": "vue-cli-service build",
    "lint": "vue-cli-service lint",
    "json": "json-server restData.js -p 3500"
  },
  ...

```

Przygotowanie przykładowej aplikacji

Aby przygotować przykładową aplikację, muszę usunąć funkcje, które nie są nam już potrzebne. Usunę także zaszyte na sztywno dane, a także dodam obsługę kategorii do już istniejących informacji o identyfikatorze, nazwie i cenie (por. rozdział 18.).

Instalacja pakietu HTTP

Zacznę jednak od dodania pakietu, który pozwoli na wykonywanie żądań HTTP. Wykonaj polecenie z listingu 19.4 w katalogu *productapp*, aby zainstalować pakiet Axios.

Listing 19.4. Instalacja pakietu

```
npm install axios@0.18.0
```

Axios to popularna biblioteka do wykonywania żądań HTTP w aplikacjach webowych. Nie jest ona przeznaczona wyłącznie do Vue.js, ale z biegiem czasu — z uwagi na jej łatwość użycia i stabilność — stała się jedną z najpopularniejszych. Nie musisz korzystać z biblioteki Axios we własnych projektach — inne możliwości opisuję w ramce „Wybór mechanizmu obsługi żądań HTTP”.

Upraszczanie komponentów

Komponent *ProductEditor* został uproszczony, przez co komponenty *input* są wyświetlane bezpośrednio, a nie za pomocą komponentu pośredniego (w rozdziale 18. zastosowałem takie rozwiązanie, aby pokazać, jak łączyć ze sobą różne części aplikacji). Listing 19.5 przedstawia uproszczony komponent.

Listing 19.5. Uproszczenie komponentu w pliku src/components/ProductEditor.vue

```
<template>
  <div>
    <div class="form-group">
      <label>ID</label>
      <input class="form-control" v-model="product.id" />
    </div>
    <div class="form-group">
      <label>Nazwa</label>
      <input class="form-control" v-model="product.name" />
    </div>
    <div class="form-group">
      <label>Kategoria</label>
      <input class="form-control" v-model="product.category" />
    </div>
    <div class="form-group">
      <label>Cena</label>
      <input class="form-control" v-model.number="product.price" />
    </div>
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="save">
        {{ editing ? "Zapisz" : "Utwórz" }}
      </button>
      <button class="btn btn-secondary" v-on:click="cancel">Anuluj</button>
    </div>
  </div>
</template>
<script>
  export default {
    data: function () {
```

```

        return {
            editing: false,
            product: {}
        }
    },
    methods: {
        startEdit(product) {
            this.editing = true;
            this.product = {
                id: product.id,
                name: product.name,
                category: product.category,
                price: product.price
            }
        },
        startCreate() {
            this.editing = false;
            this.product = {};
        },
        save() {
            this.eventBus.$emit("complete", this.product);
            this.startCreate();
        },
        cancel() {
            this.product = {};
            this.editing = false;
        }
    },
    inject: ["eventBus"],
    created() {
        this.eventBus.$on("create", this.startCreate);
        this.eventBus.$on("edit", this.startEdit);
    }
}
</script>

```

Następnie upraszczam komponent ProductDisplay, aby usunąć dane zaszyte na sztywno i dodać obsługę wyświetlania wartości kategorii (listing 19.6).

Listing 19.6. Uproszczenie komponentu w pliku *src/components/ProductDisplay.vue*

```

<template>
<div>
    <table class="table table-sm table-striped table-bordered">
        <tr>
            <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
        </tr>
        <tbody>
            <tr v-for="p in products" v-bind:key="p.id">
                <td>{{ p.id }}</td>
                <td>{{ p.name }}</td>
                <td>{{ p.category }}</td>
                <td>{{ p.price }}</td>
                <td>
                    <button class="btn btn-sm btn-primary"
                           v-on:click="editProduct(p)">
                        Edytuj
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
</div>

```

```

        </tr>
        <tr v-if="products.length == 0">
            <td colspan="5" class="text-center">Brak danych</td>
        </tr>
    </tbody>
</table>
<div class="text-center">
    <button class="btn btn-primary" v-on:click="createNew">
        Utwórz nowy
    </button>
</div>
</div>
</template>
<script>
    import Vue from "vue";
    export default {
        data: function () {
            return {
                products: []
            }
        },
        methods: {
            createNew() {
                this.eventBus.$emit("create");
            },
            editProduct(product) {
                this.eventBus.$emit("edit", product);
            }
        },
        inject: ["eventBus"]
    }
</script>

```

Na zakończenie upraszczam komponent główny — App — usuwając przycisk do przełączania koloru, a także powiązane metodę i usługę (listing 19.7).

Listing 19.7. Uproszczenie komponentu w pliku src/App.vue

```

<template>
    <div class="container-fluid">
        <div class="row">
            <div class="col-8 m-3"><product-display/></div>
            <div class="col m-3"><product-editor/></div>
        </div>
    </div>
</template>
<script>
    import ProductDisplay from "./components/ProductDisplay";
    import ProductEditor from "./components/ProductEditor";
    export default {
        name: 'App',
        components: { ProductDisplay, ProductEditor }
    }
</script>

```

Uruchamianie przykładowej aplikacji i serwera HTTP

W tym rozdziale są wymagane dwie instancje wiersza poleceń (terminala): jedna do uruchomienia serwera HTTP, a druga do uruchomienia narzędzi deweloperskich Vue.js. Otwórz nowe okno wiersza poleceń, przejdź do katalogu *productapp* i wykonaj polecenie z listingu 19.8, aby uruchomić serwer HTTP.

Listing 19.8. Uruchamianie REST-owego serwera

```
npm run json
```

Serwer rozpoczęte nasłuchiwanie żądań na porcie 3500. Aby sprawdzić, czy serwer działa poprawnie, przejdź pod adres *http://localhost:3500/products/1*. Jeśli wszystko pójdzie zgodnie z planem, w przeglądarce zobaczysz następujące dane w formacie JSON:

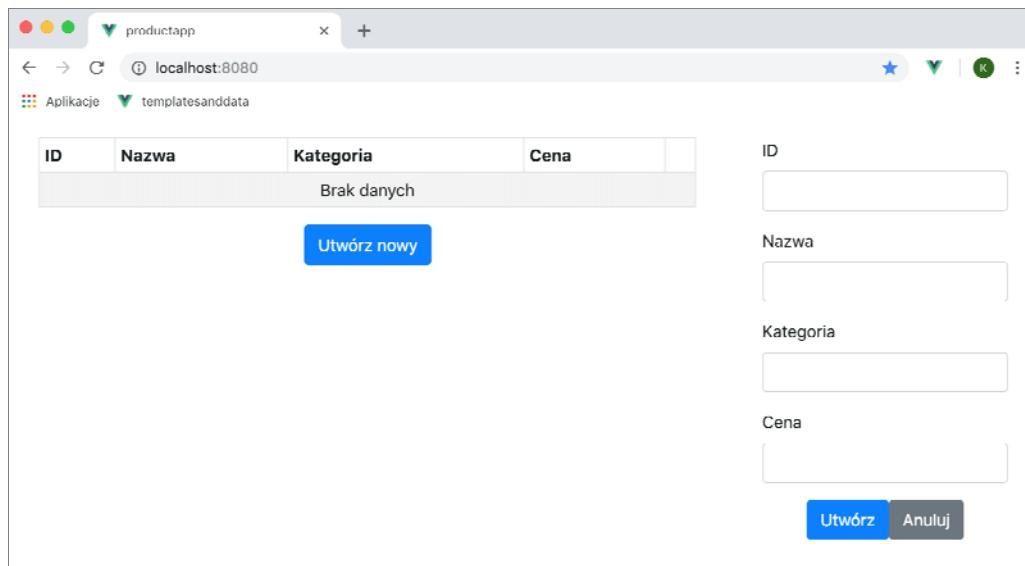
```
...
{
  "id": 1,
  "name": "Kajak",
  "category": "Sporty wodne",
  "price": 275
}
...
```

Serwer HTTP pozostaw otwarty, a obok otwórz drugi wiersz poleceń. Przejdź do katalogu *productapp* i wykonaj polecenie z listingu 19.9, aby uruchomić narzędzia deweloperskie Vue.js.

Listing 19.9. Uruchamianie narzędzi deweloperskich Vue.js

```
npm run serve
```

Po zakończeniu inicjalizacji otwórz okno przeglądarki i przejdź pod adres *http://localhost:8080*, gdzie zobaczysz przykładową aplikację jak na rysunku 19.1.



Rysunek 19.1. Uruchomienie przykładowej aplikacji

Omówienie REST-owych usług sieciowych

Typową metodą na dostarczenie danych do aplikacji jest skorzystanie ze wzorca REST (ang. *Representational State Transfer* – zmiana stanu przez reprezentację) w celu utworzenia usługi sieciowej. REST sam w sobie nie ma określonej specyfikacji, co doprowadziło do powstania wielu różnych konkretnych zbiorów reguł aspirujących do miania REST-owych. Z pewnością można wyróżnić części wspólne, które są użyteczne w tworzeniu aplikacji webowych.

Podstawową zasadą REST-owej usługi sieciowej jest wykorzystanie dobrzejęstw protokołu HTTP, dzięki czemu metody żądań — zwane też czasownikami (ang. *verbs*) — określają rodzaj operacji do wykonania po stronie serwera. Adres URL z kolei określa jeden lub więcej obiektów danych, dla których operacja zostanie zastosowana.

Przykładem jest poniższy adres URL, który może odwoływać się do konkretnego produktu w przykładowej aplikacji:

`http://localhost:3500/products/2`

Pierwszy segment adresu URL — `products` — wskazuje na zbiór obiektów, na których będzie wykonywana operacja. Dzięki tej części adresu jeden serwer może udostępniać wiele usług, z których każda działa na swoich danych. Drugi segment — `2` — określa pojedynczy obiekt z kolekcji `products`. W tym przykładzie jest to wartość właściwości `id`, która jednoznacznie identyfikuje obiekt w kolekcji `products`, będący obiektem kamizelki ratunkowej.

Metoda protokołu HTTP jest używana do określenia rodzaju operacji wykonanej na obiekcie. Gdy przed chwilą testowaliśmy REST-owy serwer, przeglądarka wysłała żądanie HTTP GET, co jest interpretowane przez serwer jako rozkaz pobrania obiektu i przesłanie go do klienta. To właśnie dlatego przeglądarka wygenerowała reprezentację obiektu kamizelki ratunkowej w formacie JSON.

W tabeli 19.3 przedstawiono najbardziej typowe połączenie metod HTTP i adresów URL i wyjaśniono, jak działa każda z nich w momencie wysłania do serwera REST-owego.

Tabela 19.3. Typowe metody HTTP i ich efekty w REST-owej usłudze sieciowej

Metoda	Adres URL	Opis
GET	<code>/products</code>	Pobiera wszystkie obiekty z kolekcji <code>products</code> .
GET	<code>/products/2</code>	Pobiera obiekt o <code>id</code> równym <code>2</code> z kolekcji <code>products</code> .
POST	<code>/products</code>	Dodaje nowy obiekt do kolekcji <code>products</code> . Ciało żądania zawiera reprezentację nowego obiektu w formacie JSON.
PUT	<code>/products/2</code>	Zamienia obiekt w kolekcji <code>products</code> o <code>id</code> równym <code>2</code> . Ciało żądania zawiera reprezentację zamienianego obiektu w formacie JSON.
PATCH	<code>/products/2</code>	Aktualizuje część właściwości obiektu w kolekcji <code>products</code> o <code>id</code> równym <code>2</code> . Ciało żądania zawiera wykaz właściwości do zaktualizowania wraz z nowymi wartościami.
DELETE	<code>/products/2</code>	Usuwa produkt o <code>id</code> równym <code>2</code> z kolekcji <code>products</code> .

Korzystając z tych operacji, należy zachować ostrożność, ponieważ różne usługi REST-owe mogą działać niejednolicie, co może wynikać z różnic we frameworkach zastosowanych do ich utworzenia, a także preferencji zespołu deweloperskiego. Warto potwierdzić, jak dana usługa sieciowa korzysta z czasowników, a także jak konstruowane są adresy URL i ciała żądań.

Do typowych różnic należy odrzucanie treści żądań zawierających wartości identyfikatorów (aby zapewnić, że są one generowane po stronie magazynu danych serwera), a także odrzucanie niektórych czasowników (np.. ignorowanie żądań PATCH, zastępowanych przez żądanie PUT).

Wybór mechanizmu obsługi żądań HTTP

Istnieją trzy różne metody realizacji asynchronicznych żądań HTTP. Możemy skorzystać z obiektu XMLHttpRequest, oryginalnego mechanizmu do wykonywania asynchronicznych żądań, który był bezpośrednio stosowany w czasach, gdy to XML był typowym formatem dla aplikacji webowych. Oto fragment kodu, który wysyła żądanie HTTP do REST-owej usługi sieciowej używanej w tym rozdziale:

```
...
let request = new XMLHttpRequest();
request.onreadystatechange = () => {
  if (request.readyState == XMLHttpRequest.DONE && request.status == 200) {
    this.products.push(...JSON.parse(request.responseText));
  }
};
request.open("GET", "http://localhost:3500/products");
request.send();
...
```

API udostępniane przez obiekt XMLHttpRequest korzysta z funkcji obsługi zdarzeń do otrzymywania aktualizacji, w tym szczegółów odpowiedzi po zakończeniu realizacji żądania na serwerze. XMLHttpRequest to dziwny mechanizm, który nie wspiera nowoczesnych mechanizmów, takich jak słowa kluczowe `async` czy `await`. Możesz jednak mieć pewność, że będzie on dostępny we wszystkich przeglądarkach, które są w stanie uruchomić aplikacje Vue.js. Więcej na jego temat znajdziesz na stronie <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>.

Drugie podejście polega na użyciu Fetch API (API pobierania), które zastępuje obiekt XMLHttpRequest. Fetch API korzysta z obietnic, a nie zdarzeń i — co do zasad — łatwiej się z nim pracuje. Poniżej przedstawiamy fragment kodu równoważny powyższemu:

```
...
fetch("http://localhost:3500/products")
  .then(response => response.json())
  .then(data => this.products.push(...data));
...
...
```

Moim zdaniem Fetch API korzysta ze zbyt dużej liczby obietnic. Metoda `fetch` jest używana do wykonywania żądania HTTP, które zwraca obietnicę generującą obiekt wyników. Metoda `json` generuje ostateczny wynik przetworzony z danych w formacie JSON. Fetch API może być używane w powiązaniu ze słowami kluczowymi `async` i `await`, jednak obsługa wielu obietnic wymaga starannego podejścia, a mimo to może powodować powstanie takich potworków:

```
...
this.products.push(
  ...await (await fetch("http://localhost:3500/products")).json());
...
...
```

Fetch API to krok naprzód w porównaniu z obiektem XMLHttpRequest, jednak nie jest ono obsługiwane we wszystkich przeglądarkach, które obsługują Vue. Więcej na jego temat znajdziesz pod adresem https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

Trzecie podejście opiera się na użyciu pakietu, który obsługuje obiekt XMLHttpRequest, jednocześnie ukrywając szczegóły stosowania tego obiektu. Taki pakiet powinien udostępniać API spójne z resztą procesu deweloperskiego Vue.js. W tym rozdziale korzystam z pakietu Axios, ponieważ jest on najpopularniejszy, ale można znaleźć również inne biblioteki. Vue.js nie ma „oficjalnej” paczki do obsługi protokołu HTTP, ale można wybierać spośród licznych opcji, nawet jeśli nie były pisane z myślą o Vue.js. Wadą zastosowania pakietu HTTP jest konieczność zwiększenia rozmiaru Twojej aplikacji, ponieważ konieczne będzie dołączenie dodatkowej paczki wymaganej przez przeglądarkę.

Konsumowanie REST-owej usługi sieciowej

Najważniejszą cechą żądań HTTP w aplikacji webowej jest ich asynchroniczność. Choć może to wydawać się oczywiste, dla niektórych brak natychmiastowej odpowiedzi z serwera nie jest jasny. W związku z tym kod obsługi żądań HTTP musi być pisany starannie. Konieczne jest skorzystanie z mechanizmów języka JavaScript, które poradzą sobie z asynchronicznymi operacjami. Z uwagi na niejasny proces realizacji żądań kod obsługi pierwszego żądania przedstawię, tłumacząc kolejne operacje krok po kroku.

Wykonywanie żądań o różnym pochodzeniu

Domyślnie przeglądarki dopuszczają politykę bezpieczeństwa, która pozwala na wykonywanie żądań HTTP do tego samego serwera co dokument, z którego pochodzi skrypt. W ten sposób redukowane jest ryzyko ataków typu **XSS** (ang. *Cross-Site Scripting* — skrypty międzywitrynowe), w których przeglądarka nieświadomie wykonuje złośliwy kod (por. http://pl.wikipedia.org/wiki/Cross-site_scripting). Dla programistów aplikacji webowych *polityka tego samego źródła* może stanowić problem, ponieważ usługi sieciowe znajdują się poza źródłem pochodzenia oryginalnego kodu JavaScript. Dwa adresy URL uznaje się za pochodzące z tego samego źródła, jeśli łączą je protokół, host i port. W tym przypadku adres usługi REST-owej ma inne pochodzenie z uwagi na inny numer portu TCP.

Protokół **CORS** (ang. *Cross-Origin Resource Sharing* — współdzielenie zasobów o różnym pochodzeniu) jest używany do wysyłania żądań do różnych źródeł. Dzięki CORS przeglądarka dołącza nagłówki w asynchronicznych żądaniach HTTP, które przekazują do serwera źródło kodu JavaScript. Odpowiedź z serwera zawiera nagłówki, które informują przeglądarkę, czy serwer akceptuje dane żądanie. Szczegóły protokołu CORS wykraczają poza zakres tej książki, ale warto zajrzeć na stronę https://en.wikipedia.org/wiki/Cross-origin_resource_sharing i do specyfikacji CORS dostępnej pod adresem www.w3.org/TR/cors.

W tym rozdziale CORS jest stosowany automatycznie. Pakiet json-server obsługuje CORS i akceptuje żądania z dowolnego źródła. Pakiet Axios używany do wykonywania żądań automatycznie dołącza nagłówki CORS. Wybierając oprogramowanie we własnych projektach, musisz jednak pamiętać o wyborze platformy, która akceptuje wszystkie żądania z pojedynczego źródła, lub skonfigurować CORS tak, aby serwer akceptował żądania danych pochodzące z aplikacji.

Obsługa danych odpowiedzi

Choć może wydawać się to niezbyt intuicyjne, pracę najlepiej jest zacząć od kodu obsługującego odpowiedzi przesypane z serwera. W listingu 19.10 do elementu script komponentu ProductDisplay dodaję metodę, która przetwarza dane produktu po otrzymaniu ich z usługi sieciowej.

Listing 19.10. Dodawanie metody w pliku src/components/ProductDisplay.vue

```
...
<script>
    import Vue from "vue";
    export default {
        data: function () {
            return {
                products: []
            }
        },
        methods: {
            createNew() {
```

```

        this.eventBus.$emit("create");
    },
    editProduct(product) {
        this.eventBus.$emit("edit", product);
    },
    processProducts(newProducts) {
        this.products.splice(0);
        this.products.push(...newProducts);
    }
},
inject: ["eventBus"]
}
</script>
...

```

Metoda `processProducts` przyjmuje tablicę obiektów produktów i korzysta z nich do zamiany zawartości właściwości danych o nazwie `products`. Jak objaśniłem w rozdziale 13., Vue.js ma problemy z wykrywaniem zmian w tablicach, dlatego korzystam z metody `splice`, aby usunąć istniejące obiekty. Następnie korzystam z operatora rozwinięcia, aby odpakować wartości z argumentu metody, i używam metody `push`, aby wypełnić tablicę. Skoro tablica `products` jest reaktywną właściwością danych, Vue.js wykryje zmiany automatycznie i zaktualizuje wiązania danych, aby odzwierciedlić nowe dane.

Wykonywanie żądania HTTP

Kolejnym krokiem jest wykonanie żądania HTTP i odpytanie REST-owej usługi sieciowej o dane. W listingu 19.11 zaimportowałem pakiet Axios i wykorzystałem go do wykonania żądania HTTP.

Listing 19.11. Wykonywanie żądania HTTP w pliku src/components/ProductDisplay.vue

```

...
<script>
import Vue from "vue";
import Axios from "axios";
const baseUrl = "http://localhost:3500/products/";
export default {
    data: function () {
        return {
            products: []
        }
    },
    methods: {
        createNew() {
            this.eventBus.$emit("create");
        },
        editProduct(product) {
            this.eventBus.$emit("edit", product);
        },
        processProducts(newProducts) {
            this.products.splice(0);
            this.products.push(...newProducts);
        }
    },
    inject: ["eventBus"],
    created() {
        Axios.get(baseUrl);
    }
}

```

```

        }
    </script>
    ...

```

Axios dostarcza metody dla każdego rodzaju żądań HTTP, np. żądania GET są wykonywane za pomocą metody get, żądania POST — za pomocą metody post itd. Dostępna jest także metoda request, która przyjmuje obiekt konfiguracji i może być używana do wykonywania wszelkich rodzajów żądań — korzystam z niej w punkcie „Tworzenie usługi obsługi błędów”.

Żądanie HTTP wykonuję w metodzie komponentu created. Moim celem jest wyzwolenie mechanizmu zmiany w momencie otrzymania danych z żądania. Zastosowanie metody created zapewnia, że właściwości danych komponentu będą przetworzone przed wykonaniem żądania do serwera HTTP.

-
- **Wskazówka** Niektórzy programiści lubią korzystać z metody mounted do wykonywania początkowych żądań HTTP. Nie ma znaczenia, z której metody korzystasz, najważniejsze jest, abyś postępował spójnie we wszystkich komponentach.
-

Otrzymywanie odpowiedzi

Wynikiem działania metody get biblioteki Axios jest Promise (obietnica), która udostępnii odpowiedź z serwera w momencie zakończenia żądania HTTP. Jak objaśniłem w rozdziale 4., metoda then pozwala na wykonanie akcji po zakończeniu obietnicy. W listingu 19.12 korzystam z metody then, aby przetworzyć odpowiedź HTTP.

Listing 19.12. Otrzymywanie odpowiedzi HTTP w pliku src/components/ProductDisplay.vue

```

...
<script>
    import Vue from "vue";
    import Axios from "axios";
    const baseUrl = "http://localhost:3500/products/";
    export default {
        data: function () {
            return {
                products: []
            }
        },
        methods: {
            createNew() {
                this.eventBus.$emit("create");
            },
            editProduct(product) {
                this.eventBus.$emit("edit", product);
            },
            processProducts(newProducts) {
                this.products.splice(0);
                this.products.push(...newProducts);
            }
        },
        inject: ["eventBus"],
        created() {
            Axios.get(baseUrl).then(resp => {
                console.log(`Odpowiedź HTTP: ${resp.status}, ${resp.statusText}`);
                console.log(`Dane odpowiedzi: ${resp.data.length} elementów`);
            });
        }
    }

```

```

        }
    }
</script>
...

```

Metoda `then` otrzymuje z biblioteki Axios obiekt, który reprezentuje odpowiedź z serwera, zawierającą właściwości opisane w tabeli 19.4.

Tabela 19.4. Właściwości odpowiedzi Axios

Nazwa	Opis
<code>status</code>	Ta właściwość zwraca kod statusu odpowiedzi, np. <code>200</code> lub <code>404</code> .
<code>statusText</code>	Ta właściwość zwraca tekst objaśnienia, który towarzyszy statusowi odpowiedzi, np. <code>OK</code> lub <code>Not Found (Nie znaleziono)</code> .
<code>headers</code>	Ta właściwość zwraca obiekt, którego właściwości reprezentują nagłówki odpowiedzi.
<code>data</code>	Ta właściwość zwraca treść zawartą w odpowiedzi.
<code>config</code>	Ta właściwość zwraca obiekt, który zawiera opcje konfiguracji zastosowane w żądaniu.
<code>request</code>	Ta właściwość zwraca obiekt XMLHttpRequest zastosowany do wykonania żądania.

W listingu 19.12 korzystam z właściwości `status` i `statusText`, aby wypisać szczegóły dotyczące odpowiedzi w konsoli przeglądarki JavaScript. Najbardziej interesującą właściwością jest `data`, ponieważ zwraca ona odpowiedź z serwera. Axios automatycznie dekoduje odpowiedzi w formacie JSON, co oznacza, że mogę odczytać właściwość `length`, aby dowiedzieć się, ile obiektów zostało uwzględnionych w odpowiedzi. Zapisz zmiany w komponencie i sprawdź konsolę JavaScript, a zobacysz następujące komunikaty:

```

...
HTTP Response: 200, OK
Response Data: 9 items
...

```

Przetwarzanie danych

Ostatni krok polega na odczytaniu właściwości `data` z obiektu odpowiedzi i przekazaniu jej do metody `processProducts`, dzięki czemu możemy zaktualizować stan aplikacji (listing 19.13).

Listing 19.13. Przetwarzanie odpowiedzi w pliku `src/components/ProductDisplay.vue`

```

...
<script>
  import Vue from "vue";
  import Axios from "axios";
  const baseUrl = "http://localhost:3500/products/";
  export default {
    data: function () {
      return {
        products: []
      }
    },
    methods: {
      createNew() {
        this.eventBus.$emit("create");
      },
      editProduct(product) {
        this.eventBus.$emit("edit", product);
      },
    }
  }

```

```

        processProducts(newProducts) {
            this.products.splice(0);
            this.products.push(...newProducts);
        }
    },
    inject: ["eventBus"],
    created() {
        Axios.get(baseUrl).then(resp => this.processProducts(resp.data));
    }
}
</script>
...

```

Komponent wykona żądanie HTTP GET w metodzie created. Po otrzymaniu odpowiedzi z serwera Axios przetworzy zawarte dane w formacie JSON i udostępní jako element odpowiedzi. Dane odpowiedzi są używane do wypełnienia tablicy products komponentu. Następująca po tym aktualizacja wywoła dyrektywę v-for w szablonie i wyświetli dane przedstawione na rysunku 19.2.

The screenshot shows a web application window titled "productapp" at "localhost:8080". The page displays a table of products with columns: ID, Nazwa, Kategoria, and Cena. Each row has a blue "Edytuj" button. To the right of the table is a form with fields for ID, Nazwa, Kategoria, and Cena, each with an associated input field. Below the form are two buttons: "Utwórz" (Create) and "Anuluj" (Cancel). At the bottom left is a blue "Utwórz nowy" (Create new) button.

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button>
4	Choragiewki narciarskie	Piłka nożna	34.95	<button>Edytuj</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button>
6	Myśląca czapeczka	Szachy	16	<button>Edytuj</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button>
8	Szachownica	Szachy	75	<button>Edytuj</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button>

Utwórz nowy

Rysunek 19.2. Pobieranie danych z usługi sieciowej

Mogę zmodyfikować ten kod, korzystając ze słów kluczowych `async` i `await`, dzięki czemu będę mógł pominąć wywołanie metody `then`. W listingu 19.14 stosuję słowo kluczowe `async` w metodzie `created`, a słowo `await` przy okazji wykonania żądania HTTP.

Listing 19.14. Uproszczenie kodu żądania w pliku `src/components/ProductDisplay.vue`

```

...
async created() {
    let data = (await Axios.get(baseUrl)).data;
    this.processProducts(data);
}
...

```

Ten kod działa identycznie jak w listingu 19.13, ale bez użycia metody `then`, określającej instrukcję, które zostaną wykonane w momencie zakończenia żądania HTTP.

Tworzenie usługi HTTP

Przed rozpoczęciem kolejnej części zmian muszę nieco zmodyfikować strukturę aplikacji. W poprzednim podrozdziale pokazałem, jak łatwo wykonywać żądania HTTP w komponencie, ale efektem mojej zmiany jest przemieszanie kodu widoku z kodem odpowiedzialnym za komunikację z serwerem. W miarę zwiększania się liczby rodzajów żądań komponent będzie musiał coraz więcej uwagi poświęcać obsłudze protokołu HTTP.

Do katalogu `src` dodaje plik `restDataSource.js` i wykorzystuję go do zdefiniowania klasy języka JavaScript (listing 19.15).

Listing 19.15. Zawartość pliku `src/restDataSource.js`

```
import Axios from "axios";
const baseUrl = "http://localhost:3500/products/";
export class RestDataSource {
    async getProducts() {
        return (await Axios.get(baseUrl)).data;
    }
}
```

Skorzystałem z możliwości tworzenia klas w języku JavaScript, aby zdefiniować klasę `RestDataSource`, która dysponuje asynchroniczną metodą `getProducts`. Metoda ta wykorzystuje Axiosa do wysłania żądania HTTP do REST-owej usługi sieciowej i zwraca pobrane dane. W listingu 19.16 tworzę instancję klasy `RestDataSource` i konfiguruje ją jako usługę w pliku `main.js`, dzięki czemu będzie ona dostępna w całej aplikacji.

Listing 19.16. Konfiguracja usługi w pliku `src/main.js`

```
import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
import { RestDataSource } from "./restDataSource";
Vue.config.productionTip = false;
new Vue({
    render: h => h(App),
    provide: function () {
        return {
            eventBus: new Vue(),
            restDataSource: new RestDataSource()
        }
    }
}).$mount('#app')
```

Teraz nowa usługa nosi nazwę `restDataSource` i będzie dostępna dla wszystkich komponentów aplikacji.

Konsumowanie usługi HTTP

Po zdefiniowaniu usługi mogę usunąć kod Axiosa z komponentu `ProductDisplay` i skorzystać z usługi (listing 19.17).

Listing 19.17. Zastosowanie usługi HTTP w pliku `src/components/ProductDisplay.vue`

```
...
<script>
    import Vue from "vue";
    // import Axios from "axios";
    // const baseUrl = "http://localhost:3500/products/";
    export default {
        data: function () {
```

```

        return {
            products: []
        }
    },
methods: {
    createNew() {
        this.eventBus.$emit("create");
    },
    editProduct(product) {
        this.eventBus.$emit("edit", product);
    },
    processProducts(newProducts) {
        this.products.splice(0);
        this.products.push(...newProducts);
    }
},
inject: ["eventBus", "restDataSource"],
async created() {
    this.processProducts(await this.dataSource.getProducts());
}
}
</script>
...

```

Właściwość inject deklaruje zależność od usługi restDataSource, która jest używana w metodzie created do pobierania danych z REST-owej usługi sieciowej i wypełniania właściwości danych o nazwie products.

Dodawanie pozostałych operacji HTTP

Teraz, gdy mamy już dobrze zdefiniowaną strukturę, mogę dodać kompletny zbiór operacji HTTP wymaganych przez aplikację, rozszerzając usługę w celu obsłużenia metod dostarczonych przez Axiosa (listing 19.18).

Listing 19.18. Dodawanie operacji w pliku src/restDataSource.js

```

import Axios from "axios";
const baseUrl = "http://localhost:3500/products/";
export class RestDataSource {
    async getProducts() {
        return (await Axios.get(baseUrl)).data;
    }
    async saveProduct(product) {
        await Axios.post(baseUrl, product);
    }
    async updateProduct(product) {
        await Axios.put(`/${baseUrl}/${product.id}`, product);
    }
    async deleteProduct(product) {
        await Axios.delete(`/${baseUrl}/${product.id}`, product);
    }
}

```

Dołączyłem metody odpowiedzialne za dodawanie nowych obiektów, aktualizację obiektów już istniejących, a także usuwanie obiektów. Wszystkie metody korzystają ze słów kluczowych `async` i `await`, które pozwalają komponentowi na oczekiwanie na wynik żądania. Jest to niezwykle ważne, ponieważ w ten sposób komponent może wstrzymać się z aktualizacją stanu do skutecznego zakończenia żądania HTTP.

- **Wskazówka** Zrestartuj pakiet json-server, aby przywrócić przykładowe dane do stanu początkowego po usunięciu lub zmianie elementów. Zawartość pliku JavaScript, utworzonego w listingu 19.2, zostanie użyta do odtworzenia bazy danych w momencie uruchomienia procesu.

W listingu 19.19 zmieniłem komponent ProductDisplay, aby skorzystać z metod zdefiniowanych w listingu 19.18 w celu obsłużenia usuwania obiektów.

Listing 19.19. Dodawanie operacji na danych w pliku *src/components/ProductDisplay.vue*

```
<template>
  <div>
    <table class="table table-sm table-striped table-bordered">
      <tr>
        <th>ID</th>
        <th>Nazwa</th>
        <th>Kategoria</th>
        <th>Cena</th>
        <th></th>
      </tr>
      <tbody>
        <tr v-for="p in products" v-bind:key="p.id">
          <td>{{ p.id }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.category }}</td>
          <td>{{ p.price }}</td>
          <td>
            <button class="btn btn-sm btn-primary"
                   v-on:click="editProduct(p)">
              Edytuj
            </button>
            <button class="btn btn-sm btn-danger"
                   v-on:click="deleteProduct(p)">
              Usuń
            </button>
          </td>
        </tr>
        <tr v-if="products.length == 0">
          <td colspan="5" class="text-center">Brak danych</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="createNew">
        Utwórz nowy
      </button>
    </div>
  </div>
</template>
<script>
  import Vue from "vue";
  export default {
    data: function () {
      return {
        products: []
      }
    },
    methods: {
```

```

createNew() {
    this.eventBus.$emit("create");
},
editProduct(product) {
    this.eventBus.$emit("edit", product);
},
async deleteProduct(product) {
    await this.dataSource.deleteProduct(product);
    let index = this.products.findIndex(p => p.id == product.id);
    this.products.splice(index, 1);
},
processProducts(newProducts) {
    this.products.splice(0);
    this.products.push(...newProducts);
},
async processComplete(product) {
    let index = this.products.findIndex(p => p.id == product.id);
    if (index == -1) {
        await this.dataSource.saveProduct(product);
        this.products.push(product);
    } else {
        await this.dataSource.updateProduct(product);
        Vue.set(this.products, index, product);
    }
}
},
inject: ["eventBus", "dataSource"],
async created() {
    this.processProducts(await this.dataSource.getProducts());
    this.eventBus.$on("complete", this.processComplete);
}
}
</script>

```

Przy wywołaniu metod asynchronicznych usługi HTTP niezwykle ważne jest zastosowanie słowa kluczowego `await`. Jeśli pominiesz to słowo, wszelkie dalsze instrukcje w metodzie komponentu zostaną wywołane natychmiast, niezależnie od wyniku żądania HTTP. W przypadku operacji, które mają przechować lub usunąć obiekty, oznacza to, że aplikacja pokaże komunikat o skutecznym zakończeniu operacji niezależnie od faktycznego jej wyniku. Na przykład zastosowanie słowa `await` w poniższej metodzie uchroni komponent od usunięcia obiektu z tablicy `products`, dopóki żądanie HTTP nie zostanie zakończone:

```

...
async deleteProduct(product) {
    await this.dataSource.deleteProduct(product);
    let index = this.products.findIndex(p => p.id == product.id);
    this.products.splice(index, 1);
},
...

```

Gdy korzystam ze słowa `await`, muszę liczyć się z tym, że dowolny błąd, który wystąpi w wyniku wykonania asynchronicznej operacji, spowoduje rzucenie wyjątku w metodzie komponentu, co powstrzyma wykonanie instrukcji zawartych w tej metodzie. W tym przypadku powstrzyma to desynchronizację danych użytkownika z danymi z serwera.

- **Wskazówka** Gdy korzystasz ze słowa kluczowego `await` w metodzie komponentu, musisz także zastosować słowo `async` (listing 19.19).

Efektem tych modyfikacji jest możliwość odczytania zmian z REST-owej usługi sieciowej, a także tworzenie, edycja i usuwanie produktów (rysunek 19.3).

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button>
4	Chorągiewki narciarskie	Piłka nożna	34.95	<button>Edytuj</button>
5	Stadion	Piłka nożna	795.00	<button>Edytuj</button>
6	Międzyczłasowa czapka	Szachy	16	<button>Edytuj</button>
7	Chwilejne krzesło	Szachy	29.95	<button>Edytuj</button>
8	Szachownica	Szachy	75	<button>Edytuj</button>
9	Królewiątko złote	Szachy	1200	<button>Edytuj</button>
100	Buty do biegania	Bieganie	100	<button>Edytuj</button>

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>
4	Chorągiewki narciarskie	Piłka nożna	34.95	<button>Edytuj</button> <button>Usuń</button>
5	Stadion	Piłka nożna	795.00	<button>Edytuj</button> <button>Usuń</button>
6	Międzyczłasowa czapka	Szachy	16	<button>Edytuj</button> <button>Usuń</button>
7	Chwilejne krzesło	Szachy	29.95	<button>Edytuj</button> <button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button> <button>Usuń</button>
9	Królewiątko złote	Szachy	1200	<button>Edytuj</button> <button>Usuń</button>

Rysunek 19.3. Wykonywanie operacji HTTP

Tworzenie usługi obsługi błędów

Vue.js nie potrafi wykryć błędów, które powstają w wyniku asynchronicznych operacji HTTP. Dodatkowa praca jest konieczna, jeśli chcemy poinformować użytkownika o zaistnieniu błędu. Osobiście uważam, że należy utworzyć komponent przeznaczony do wyświetlania błędów, a to klasa RestDataSource powinna wyświetlać powiadomienia o błędach, wyświetlając własne zdarzenia za pomocą szyny zdarzeń. W listingu 19.20 dodaje obsługę wyjątków w klasie RestDataSource, generując własne zdarzenie.

Listing 19.20. Obsługa błędów w pliku src/restDataSource.js

```
import Axios from "axios";
const baseUrl = "http://localhost:3500/products/";
export class RestDataSource {
    constructor(bus) {
        this.eventBus = bus;
    }
    async getProducts() {
        return (await this.sendRequest("GET", baseUrl)).data;
    }
    async saveProduct(product) {
        await this.sendRequest("POST", baseUrl, product);
    }
    async updateProduct(product) {
        await this.sendRequest("PUT", `${baseUrl}${product.id}`, product);
    }
    async deleteProduct(product) {
        await this.sendRequest("DELETE", `${baseUrl}${product.id}`, product);
    }
    async sendRequest(httpMethod, url, product) {
        try {
            return await Axios.request({
                method: httpMethod,
                url: url,
                data: product
            });
        
```

```

        } catch (err) {
            if (err.response) {
                this.eventBus.$emit("httpError",
                    `${err.response.statusText} - ${err.response.status}`);
            } else {
                this.eventBus.$emit("httpError", "HTTP Error");
            }
            throw err;
        }
    }
}

```

Zwykłe klasy nie uczestniczą w cyklu życia komponentu i nie mogą używać właściwości inject w celu otrzymywania usług. W związku z tym dodałem konstruktor, który przyjmuje szynę zdarzeń, i przepisałem klasę, dzięki czemu wszystkie metody korzystają z metody sendRequest, która wewnętrznie używa metody request biblioteki Axios. Ta metoda pozwala na określenie szczegółów żądania za pomocą obiektu konfiguracji i umożliwia skonsolidowanie kodu, który wykonuje żądania HTTP, dzięki czemu mogę spóźnione obsługiwać błędy.

Axios nie zawsze jest w stanie dostarczyć obiekt, który zawiera odpowiedź. Dzieje się tak na przykład w sytuacji, gdy żądanie przekracza czas potrzebny na wykonanie. Wówczas dostarczam ogólny opis, który wskazuje, że problem wyniknął z żądania HTTP.

Metody Axiosa rzucają błąd, gdy żądanie HTTP zwraca kod statusu z grup 400 i 500, które wskazują na zaistnienie problemu. W listingu 19.20 stosuję blok try/catch, aby złapać wyjątek i wysłać własne zdarzenie o nazwie httpError. Obiekt otrzymany w bloku catch to właściwość response, która zwraca obiekt reprezentujący odpowiedź z serwera. Obiekt ten zawiera właściwości opisane w tabeli 19.4. Skorzystałem z nich, aby skonstruować prosty komunikat towarzyszący własnemu zdarzeniu.

- **Wskazówka** Zwróć uwagę, że mimo wystąpienia własnego zdarzenia wciąż rzucam (throw) to zdarzenie. Wynika to z faktu, że to komponent, który zapoczątkował zdarzenie HTTP, otrzymuje wyjątek i nie można oczekiwac od niego automatycznej aktualizacji lokalnej reprezentacji danych. Bez instrukcji throw tylko odbiorcy własnego zdarzenia będą wiedzieć, że wystąpił problem.

Konstruktor z listingu 19.20 wymaga szyny zdarzeń, zadeklarowanej w pliku *main.js* (listing 19.21).

Listing 19.21. Konfiguracja usługi w pliku *src/main.js*

```

import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
import { RestDataSource } from "./restDataSource";
Vue.config.productionTip = false
new Vue({
    render: h => h(App),
    data: {
        eventBus: new Vue()
    },
    provide: function () {
        return {
            eventBus: this.eventBus,
            restDataSource: new RestDataSource(this.eventBus)
        }
    }
}).$mount('#app')

```

Właściwości, które definiują usługi, nie mogą odwoływać się do innych usług, dlatego definiuję właściwość danych odpowiedzialną za utworzenie szyny zdarzeń, którą udostępniam bezpośrednio za pomocą własnej właściwości `provide`, a także za pomocą argumentu konstruktora klasy `RestDataSource`.

Tworzenie komponentu do wyświetlania zdarzeń

Na koniec muszę utworzyć komponent, który otrzyma własne zdarzenia i wyświetli komunikat o błędach. Do katalogu `src/components` dodaję plik `ErrorDisplay.vue` o treści z listingu 19.22.

Listing 19.22. Zawartość pliku `src/components/ErrorDisplay.vue`

```
<template>
<div v-if="error" class="bg-danger text-white text-center p-3 h3">
    Wystąpił błąd
    <h6>{{ message }}</h6>
    <a href="/" class="btn btn-secondary">OK</a>
</div>
</template>
<script>
export default {
  data: function () {
    return {
      error: false,
      message: ""
    }
  },
  methods: {
    handleError(err) {
      this.error = true;
      this.message = err;
    }
  },
  inject: ["eventBus"],
  created() {
    this.eventBus.$on("httpError", this.handleError);
  }
}
</script>
```

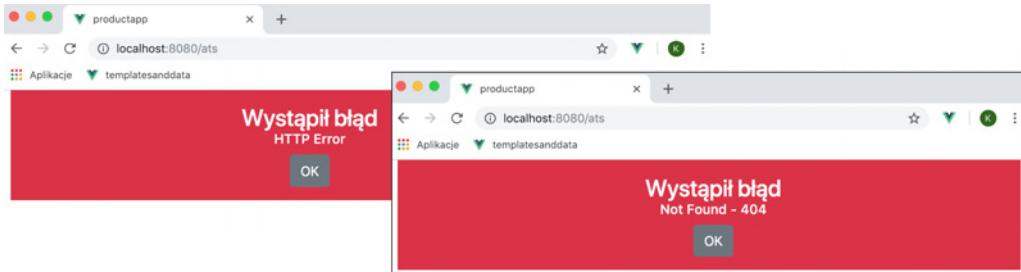
Ten komponent rejestruje się na własne zdarzenie w metodzie `created` za pomocą szyny zdarzeń i reaguje na otrzymanie zdarzenia, ujawniając element za pomocą dyrektywy `v-if`. Aby przedstawić komponent użytkownikowi, dokonam zmian widocznych w komponencie głównym aplikacji (listing 19.23).

Listing 19.23. Wdrażanie komponentu w pliku `src/App.vue`

```
<template>
<div class="container-fluid">
  <div class="row">
    <div class="col"><error-display /></div>
  </div>
  <div class="row">
    <div class="col-8 m-3"><product-display/></div>
    <div class="col m-3"><product-editor/></div>
  </div>
</div>
</template>
<script>
import ProductDisplay from "./components/ProductDisplay";
```

```
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";
export default {
  name: 'App',
  components: { ProductDisplay, ProductEditor, ErrorDisplay }
}
</script>
```

Instrukcja import przypisuje nazwę ErrorDisplay do komponentu, co prowadzi do rejestracji go w ramach aplikacji. Dzięki temu element error-display może być dodany do szablonu komponentu głównego aplikacji. Za sprawą wprowadzonego komponentu wszystkie błędy napotkane w trakcie obsługi żądania HTTP są wyświetlane użytkownikowi (rysunek 19.4).



Rysunek 19.4. Wyświetlanie błędów

- **Wskazówka** Jeśli chcesz przetestować obsługę błędów, to najprostszą metodą jest zatrzymanie procesu json-server i zmiana wartości atrybutu baseUrl w klasie RestDataSource w taki sposób, aby wskazywała ona na adres URL, który nie istnieje, np. <http://localhost:3500/hats/>.

Przywracanie kontroli po błędach komunikacji

Gdy muszę obsługiwać błędy w komponentach, stosuję podejście wszystko-albo-nic. W takiej sytuacji użytkownik widzi przycisk OK, za pomocą którego przechodzi do głównego adresu URL, czyli w praktyce następuje odświeżenie aplikacji i wczytanie najbardziej aktualnych danych. Wadą tego podejścia jest utrata lokalnego stanu, co może prowadzić do poiryutowania użytkownika, zwłaszcza w sytuacji, gdy próbuje on wykonywać skomplikowane zadanie za każdym razem bezskutecznie.

W związku z tym lepiej jest pozwolić użytkownikowi poprawić dane i wykonać żądanie HTTP ponownie. Takie podejście można zastosować tylko, jeśli przyczyna problemu jest znana, a rozwiązanie — oczywiste. Nie zawsze łatwo jest rozwiązać problemy z żądaniem HTTP, nawet jeśli serwer dostarcza dodatkowych informacji.

Podsumowanie

W tym rozdziale pokazałem, jak można skorzystać z REST-owej usługi sieciowej w aplikacji Vue.js za pomocą pakietu Axios. Zademonstrowałem operacje na danych za pomocą żądań HTTP, a także utworzyłem odrębną klasę do obsługi żądań HTTP. Omówiłem też sposoby obsługi błędów z tym związanymi. W następnym rozdziale omówię sposoby użycia pakietu Vuex w celu utworzenia współdzielonego magazynu danych.

ROZDZIAŁ 20.

Stosowanie magazynu danych

W tym rozdziale pokażę, jak zastosować pakiet Vuex do utworzenia magazynu danych, który pozwala na współdzielenie danych przez całą aplikację i koordynuje współpracę komponentów. Tabela 20.1 umiejscawia magazyn danych Vuex w szerszym kontekście.

Tabela 20.1. Umiejscowienie magazynu danych Vuex w szerszym kontekście

Pytanie	Odpowiedź
Czym jest magazyn danych?	Magazyn danych to wspólne repozytorium stanu aplikacji, zarządzane przez pakiet Vuex, będący oficjalną częścią projektu Vue.js.
Dlaczego jest użyteczny?	Magazyn danych może uprościć zarządzanie danymi przez udostępnienie ich w całej aplikacji.
Jak się z niego korzysta?	Magazyn danych jest tworzony za pomocą pakietu Vuex i rejestrowany w pliku <i>main.js</i> , dzięki czemu każdy komponent ma do niego dostęp przy użyciu specjalnej właściwości <code>\$store</code> .
Czy są jakieś pułapki lub ograniczenia?	Magazyny danych Vuex działają w dość specyficzny sposób, który na początku może wydawać się mało intuicyjny. Warto, żebyś miał włączony tryb ścisły do czasu, gdy nie zapoznasz się lepiej z Vuex. Nie zapomnij jednak wyłączyć tej funkcji przed wdrożeniem aplikacji.
Czy są jakieś rozwiązania alternatywne?	Jeśli nie lubisz biblioteki Vuex, to aby osiągnąć podobne efekty, możesz korzystać z wstrzykiwania zależności i szyny zdarzeń, opisanych w rozdziale 18.

Tabela 20.2 podsumowuje rozdział.

Przygotowania do tego rozdziału

W tym rozdziale korzystam z aplikacji *productapp* z rozdziału 19. Aby uruchomić REST-ową usługę sieciową, otwórz wiersz poleceń i wykonaj polecenie z listingu 20.1 w katalogu *productapp*.

Tabela 20.2. Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Utwórz magazyn danych.	Zdefiniuj nowy moduł i skorzystaj z niego, aby zarejestrować wtyczkę Vuex i utworzyć obiekt Vuex.Store.	20.6
Zarejestruj magazyn danych.	Zimportuj moduł magazynu danych i dodaj właściwość store do konfiguracji obiektu Vue.	20.7
Uzyskaj dostęp do magazynu danych w komponencie.	Skorzystaj z właściwości \$store.	20.8
Wprowadź zmianę w magazynie danych.	Wykonaj mutację.	20.9
Zdefiniuj właściwość obliczaną w magazynie danych.	Skorzystaj z gettera.	20.10 – 20.13
Wykonaj zadanie asynchroniczne w magazynie danych.	Skorzystaj z akcji.	20.14 – 20.16
Obserwuj zmiany w magazynie danych.	Skorzystaj z obserwatora.	20.17 – 20.19
Powiąz mechanizmy magazynu danych z właściwościami obliczanymi i metodami komponentu.	Skorzystaj z funkcji mapujących.	20.20
Podziel magazyn danych na kilka odrębnych plików.	Utwórz dodatkowe moduły magazynu danych.	20.21 – 20.24
Wyodrębnij niektóre mechanizmy do modułu z pozostałej części magazynu danych.	Skorzystaj z mechanizmu przestrzeni nazw.	20.25 – 20.26

Listing 20.1. Uruchamianie usługi sieciowej

```
npm run json
```

■ **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Otwórz drugie okno wiersza poleceń, przejdź do katalogu *productapp* i wykonaj polecenie z listingu 20.2, aby pobrać i zainstalować pakiet Vuex.

Listing 20.2. Instalacja pakietu Vuex

```
npm install vuex@3.0.1
```

Aby przygotować się do tego rozdziału, usunąłem z komponentu *ProductDisplay* wszystkie instrukcje, które zarządzają danymi produktu, i skorzystałem z szyny zdarzeń, pozostawiając puste metody wywoływane w momencie kliknięcia przycisku (listing 20.3).

Listing 20.3. Uproszczenie treści pliku src/components/ProductDisplay.vue

```

<template>
  <div>
    <table class="table table-sm table-striped table-bordered">
      <tr>
        <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
      </tr>
      <tbody>
        <tr v-for="p in products" v-bind:key="p.id">
          <td>{{ p.id }}</td>
          <td>{{ p.name }}</td>
          <td>{{ p.category }}</td>
          <td>{{ p.price }}</td>
          <td>
            <button class="btn btn-sm btn-primary"
                   v-on:click="editProduct(p)">
              Edytuj
            </button>
            <button class="btn btn-sm btn-danger"
                   v-on:click="deleteProduct(p)">
              Usuń
            </button>
          </td>
        </tr>
        <tr v-if="products.length == 0">
          <td colspan="5" class="text-center">Brak danych</td>
        </tr>
      </tbody>
    </table>
    <div class="text-center">
      <button class="btn btn-primary" v-on:click="createNew">
        Utwórz nowy
      </button>
    </div>
  </div>
</template>
<script>
  export default {
    data: function () {
      return {
        products: []
      }
    },
    methods: {
      createNew() { },
      editProduct(product) { },
      deleteProduct(product) { }
    }
  }
</script>

```

Usunąłem także szynę zdarzeń z komponentu ProductEditor (listing 20.4).

Listing 20.4. Uproszczenie treści pliku src/components/ProductEditor.vue

```

<template>
  <div>
    <div class="form-group">

```

```

<label>ID</label>
<input class="form-control" v-model="product.id" />
</div>
<div class="form-group">
    <label>Nazwa</label>
    <input class="form-control" v-model="product.name" />
</div>
<div class="form-group">
    <label>Kategoria</label>
    <input class="form-control" v-model="product.category" />
</div>
<div class="form-group">
    <label>Cena</label>
    <input class="form-control" v-model.number="product.price" />
</div>
<div class="text-center">
    <button class="btn btn-primary" v-on:click="save">
        {{ editing ? "Zapisz" : "Utwórz" }}
    </button>
    <button class="btn btn-secondary" v-on:click="cancel">Anuluj</button>
</div>
</div>
</template>
<script>
    export default {
        data: function() {
            return {
                editing: false,
                product: {}
            }
        },
        methods: {
            save() {},
            cancel() {}
        }
    }
</script>

```

Zapisz zmiany i wykonaj polecenie z listingu 20.5 w katalogu *productapp*, aby uruchomić narzędzia deweloperskie Vue.js.

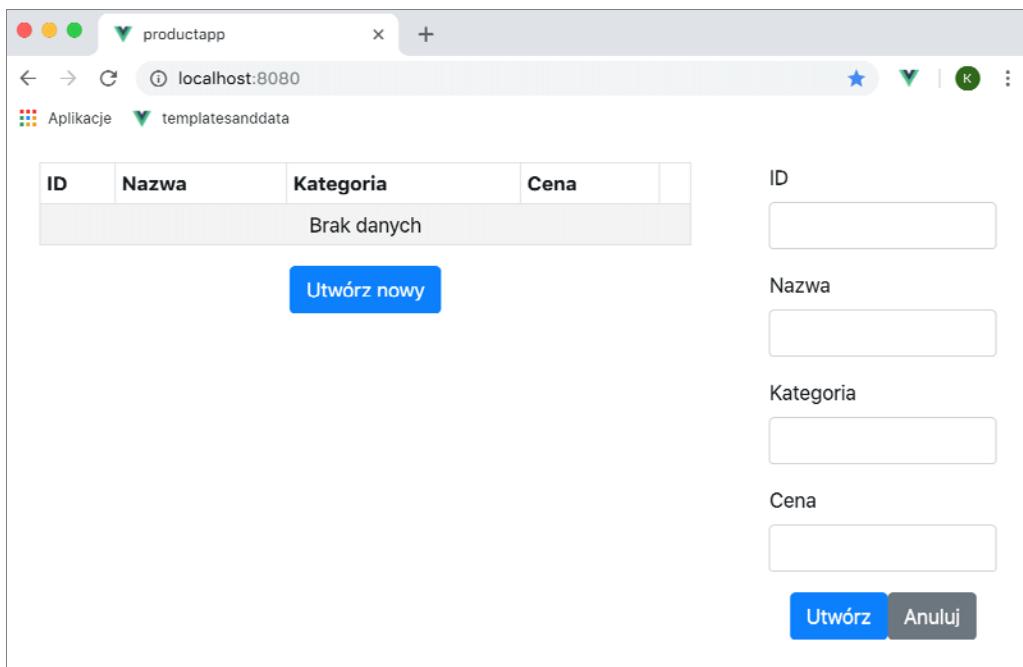
Listing 20.5. Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Po zakończeniu procesu inicjalizacji otwórz nowe okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć przykładową aplikację (rysunek 20.1).

Tworzenie i używanie magazynu danych

Proces rozpoczęcia pracy z współdzielonym stanem aplikacji składa się z kilku czynności, jednak gdy uporamy się z początkową konfiguracją, późniejsze rozszerzanie aplikacji będzie już znacznie łatwiejsze i szybsze. Zgodnie z przyjętą praktyką zaczniemy od utworzenia katalogu o nazwie *store*. W katalogu *src/store* umieszczać plik *index.js* o treści z listingu 20.6.



Rysunek 20.1. Przykładowa aplikacja po uruchomieniu

Listing 20.6. Zawartość pliku src/store/index.js

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
export default new Vuex.Store({
  state: {
    products: [
      {
        id: 1,
        name: "Produkt #1",
        category: "Test",
        price: 100
      },
      {
        id: 2,
        name: "Produkt #2",
        category: "Test",
        price: 150
      },
      {
        id: 3,
        name: "Produkt #3",
        category: "Test",
        price: 200
      }
    ],
    mutations: {
      saveProduct(currentState, product) {
        let index = currentState.products.findIndex(p => p.id == product.id);
```

```
        if (index == -1) {
            currentState.products.push(product);
        } else {
            Vue.set(currentState.products, index, product);
        }
    },
deleteProduct(currentState, product) {
    let index = currentState.products.findIndex(p => p.id == product.id);
    currentState.products.splice(index, 1);
}
})
```

W ten sposób uzyskaliśmy podstawowy magazyn danych Vuex. Ustalenie podstawowej struktury jest niezwykle ważne, dlatego przejdę przez listing 20.6 krok po kroku. Pierwsze dwie instrukcje importują Vue.js i Vuex:

```
...  
import Vue from "vue";  
import Vuex from "vuex";  
...
```

Kolejna instrukcja włącza Vuex:

• • •
Vue
• • •

Vuex jest dostarczany w postaci wtyczki Vue.js, która pozwala na rozszerzanie standardowych funkcji Vue.js (por. rozdział 26.). Aby zainstalować wtyczkę, należy skorzystać z metody `Vue.use`. Kolejna instrukcja tworzy magazyn danych i definiuje go jako domyślnie eksportowany element z modułu JavaScript:

```
...  
export default new Vuex.Store({  
  ...  
})
```

Słowo kluczowe `new` jest używane do utworzenia nowego obiektu, `Vuex.Store`, który przyjmuje argument w postaci obiektu konfiguracji.

Omówienie podziału na stan i mutacje

Jednym z największych wyzwań w pracy z magazynami danych jest fakt, że dane są przeznaczone tylko do odczytu. Aby wprowadzić jakiekolwiek zmiany, konieczne jest skorzystanie z oddzielnych funkcji, nazywanych **mutacjami** (ang. *mutations*). W momencie tworzenia magazynu danych dane aplikacji są definiowane za pomocą właściwości state. Zakres zmian, które mogą być wprowadzone w ramach tych danych, jest określany przy użyciu właściwości mutations. W przypadku magazynu danych zdefiniowanego w listingu 20.6 występuje tylko jeden element.

```
...  
state: {  
    products: [{  
        id: 1,  
        name: "Produkt #1",  
        category: "Test",  
        price: 100  
    },  
    {  
        id: 2,
```

```

        name: "Produkt #2",
        category: "Test",
        price: 150
    },
    {
        id: 3,
        name: "Produkt #3",
        category: "Test",
        price: 200
    }
],
},
...

```

Właściwość state jest używana do zdefiniowania właściwości products, która z kolei otrzymuje tablicę obiektów. Magazyn danych z listingu 20.6 definiuje także dwie mutacje.

```

...
mutations: {
    saveProduct(currentState, product) {
        let index = currentState.products.findIndex(p => p.id == product.id);
        if (index == -1) {
            currentState.products.push(product);
        } else {
            Vue.set(currentState.products, index, product);
        }
    },
    deleteProduct(currentState, product) {
        let index = currentState.products.findIndex(p => p.id == product.id);
        currentState.products.splice(index, 1);
    }
}
...

```

Mutacje to funkcje, które otrzymują aktualny stan magazynu danych, a także opcjonalny argument określający kontekst wprowadzenia zmian. W tym przykładzie mutacja saveProduct otrzymuje obiekt i dodaje go do tablicy products lub zamienia istniejący obiekt w tablicy. Mutacja deleteProduct otrzymuje obiekt i korzysta z jego wartości id, aby usunąć odpowiedni obiekt z tablicy products.

■ **Wskazówka** Zwróć uwagę, że skorzystałem z metody `Vue.set`, aby zamienić element w tablicy w mutacji `saveProduct` w listingu 20.6. Magazyny danych zmagają się z tymi samymi problemami dotyczącymi wykrywania zmian, co reszta mechanizmów aplikacji Vue.js opisanych w rozdziale 13.

Mutacje mają dostęp do bieżącego stanu magazynu danych dzięki pierwszemu argumentowi:

```

...
saveProduct(currentState, product) {
    let index = currentState.products.findIndex(p => p.id == product.id);
    if (index == -1) {
        currentState.products.push(product);
    } else {
        Vue.set(currentState.products, index, product);
    }
},
...

```

Nie musisz korzystać ze słowa kluczowego `this` w mutacjach — dostęp do danych jest możliwy tylko za pomocą pierwszego argumentu. Większość programistów jest przyzwyczajona do swobodnego dostępu do danych, dlatego może wydawać się dziwne definiowanie specjalnych funkcji przeznaczonych tylko w tym celu. Z czasem zauważysz, że takie podejście ma też wiele korzyści, zwłaszcza w dużych i złożonych aplikacjach.

Udostępnianie magazynu danych Vuex

Po utworzeniu magazynu danych musimy udostępnić go komponentom aplikacji. W tym celu należy wprowadzić kilka zmian w pliku `main.js` (listing 20.7).

Listing 20.7. Konfiguracja magazynu danych Vuex w pliku src/main.js

```
import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
import {
    RestDataSource
} from "./restDataSource";
import store from "./store";

Vue.config.productionTip = false
new Vue({
    render: h => h(App),
    data: {
        eventBus: new Vue()
    },
    store,
    provide: function() {
        return {
            eventBus: this.eventBus,
            restDataSource: new RestDataSource(this.eventBus)
        }
    }
}).$mount('#app')
```

Instrukcja `import` jest używana do importowania modułu z katalogu `store` (co doprowadzi do automatycznego wczytania pliku `index.js`) i przypisania nazwy `store`, która w konsekwencji służy do zdefiniowania właściwości w konfiguracji obiektu `Vue`. W ten sposób obiekt magazynu Vuex utworzony w pliku `index.js` udostępniamy wszystkim komponentom za pomocą specjalnej zmiennej `$store`, co pokazuję już w kolejnym punkcie.

Stosowanie magazynu danych

Podział na dane i mutacje może wydawać się dziwny w momencie utworzenia magazynu danych, ale z pewnością pasuje do struktury komponentów. W listingu 20.8 aktualizuję komponent `ProductDisplay`, dzięki czemu komponent wczytuje dane z magazynu danych i korzysta z mutacji `deleteProduct` w momencie kliknięcia przycisku `Usuń`.

Listing 20.8. Zastosowanie magazynu danych w pliku src/components/ProductDisplay.vue

```
...
<script>
export default {
    // data: function () {
    //     return {

```

```
//      products: []
// }
// },
computed: {
  products() {
    return this.$store.state.products;
  }
},
methods: {
  createNew() {},
  editProduct(product) {},
  deleteProduct(product) {
    this.$store.commit("deleteProduct", product);
  }
}
}
</script>
...

```

Dostęp do danych w magazynie jest uzyskiwany za pomocą właściwości `$store`, tworzonej przez dodanie magazynu do obiektu konfiguracji Vue, pokazanego w listingu 20.7. Wartości magazynu danych są dostępne tylko do odczytu, dlatego możemy skorzystać z nich we właściwościach obliczanych. W tym przykładzie zamieniłem właściwość danych `products` na właściwość obliczaną o takiej samej nazwie, która zwraca dane z magazynu.

```
...
products() {
  return this.$store.state.products;
}
...

```

Właściwość `$store` zwraca magazyn danych, a właściwość `state` daje dostęp do pojedynczych właściwości danych, takich jak właściwość `products` użyta w tym przykładzie.

Stosowanie trybu ścisłego w celu uniknięcia bezpośrednich zmian stanu

Vuex domyślnie nie wymusza oddzielenia stanu od operacji, co oznacza, że komponenty mogą modyfikować właściwości bezpośrednio w sekcji `state` magazynu. Łatwo jest zapomnieć, że zmiany nie powinny być dokonywane bezpośrednio, z czego wynika, że wszelkie narzędzia deweloperskie, takie jak Vue Devtools, nie będą w stanie wykrywać zmian.

W związku z tym Vuex dostarcza właściwość `strict`, która monitoruje stan właściwości magazynu i rzuca wyjątek, jeśli dokonano bezpośredniej zmiany. Aby włączyć tę funkcję, należy dodać właściwość `strict` do obiektu konfiguracji zastosowanego do utworzenia magazynu:

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
export default new Vuex.Store({
  strict: true,
  state: {
    products: [
      {
        id: 1,
        name: "Produkt #1",
        category: "Test",
        price: 100
      },
    ],
  }
});
```

```

    {
      id: 2,
      name: "Produkt #2",
      category: "Test",
      price: 150
    },
    {
      id: 3,
      name: "Produkt #3",
      category: "Test",
      price: 200
    }
  ]
},
mutations: {
  saveProduct(currentState, product) {
    let index = currentState.products
      .findIndex(p => p.id == product.id);
    if (index == -1) {
      currentState.products.push(product);
    } else {
      Vue.set(currentState.products, index, product);
    }
  },
  deleteProduct(currentState, product) {
    let index = currentState.products
      .findIndex(p => p.id == product.id);
    currentState.products.splice(index, 1);
  }
}
})
)
```

```

To ustawienie powinno być używane tylko w trakcie tworzenia aplikacji, ponieważ stosuje ono dość kosztowne operacje, które w środowisku produkcyjnym mogą zauważalnie wpływać na wydajność aplikacji.

---

Zastosowanie mutacji w magazynie danych odbywa się za pomocą metody `$store.commit` i jest to proces podobny do wyzwalania zdarzenia. Pierwszym argumentem metody `commit` jest nazwa mutacji do użycia, wyrażona w postaci tekstu, po której następuje opcjonalny „ładunek” — argument dający kontekst mutacji. W tym przykładzie dodałem ciało do metody `deleteProduct` komponentu, która jest wyzwalana w momencie kliknięcia przycisku *Usuń*. W metodzie wykonuję mutację o tej samej nazwie.

```

...
deleteProduct(product) {
 this.$store.commit("deleteProduct", product);
}
...

```

Skoro wiesz już, jak wartości są odczytywane i modyfikowane, możemy zintegrować magazyn danych z resztą aplikacji. W listingu 20.9 aktualizuję komponent `ProductEditor`, dzięki czemu jest w stanie modyfikować magazyn danych w celu tworzenia nowych produktów.

*Listing 20.9. Zastosowanie magazynu danych w pliku src/components/ProductEditor.vue*

```

...
<script>
export default {
 data: function() {
 return {

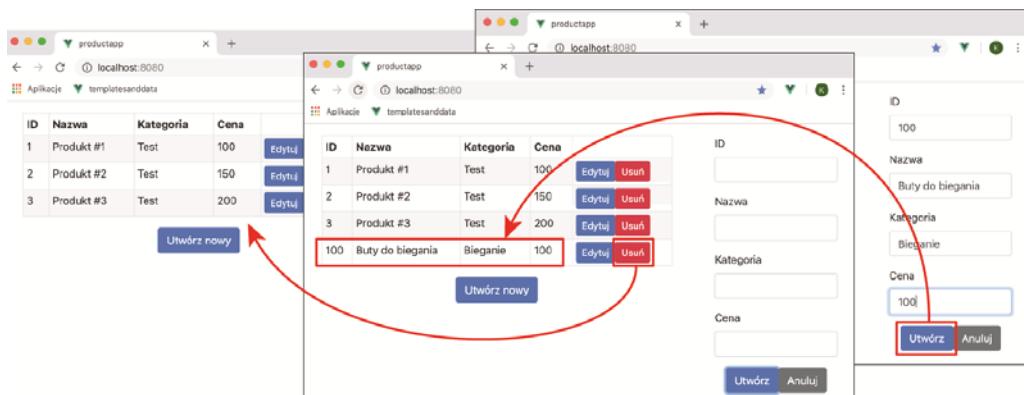
```

```

 editing: false,
 product: {}
 },
 methods: {
 save() {
 this.$store.commit("saveProduct", this.product);
 this.product = {};
 },
 cancel() {}
 }
}
</script>
...

```

Metoda `save` została zmodyfikowana, aby była w stanie zaktualizować mutację `addProduct` magazynu danych. Mutacja doda nowy produkt do tablicy `products` w sekcji `state` magazynu danych. Dzięki zmianom użytkownik może wypełnić pola formularza i kliknąć przycisk *Utwórz*, aby dodać nowy produkt, lub kliknąć przycisk *Usuń*, aby usunąć któryś z istniejących (rysunek 20.2).



Rysunek 20.2. Zastosowanie magazynu danych

## Dostęp do magazynu danych poza komponentami

Magazyn danych jest dostępny za pomocą właściwości `$store` w dowolnym komponencie aplikacji. Jeśli chcesz uzyskać dostęp do magazynu danych w części aplikacji, która nie jest komponentem, musisz skorzystać z instrukcji `import`:

```

...
import dataStore from "../store";
...

```

Ta instrukcja jest użyta w przykładzie z rozdziału 24., w którym korzystam z magazynu danych w pliku niebędącym komponentem. Przypisuje ona magazyn danych do nazwy `dataStore`. Wyrażenie, które występuje po słowie kluczowym `from`, stanowi ścieżkę do katalogu `store`, zawierającego plik `index.js`. Po skutecznym zimportowaniu pliku możesz odwoływać się do magazynu danych np. tak:

```

...
dataStore.commit("setComponentLoading", true);
...

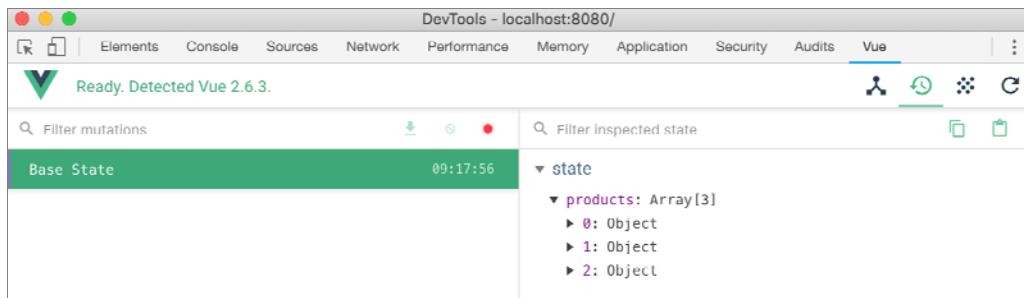
```

To kolejna instrukcja z rozdziału 24., w którym dowiesz się więcej na ten temat.

## Analiza zmian w magazynie danych

Główną zaletą stosowania magazynu danych jest oddzielenie stanu aplikacji od komponentów, co pozwala uprościć ich strukturę, umożliwia łatwą interakcję między nimi i powoduje, że cały projekt staje się łatwiejszy do zrozumienia i testowania.

Wykonywanie zmian tylko za pomocą mutacji pozwala na śledzenie stanu magazynu danych. Wtyczka Vue Devtools ma wbudowane wsparcie dla Vuex. Uruchom narzędzia przeglądarki F12 i przejdź na zakładkę *Vue*, a zobaczysz przycisk z ikonką zegarka, który pokaże dane w magazynie Vuex w momencie kliknięcia tegoż przycisku (rysunek 20.3).



Rysunek 20.3. Analiza magazynu danych

Jak widać, sekcja *state* zawiera właściwość *products*, której wartość stanowi tablica trzech obiektów. Zaproznaj się z treścią każdego z tych obiektów i zobacz, jakie właściwości są w nich zdefiniowane.

Okno F12 pozostaw otwarte. Przejdz do głównego okna przeglądarki i utwórz trzy nowe elementy, wprowadzając informacje z tabeli 20.3. Po wprowadzeniu szczegółów każdego produktu kliknij przycisk *Utwórz*, co spowoduje dodanie każdego z nich do tabeli wyświetlonej przez komponent *ProductDisplay*.

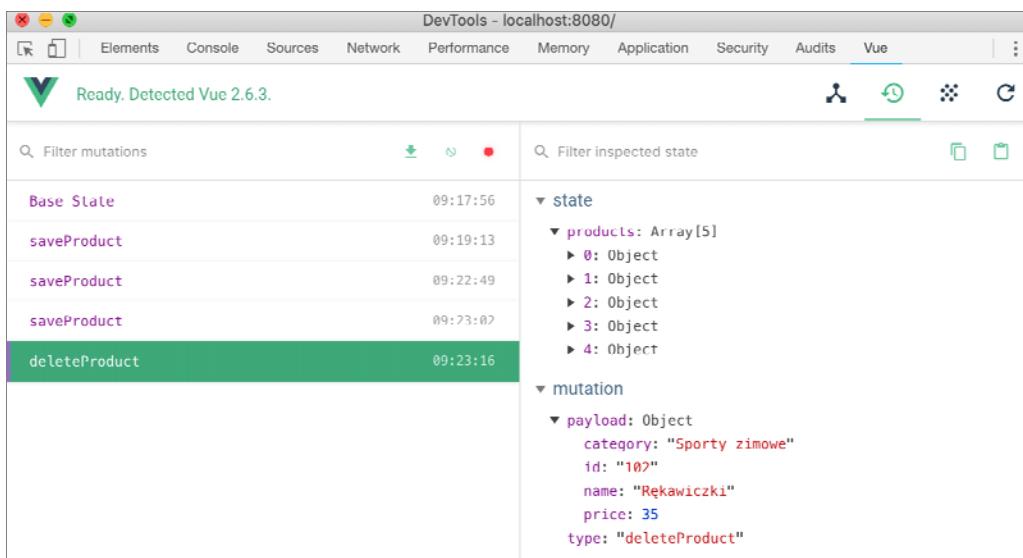
Tabela 20.3. Produkty do zastosowania w procesie testów magazynu Vuex

| ID  | Nazwa              | Kategoria     | Cena |
|-----|--------------------|---------------|------|
| 100 | Buty do biegania   | Bieganie      | 100  |
| 101 | Deska snowboardowa | Sporty zimowe | 500  |
| 102 | Rękawiczki         | Sporty zimowe | 35   |

Po utworzeniu tych produktów kliknij przycisk *Usuń* przy produkcie *Rękawiczki*, dzięki czemu usuniesz go z listy.

Przy wprowadzaniu każdej zmiany w zakładce Vue Devtools zobaczysz efekt każdej mutacji zastosowanej wobec danych wraz ze zrzutem danych z magazynu i szczegółami na temat ładunku dostarczonego przy każdej zmianie (rysunek 20.4).

Każda zmiana może zostać wycofana, dzięki czemu magazyn danych zostanie przywrócony do poprzedniego stanu. Cały stan może zostać wyeksportowany, a potem zimportowany ponownie. Możliwość analizy każdej zmiany w magazynie i efektów tych zmian jest niezwykle przydatna, ponieważ upraszcza debugowanie skomplikowanych problemów. To wszystko wymaga rzeczy jasna zastosowania wyłącznie mutacji do zmiany stanu — wszelkie zmiany dokonane bezpośrednio na właściwościach nie będą wykryte (patrz ramka „*Stosowanie trybu ścisłego w celu uniknięcia bezpośrednich zmian stanu*”).



Rysunek 20.4. Zmieniający się stan magazynu danych

## Definiowanie właściwości obliczanych w magazynie danych

Gdy przeniesiesz dane aplikacji do magazynu, z pewnością zauważysz, że wiele komponentów musi wykonywać te same operacje na stanie, aby pobrać wymagane wartości. Zamiast powielać ten sam kod w różnych komponentach, możesz skorzystać z funkcji getterów, które stanowią odpowiednik właściwości obliczanych w komponencie. W listingu 20.10 dodaję dwa gettery do magazynu danych, aby przekształcić dane produktu.

*Listing 20.10. Dodawanie getterów w pliku src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
Vue.use(Vuex);
export default new Vuex.Store({
 state: {
 products: [
 {
 id: 1,
 name: "Produkt #1",
 category: "Test",
 price: 100
 },
 {
 id: 2,
 name: "Produkt #2",
 category: "Test",
 price: 150
 },
 {
 id: 3,
 name: "Produkt #3",
 category: "Test",
 price: 200
 }
]
 }
})
```

```
 category: "Test",
 price: 200
 }
]
},
mutations: {
 saveProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 if (index == -1) {
 currentState.products.push(product);
 } else {
 Vue.set(currentState.products, index, product);
 }
 },
 deleteProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 currentState.products.splice(index, 1);
 }
},
getters: {
 orderedProducts(state) {
 return state.products.concat().sort((p1, p2) => p2.price - p1.price);
 },
 filteredProducts(state, getters) {
 return getters.orderedProducts.filter(p => p.price > 100);
 }
}
})
```

Gettery są dodawane do właściwości getters obiektu konfiguracji magazynu. Każdy getter to funkcja, która otrzymuje obiekt zawierający stan magazynu danych, a wynikiem jest obliczona wartość. Pierwszy zdefiniowany przez mnie getter sortuje dane produktów, uzyskując dostęp do tabeli produktów za pomocą parametru state.

```
...
orderedProducts(state) {
 return state.products.concat().sort((p1, p2) => p2.price - p1.price);
},
```

Jeden getter może korzystać z drugiego, definiując drugi parametr, dzięki któremu otrzyma dostęp do pozostałych getterów. Drugi zdefiniowany przeze mnie getter pobiera produkty posortowane przez getter orderedProducts, a następnie pozostawia tylko te, których cena jest większa od 100.

```
...
filteredProducts(state, getters) {
 return getters.orderedProducts.filter(p => p.price > 100);
}
...
```

Możliwość składania getterów w celu uzyskania bardziej skomplikowanych wyników to użyteczna funkcja, która zmniejsza duplikację kodu.

- **Ostrzeżenie** Gettery nie powinny wprowadzać zmian w magazynie danych. Z tego względu skorzystałem z metody concat w getterze orderedProducts, ponieważ metoda sort sortuje obiekty w tablicy w miejscu. Metoda concat generuje nową tablicę i zapewnia, że odczyt wartości gettera nie spowoduje zmiany stanu magazynu danych.

## Stosowanie gettera w komponencie

Dostęp do gettera jest możliwy za pomocą właściwości `this.$store.getters`, połączonej z nazwą gettera. Aby skorzystać z gettera `myGetter`, wystarczy podać wyrażenie `this.$store.getters.myGetter`. W listingu 20.11 zmieniam właściwość obliczoną zdefiniowaną w komponencie `ProductDisplay`, dzięki czemu pobieram z gettera `filteredProducts`.

*Listing 20.11. Zastosowanie gettera w pliku src/components/ProductDisplay.vue*

```
...
<script>
export default {
 computed: {
 products() {
 return this.$store.getters.filteredProducts;
 }
 },
 methods: {
 createNew() {},
 editProduct(product) {},
 deleteProduct(product) {
 this.$store.commit("deleteProduct", product);
 }
 }
}
</script>
...
```

Wiązanie `v-for` w szablonie komponentu odczytuje wartość właściwości obliczanej `products`, która pobiera wartość z gettera w magazynie danych. Wynikiem jest przefiltrowany i uporządkowany zbiór produktów (rysunek 20.5).

| ID | Nazwa      | Kategoria | Cena |                                               |
|----|------------|-----------|------|-----------------------------------------------|
| 3  | Produkt #3 | Test      | 200  | <button>Edytuj</button> <button>Usuń</button> |
| 2  | Produkt #2 | Test      | 150  | <button>Edytuj</button> <button>Usuń</button> |

**Utwórz nowy**

ID

Nazwa

Kategoria

Cena

**Utwórz** **Anuluj**

*Rysunek 20.5. Zastosowanie gettera z magazynu danych*

## Przekazywanie argumentów do getterów

Gettery mogą przyjmować również dodatkowe argumenty, dostarczone przez komponent. Dzięki nim można wpłynąć na wartość generowaną przez dany getter. W listingu 20.12 zmieniam getter `filteredProducts`, dzięki czemu wartość użyta w trakcie filtrowania produktów może być przekazana w formie argumentu.

*Listing 20.12. Zastosowanie argumentu gettera w pliku src/store/index.js*

```
...
getters: {
 orderedProducts(state) {
 return state.products.concat().sort((p1, p2) => p2.price - p1.price);
 },
 filteredProducts(state, getters) {
 return (amount) => getters.orderedProducts.filter(p => p.price > amount);
 }
}
...
```

Składnia może wydawać się dziwna, ale otrzymanie argumentu przez getter wymaga, aby getter zwrócił funkcję. Funkcja ta zostanie wywołana w momencie odczytu gettera i przekazania argumentu przez komponent. W tym przykładzie wynikiem gettera jest funkcja, która przyjmuje argument `amount`, przydatny do wykonania operacji filtrowania. W listingu 20.13 zmieniam właściwość obliczaną w komponencie `ProductDisplay`, aby przekazać wartość do gettera.

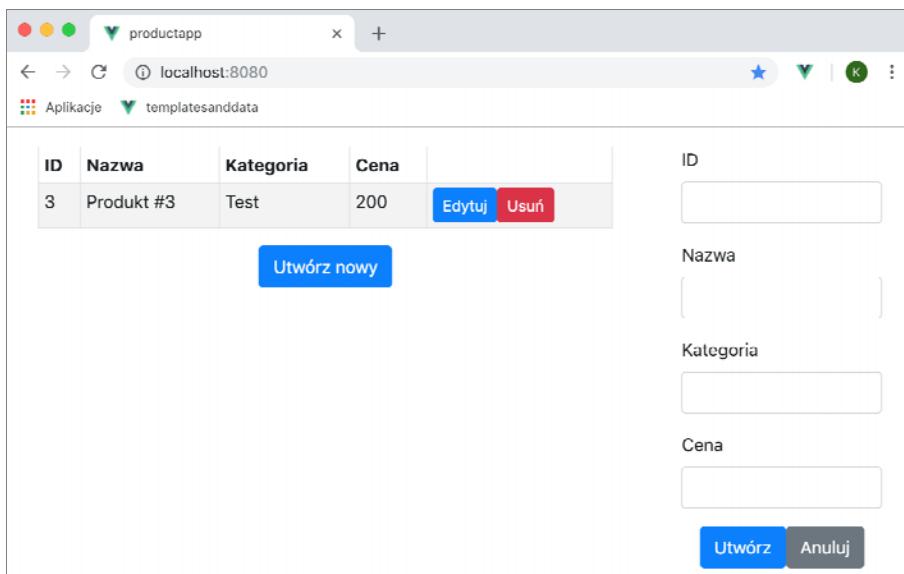
*Listing 20.13. Zastosowanie argumentu gettera w pliku src/components/ProductDisplay.vue*

```
...
<script>
export default {
 computed: {
 products() {
 return this.$store.getters.filteredProducts(175);
 }
 },
 methods: {
 createNew() {},
 editProduct(product) {},
 deleteProduct(product) {
 this.$store.commit("deleteProduct", product);
 }
 }
}
</script>
...
```

W wyniku działania kodu tylko jeden obiekt będzie wyświetlany w tabeli, ponieważ tylko on spełnia warunek w postaci wartości podanej w komponencie (rysunek 20.6).

## Wykonywanie operacji asynchronicznych

Mutacje wykonują operacje synchroniczne, co oznacza, że świetnie radzą sobie w pracy z danymi lokalnymi. Z REST-ową usługą sieciową — już niekoniecznie. W przypadku zadań asynchronicznych Vuex dostarcza **mechanizm akcji** (ang. *actions*), który pozwala na wykonanie asynchronicznych zadań. Wyniki tych zadań są umieszczane w magazynie danych za pomocą mutacji. W listingu 20.14 do magazynu danych dodaję akcje, które konsumują REST-ową usługę sieciową.



Rysunek 20.6. Zastosowanie argumentu gettera

*Listing 20.14. Dodawanie akcji w pliku src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
 state: {
 products: []
 },
 mutations: {
 saveProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 if (index == -1) {
 currentState.products.push(product);
 } else {
 Vue.set(currentState.products, index, product);
 }
 },
 deleteProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 currentState.products.splice(index, 1);
 }
 },
 getters: {
 orderedProducts(state) {
 return state.products.concat().sort((p1, p2) => p2.price - p1.price);
 },
 filteredProducts(state, getters) {
 return (amount) => getters.orderedProducts.filter(p => p.price > amount);
 }
 },
 actions: {
```

```

 async getProductsAction(context) {
 (await Axios.get(baseUrl)).data
 .forEach(p => context.commit("saveProduct", p));
 },
 async saveProductAction(context, product) {
 let index = context.state.products.findIndex(p => p.id == product.id);
 if (index == -1) {
 await Axios.post(baseUrl, product);
 } else {
 await Axios.put(`${baseUrl}${product.id}`, product);
 }
 context.commit("saveProduct", product);
 },
 async deleteProductAction(context, product) {
 await Axios.delete(`${baseUrl}${product.id}`);
 context.commit("deleteProduct", product);
 }
 }
})

```

Akcje definiuje się przez dodanie właściwości actions do obiektu konfiguracji magazynu. Każda akcja to funkcja, która otrzymuje obiekt kontekstu, dający dostęp do stanu, getterów i mutacji. Akcje nie mogą zmieniać stanu danych bezpośrednio — konieczna jest praca za pośrednictwem mutacji za pomocą metody commit. W listingu usuwam testowe dane z tablicy products, a także definiuję trzy akcje, które korzystają z Axiosa w celu skomunikowania się z REST-ową usługą sieciową i aktualizacji danych za pomocą mutacji.

- **Wskazówka** Nie musisz uwzględniać słowa Action w nazwach swoich akcji. Stosuję taką konwencję, aby rozróżnić akcje i mutacje, niemniej jest to mój osobisty wybór, a nie odgórny przymus.

W listingu 20.15 aktualizuję komponent ProductDisplay w celu dodania obsługi akcji magazynu związanych z pobraniem początkowych danych z usługi sieciowej w momencie kliknięcia przycisku *Usuń*.

*Listing 20.15. Zastosowanie akcji w pliku src/components/ProductDisplay.vue*

```

...
<script>
export default {
 computed: {
 products() {
 return this.$store.state.products;
 }
 },
 methods: {
 createNew() {},
 editProduct(product) {},
 deleteProduct(product) {
 this.$store.dispatch("deleteProductAction", product);
 }
 },
 created() {
 this.$store.dispatch("getProductsAction");
 }
}
</script>
...

```

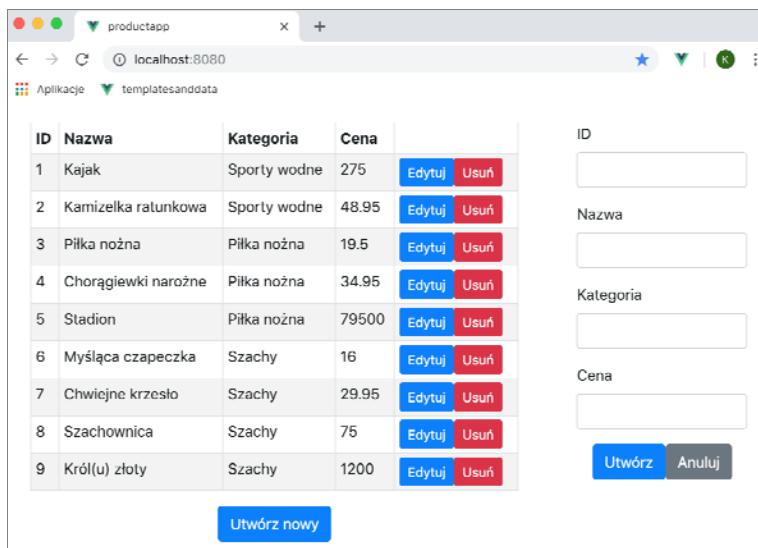
Akcje są wykonywane za pomocą metody `dispatch` przyjmującej nazwę argumentu, którego chcemy użyć, a także opcjonalny argument, który zostanie przekazany do akcji, podobnie do sposobu użycia mutacji. Nie ma możliwości wypełnienia magazynu Vuex w momencie jego utworzenia, dlatego korzystam z metody `created` komponentu opisanej w rozdziale 17., aby wywołać akcję `getProductsAction` i pobrać początkowe dane z usługi sieciowej (zmieniłem także dane zastosowane we właściwości obliczanej `products`, dzięki czemu są one pobierane ze stanu, a nie getterów, co zapewnia, że wyświetlane zostaną wszystkie dane pobrane z serwera).

W listingu 20.16 modyfikuję komponent `ProductEditor`, dodając akcje do przechowywania i modyfikacji danych w usłudze sieciowej.

**Listing 20.16.** Zastosowanie akcji w pliku `src/components/ProductEditor.vue`

```
...
<script>
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 save() {
 this.$store.dispatch("saveProductAction", this.product);
 this.product = {};
 },
 cancel() {}
 }
}
</script>
...
```

W tym momencie dane są pobierane z usługi sieciowej (rysunek 20.7), a dodawanie lub usuwanie obiektów doprowadzi do analogicznej zmiany po stronie serwera.



The screenshot shows a web application interface. On the left, there is a table with columns: ID, Nazwa, Kategoria, and Cena. The table contains 9 rows of data. On the right, there is a form with fields for ID, Nazwa, Kategoria, and Cena, each with an input field. Below the form are two buttons: 'Utwórz' (Create) and 'Anuluj' (Cancel). At the bottom left of the main area is a blue button labeled 'Utwórz nowy' (Create new).

| ID | Nazwa               | Kategoria    | Cena  |                                               |
|----|---------------------|--------------|-------|-----------------------------------------------|
| 1  | Kajak               | Sporty wodne | 275   | <button>Edytuj</button> <button>Usuń</button> |
| 2  | Kamizelka ratunkowa | Sporty wodne | 48.95 | <button>Edytuj</button> <button>Usuń</button> |
| 3  | Piłka nożna         | Piłka nożna  | 19.5  | <button>Edytuj</button> <button>Usuń</button> |
| 4  | Chorągiewki narożne | Piłka nożna  | 34.95 | <button>Edytuj</button> <button>Usuń</button> |
| 5  | Stadion             | Piłka nożna  | 79500 | <button>Edytuj</button> <button>Usuń</button> |
| 6  | Myśląca czapeczka   | Szachy       | 16    | <button>Edytuj</button> <button>Usuń</button> |
| 7  | Chwiczące krzesło   | Szachy       | 29.95 | <button>Edytuj</button> <button>Usuń</button> |
| 8  | Szachownica         | Szachy       | 75    | <button>Edytuj</button> <button>Usuń</button> |
| 9  | Król(u) złoty       | Szachy       | 1200  | <button>Edytuj</button> <button>Usuń</button> |

**Rysunek 20.7.** Zastosowanie akcji magazynu danych

## Otrzymywanie powiadomień o zmianach

Magazyn danych może być używany do zarządzania całym stanem aplikacji, a nie tylko danymi wyświetlanymi użytkownikowi. W przykładowej aplikacji użytkownik wybiera produkt do edycji przez kliknięcie przycisku wyświetlanego przez komponent `ProductDisplay`. Funkcje związane z edycją dostarcza za to komponent `ProductEditor`. Aby umożliwić tego typu zgranie pomiędzy komponentami, Vuex dostarcza obserwatorów, którzy wypowiadają powiadomienia o zmianie wartości danych, odpowiadających obserwatorom działającym w komponentach Vue.js (por. rozdział 17.). W listingu 20.17 do magazynu danych dodaję właściwość stanu, która wskazuje produkt wybrany przez użytkownika, wraz z mutacją, która ustawia jego wartość.

**Listing 20.17.** Dodawanie właściwości stanu i mutacji w pliku `src/store/index.js`

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
 state: {
 products: [],
 selectedProduct: null
 },
 mutations: {
 saveProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 if (index == -1) {
 currentState.products.push(product);
 } else {
 Vue.set(currentState.products, index, product);
 }
 },
 deleteProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 currentState.products.splice(index, 1);
 },
 selectProduct(currentState, product) {
 currentState.selectedProduct = product;
 }
 },
 getters: {
 // ...gettery zostały pominięte...
 },
 actions: {
 // ...akcje zostały pominięte...
 }
})
```

W listingu 20.18 zaktualizowałem komponent `ProductDisplay` tak, aby wywoływał mutację po kliknięciu przycisku *Utwórz nowy* lub jednego z przycisków *Edytuj*.

**Listing 20.18.** Zastosowanie mutacji w pliku `src/components/ProductDisplay.vue`

```
...
<script>
export default {
 computed: {
 products() {
```

```

 return this.$store.state.products;
 }
},
methods: {
 createNew() {
 this.$store.commit("selectProduct");
 },
 editProduct(product) {
 this.$store.commit("selectProduct", product);
 },
 deleteProduct(product) {
 this.$store.dispatch("deleteProductAction", product);
 }
},
created() {
 this.$store.dispatch("getProductsAction");
}
}
</script>
...

```

Gdy użytkownik kliknie przycisk *Utwórz nowy*, zostanie wywołana metoda `createNew` komponentu. Doprowadzi to do wyzwolenia mutacji `selectProduct` bez żadnego argumentu i ustawienia właściwości stanu `selectedProduct` na null. Oznacza to, że użytkownik nie wybrał produktu do edycji. Gdy użytkownik kliknie jeden z przycisków *Edytuj*, zostanie wywołana metoda `editProduct`, która wzywala tę samą mutację, ale już z obiektem produktu, sygnalizując, że to właśnie ten obiekt użytkownik chce edytować.

Aby zareagować na sygnały wysyłane z komponentu `ProductDisplay` przy użyciu magazynu danych, dodaję obserwatora Vuex do komponentu `ProductEditor` (listing 20.19).

**Listing 20.19.** Obserwowanie wartości magazynu danych w pliku `src/components/ProductEditor.vue`

```

...
<script>
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 save() {
 this.$store.dispatch("saveProductAction", this.product);
 this.product = {};
 },
 cancel() {
 this.$store.commit("selectProduct");
 }
 },
 created() {
 this.$store.watch(state => state.selectedProduct,
 (newValue, oldValue) => {
 if (newValue == null) {
 this.editing = false;
 this.product = {};
 } else {
 this.editing = true;
 this.product = {};
 }
 }
)
 }
}
</script>

```

```

 Object.assign(this.product, newValue);
 });
}
</script>
...

```

Do utworzenia obserwatora wykorzystałem metodę cyklu życia komponentu created, opisaną w rozdziale 17. Osiągnąłem to przy użyciu metody watch, która jest dostępna za pomocą wyrażenia this.\$store.watch. Metoda watch przyjmuje dwie funkcje: pierwsza z nich jest używana do ustalenia wartości danych na watch, a druga jest wywoływana, gdy dojdzie do zmiany zaznaczonych danych, i korzysta z tego samego rodzaju funkcji co przy obserwatorach komponentów.

- **Wskazówka** Wynikiem wywołania metody watch jest funkcja, która może być wywołana w celu zatrzymania otrzymywania powiadomień. Przykład takiej funkcji znajdziesz w rozdziale 21.

W tym momencie komponent reaguje na sygnały wysypane za pomocą magazynu danych, zmieniając lokalny stan i kopując wartości z zaznaczonego produktu, o ile taki został wybrany. Aby sprawdzić zmiany, kliknij przycisk *Utwórz nowy* lub jeden z przycisków *Edytuj* i przekonaj się, jak zareagują pola formularza (rysunek 20.8).

The screenshot shows a web browser window with the title 'productapp' at 'localhost:8080'. The page displays a table of products:

| ID | Nazwa               | Kategoria    | Cena  |                                               |
|----|---------------------|--------------|-------|-----------------------------------------------|
| 1  | Kajak               | Sporty wodne | 275   | <button>Edytuj</button> <button>Usuń</button> |
| 2  | Kamizelka ratunkowa | Sporty wodne | 48.95 | <button>Edytuj</button> <button>Usuń</button> |
| 3  | Piłka nożna         | Piłka nożna  | 19.5  | <button>Edytuj</button> <button>Usuń</button> |
| 4  | Chorągiewki narożne | Piłka nożna  | 34.95 | <button>Edytuj</button> <button>Usuń</button> |
| 5  | Stadion             | Piłka nożna  | 79500 | <button>Edytuj</button> <button>Usuń</button> |
| 6  | Myśląca czapeczka   | Szachy       | 16    | <button>Edytuj</button> <button>Usuń</button> |
| 7  | Chwiejne krzesło    | Szachy       | 29.95 | <button>Edytuj</button> <button>Usuń</button> |
| 8  | Szachownica         | Szachy       | 75    | <button>Edytuj</button> <button>Usuń</button> |
| 9  | Król(u) złoty       | Szachy       | 1200  | <button>Edytuj</button> <button>Usuń</button> |

A modal dialog is open for the product with ID 5 ('Stadion'). The dialog contains fields for 'ID' (5), 'Nazwa' (Stadion), 'Kategoria' (Piłka nożna), and 'Cena' (79500). It includes two buttons at the bottom: 'Zapisz' (Save) and 'Anuluj' (Cancel). An arrow points from the 'Edytuj' button in the table row to the 'Edytuj' button in the dialog.

Rysunek 20.8. Zastosowanie obserwatora magazynu danych

## Omówienie dwukierunkowości i lokalnego stanu komponentu

W kodzie z listingu 20.19 pojawiła się ciekawa konstrukcja: korzystam z funkcji `Object.assign`, aby skopiować wartości z obiektu w magazynie danych do lokalnej właściwości `product`:

```
...
} else {
 this.editing = true;
 this.product = {};
 Object.assign(this.product, newValue);
}
...
```

Można by zapytać, dlaczego nie pracowałem bezpośrednio z właściwością stanu z magazynu danych. Przede wszystkim zastosowanie dyrektywy `v-bind` do utworzenia dwukierunkowego wiązania z wartościami danych Vuex jest dziwne. Można tak zrobić, ale wymaga to zastosowania oddzielnego wiązania danych i zdarzeń. Można też utworzyć właściwości obliczane, które mają zarówno gettery, jak i settery. Nie jest to świetne rozwiązanie, dlatego zdecydowanie wolę korzystać ze stanu lokalnego w komponencie, jak w tym przykładzie.

Poza tym bezpośrednia praca z obiektem z magazynu danych może wydawać się myląca dla użytkownika. Skoro obiekt będący w magazynie danych to ten sam obiekt, który jest wyświetlany w dyrektywie `v-for` w komponencie `ProductDisplay`, to wszelkie zmiany dokonane w elementach `input` będą trafiać natychmiast do tabeli. Jeśli użytkownik anuluje operację bez kliknięcia przycisku *Zapisz*, zmiany nie zostaną wysłane do usługi sieciowej, ale wciąż będą wyświetlane w aplikacji. Użytkownik może więc pomyśleć, że zmiany zostały zapisane, a jest to nieprawda.

Zaczynając pracę z Vuex, możesz ulec pokusie, aby wykonywać wszystko w aplikacji na podstawie danych w magazynie. Nie zawsze jest to jednak najwłaściwsze podejście — niektóre funkcje aplikacji najlepiej jest obsłużyć, łącząc możliwości magazynu danych z lokalnym stanem komponentu.

## Mapowanie funkcji magazynu danych w komponentach

Zastosowanie funkcji magazynu danych w komponentach prowadzi do powstania dużej ilości podobnego kodu, w którym właściwości obliczane służą do uzyskania dostępu do zmiennych stanu, a metody — do wyzwolenia mutacji i akcji. Vuex dostarcza szereg funkcji, które pozwalają na generowanie funkcji dających automatyczny dostęp do funkcji magazynu danych. Dzięki tym funkcjom jesteśmy w stanie uprościć komponenty. Tabela 20.4 opisuje funkcje mapowania.

Tabela 20.4. Funkcje mapujące mechanizmy Vuex

| Nazwa                     | Opis                                                                                                                      |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>mapState</code>     | Ta funkcja generuje właściwości obliczane, które dają dostęp do właściwości stanu ( <code>state</code> ) magazynu danych. |
| <code>mapGetters</code>   | Ta funkcja generuje właściwości obliczane, które dają dostęp do getterów magazynu danych.                                 |
| <code>mapMutations</code> | Ta funkcja generuje metody, które dają dostęp do mutacji magazynu danych.                                                 |
| <code>mapActions</code>   | Ta funkcja generuje metody, które dają dostęp do akcji magazynu danych.                                                   |

Opisane w tabeli 20.4 funkcje udostępniają mechanizmy z magazynu danych tuż obok lokalnych właściwości i metod komponentu. W listingu 20.20 korzystam z tych funkcji, aby udostępnić magazyn danych w komponencie ProductDisplay.

**Listing 20.20.** Generowanie elementów w komponencie w pliku *src/components/ProductDisplay.vue*

```
<template>
 <div>
 <table class="table table-sm table-striped table-bordered">
 <tr>
 <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
 </tr>
 <tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <button class="btn btn-sm btn-primary"
 v-on:click="editProduct(p)">
 Edytuj
 </button>
 <button class="btn btn-sm btn-danger"
 v-on:click="deleteProduct(p)">
 Usuń
 </button>
 </td>
 </tr>
 <tr v-if="products.length == 0">
 <td colspan="5" class="text-center">No Data</td>
 </tr>
 </tbody>
 </table>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="createNew()">
 Utwórz nowy
 </button>
 </div>
 </div>
</template>
<script>
 import {
 mapState,
 mapMutations,
 mapActions
 } from "vuex";
 export default {
 computed: {
 ...mapState(["products"])
 },
 methods: {
 ...mapMutations({
 editProduct: "selectProduct",
 createNew: "selectProduct",
 }),
 ...mapActions({
 getProducts: "getProductsAction",
 })
 }
 }
</script>
```

```

 deleteProduct: "deleteProductAction"
 })
},
created() {
 this.getProducts();
}
}
</script>

```

Funkcje opisane w tabeli 20.4 są importowane z modułu vuex i są wdrażane przy użyciu operatora rozwińienia (operator ...) w celu utworzenia właściwości i metod, które wiążą komponent z magazynem danych.

Istnieją dwie metody tworzenia mapowań z magazynem danych. Pierwsza z nich polega na przekazaniu tablicy tekstów do funkcji mapowania, która informuje Vuex o konieczności utworzenia właściwości lub metod o nazwie pasującej do tej użytej w magazynie danych:

```

...
computed: {
 ...mapState(["products"])
},
...

```

W ten sposób informujemy Vuex o konieczności utworzenia właściwości obliczanej o nazwie products, która zwróci wartość właściwości stanu magazynu danych o nazwie products. Jeśli nie chcesz używać nazw zdefiniowanych w magazynie danych, musisz przekazać obiekt wiążący nazwy użyte w magazynie danych z nazwami, pod którymi będą one znane w komponencie:

```

...
...mapActions({
 getProducts: "getProductsAction",
 deleteProduct: "deleteProductAction"
})
...

```

W tym przykładzie Vuex utworzy metody getProducts i deleteProduct, które będą wykonywać akcje magazynu danych o nazwach, odpowiednio, getProductsAction i deleteProductAction.

### Wykonywanie zmapowanych metod bez argumentów

Metody generowane przez funkcje mapujące w przypadku mutacji i akcji będą przyjmować argumenty, które następnie będą przekazywane do magazynu danych. Musisz uważać, jeśli chcesz wywołać taką metodę bezpośrednio z poziomu wiązania zdarzenia bez przekazania argumentu. W listingu 20.20 musiałem wprowadzić zmianę w wiązaniu zdarzenia dla przycisku *Utwórz nowy*.

```

...
<button class="btn btn-primary" v-on:click="createNew()">
...

```

Poprzednio wyrażenie dyrektywy v-on zawierało jedynie nazwę metody. W ten sposób wywołaliśmy metodę zmapowaną, utworzoną przez Vuex, przekazując obiekt zdarzenia dostarczony przez dyrektywę, który w konsekwencji trafił do zmapowanej mutacji. Przykładowa aplikacja wymaga, aby mutacja była wywoływana bez argumentu, dlatego dodałem pustą parę nawiasów do wyrażenia dyrektywy v-on w celu uniknięcia przekazania obiektu zdarzenia jako argumentu.

# Stosowanie modułów magazynu danych

W miarę wzrostu złożoności aplikacji wzrasta również ilość kodu i danych w magazynie. Vuex wspiera tworzenie modułów, co pozwala na podział magazynu danych na samowystarczalne moduły, które łatwiej jest tworzyć i zrozumieć, a także łatwo jest nimi zarządzać.

Każdy moduł jest zdefiniowany w odrębnym pliku. W ramach przykładu do katalogu *src/store* dodaję plik *preferences.js* o treści z listingu 20.21.

**Listing 20.21.** Zawartość pliku *src/store/preferences.js*

```
export default {
 state: {
 stripedTable: true,
 primaryEditButton: false,
 dangerDeleteButton: false
 },
 getters: {
 editClass(state) {
 return state.primaryEditButton ? "btn-primary" : "btn-secondary";
 },
 deleteClass(state) {
 return state.dangerDeleteButton ? "btn-danger" : "btn-secondary";
 },
 tableClass(state, payload, rootState) {
 return rootState.products.length > 0 &&
 rootState.products[0].price > 500 ? "table-striped" : ""
 }
 },
 mutations: {
 setEditButtonColor(currentState, primary) {
 currentState.primaryEditButton = primary;
 },
 setDeleteButtonColor(currentState, danger) {
 currentState.dangerDeleteButton = danger;
 }
 }
}
```

Definiując moduł, tworzysz obiekt języka JavaScript o takim samym stanie, getterach, mutacjach i akcjach, które poznaliśmy w tym rozdziale. Różnica polega na tym, że obiekt kontekstu przekazany do tych funkcji będzie zawierać jedynie dane stanu z tego modułu. Jeśli chcesz uzyskać dostęp do magazynu danych w głównym module, musisz podać dodatkowy argument:

```
...
tableClass(state, payload, rootState) {
 return rootState.products.length > 0
 && rootState.products[0].price > 500 ? "table-striped" : ""
}
...
```

Ten getter przyjmuje parametr *rootState*, który daje dostęp do właściwości *products*. Korzystamy z niego, użależniając wartość gettera od ceny pierwszego produktu w tablicy.

- **Wskazówka** Gdy korzystasz z głównego modułu magazynu danych, musisz zdefiniować argument ładunku (ang. *payload*), nawet jeśli go nie używasz (por. listing 20.21).

## Rejestrowanie i stosowanie modułu magazynu danych

Aby dołączyć moduł w magazynie danych, musisz skorzystać z instrukcji `import` i skonfigurować magazyn danych za pomocą właściwości `modules` w listingu 20.22.

*Listing 20.22. Rejestrowanie modułu w pliku src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import PrefsModule from "./preferences";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
 modules: {
 prefs: PrefsModule
 },
 state: {
 products: [],
 selectedProduct: null
 },
 //...pozostałe funkcje zostały pominięte...
})
```

Vuex łączy stan, gettery, mutacje i akcje zdefiniowane w module z tymi zdefiniowanymi w pliku `index.js`, co oznacza, że komponenty nie muszą martwić się o lokalizację poszczególnych elementów magazynu danych. W listingu 20.23 modyfikuję komponent `ProductDisplay`, dzięki czemu obsługuje on dane i mutacje zdefiniowane w module.

*Listing 20.23. Zastosowanie funkcji modułu w pliku src/components/ProductDisplay.vue*

```
<template>
<div>
 <table class="table table-sm table-bordered" v-bind:class="tableClass">
 <tr>
 <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena </th><th></th>
 </tr>
 <tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <button class="btn btn-sm"
 v-bind:class="editClass"
 v-on:click="editProduct(p)">
 Edytuj
 </button>
 <button class="btn btn-sm"
 v-bind:class="deleteClass"
 v-on:click="deleteProduct(p)">
 Usuń
 </button>
 </td>
 </tr>
 <tr v-if="products.length == 0">
 <td colspan="5" class="text-center">Brak danych</td>
 </tr>
 </tbody>
 </table>
</div>
```

```

 </tr>
 </tbody>
</table>
<div class="text-center">
 <button class="btn btn-primary" v-on:click="createNew()">
 Utwórz nowy
 </button>
</div>
</div>
</template>
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapGetters(["tableClass", "editClass", "deleteClass"])
 },
 methods: {
 ...mapMutations({
 editProduct: "selectProduct",
 createNew: "selectProduct",
 }),
 ...mapMutations(["setEditButtonColor", "setDeleteButtonColor"]),
 ...mapActions({
 getProducts: "getProductsAction",
 deleteProduct: "deleteProductAction"
 })
 },
 created() {
 this.getProducts();
 this.setEditButtonColor(false);
 this.setDeleteButtonColor(false);
 }
}
</script>

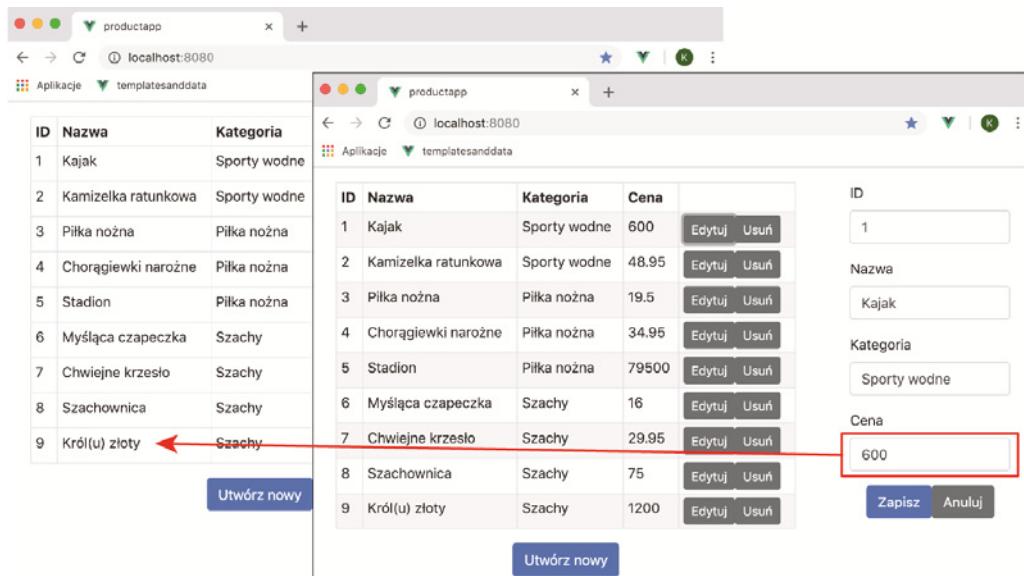
```

Skorzystałem z funkcji mapowania, aby powiązać gettery i mutacje dostarczone przez moduł magazynu danych z odpowiadającymi im wartościami w szablonie komponentu za pomocą dyrektywy `v-bind`. Wywołuję mutację w metodzie `created` komponentu. Jeśli zmienisz wartość właściwości `price` pierwszego elementu w tablicy na więcej niż 500, zobaczysz efekt działania gettera, który uzyskuje dostęp do stanu magazynu danych (rysunek 20.9).

Gdy wartość jest mniejsza niż 500, wiersze nie zostaną pokolorowane, w przeciwnym razie — już tak.

## Dostęp do stanu modułu

Jeśli chcesz uzyskać dostęp do właściwości stanu zdefiniowanych w module, musisz zastosować nieco inne podejście. Choć gettery, mutacje i akcje są integrowane z resztą magazynu danych bezproblemowo, stan jest trzymany oddzielnie. Dostęp do niego jest możliwy za pomocą nazwy przypisanej do modułu w momencie importowania (listing 20.24).



Rysunek 20.9. Zastosowanie mechanizmów dostarczonych w module magazynu danych

Listing 20.24. Dostęp do stanu modułu w pliku src/components/ProductDisplay.vue

```
<template>
<div>
 <table class="table table-sm table-bordered"
 v-bind:class="'table-striped' == useStripedTable">
 <tr>
 <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
 </tr>
 <tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <button class="btn btn-sm"
 v-bind:class="editClass"
 v-on:click="editProduct(p)">
 Edytuj
 </button>
 <button class="btn btn-sm"
 v-bind:class="deleteClass"
 v-on:click="deleteProduct(p)">
 Usuń
 </button>
 </td>
 </tr>
 <tr v-if="products.length == 0">
 <td colspan="5" class="text-center">Brak danych</td>
 </tr>
 </tbody>
 </table>
 <div class="text-center">
```

```

 <button class="btn btn-primary" v-on:click="createNew()">
 Utwórz nowy
 </button>
 </div>
</div>
</template>
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 useStripedTable: state => state.prefs.stripedTable
 }),
 ...mapGetters(["tableClass", "editClass", "deleteClass"])
 },
 methods: {
 ...mapMutations({
 editProduct: "selectProduct",
 createNew: "selectProduct",
 }),
 ...mapMutations(["setEditButtonColor", "setDeleteButtonColor"]),
 ...mapActions({
 getProducts: "getProductsAction",
 deleteProduct: "deleteProductAction"
 })
 },
 created() {
 this.getProducts();
 this.setEditButtonColor(false);
 this.setDeleteButtonColor(false);
 }
}
</script>

```

Metoda `mapState` otrzymuje obiekt, którego właściwości zostaną użyte jako nazwy lokalnych właściwości. Wartościami tych lokalnych właściwości będą funkcje, które otrzymują obiekt stanu i wybierają wymaganą wartość magazynu danych. W tym przykładzie korzystam z funkcji, która tworzy właściwość obliczaną o nazwie `useStripedTable`, powiązaną z właściwością stanu `stripedTable` zdefiniowaną w module `prefs`.

## **Stosowanie przestrzeni nazw modułów**

Domyślne zachowanie modułu polega na połączeniu getterów, mutacji i akcji z magazynem danych, dzięki czemu komponenty nie muszą znać struktury magazynu danych. Stan magazynu jest trzymany osobno, przez co wymagany jest dostęp za pomocą prefiku. Jeśli włączysz funkcję przestrzeni nazw (listing 20.25), to wszystkie rodzaje mechanizmów będą dostępne za pomocą prefiku.

**Listing 20.25.** Włączenie funkcji przestrzeni nazw w pliku `src/store/preferences.js`

```

export default {
 namespaced: true,
 state: {

```

```

 stripedTable: true,
 primaryEditButton: true,
 dangerDeleteButton: false
 },
 getters: {
 editClass(state) {
 return state.primaryEditButton ? "btn-primary" : "btn-secondary";
 },
 deleteClass(state) {
 return state.dangerDeleteButton ? "btn-danger" : "btn-secondary";
 },
 tableClass(state, payload, rootState) {
 return rootState.products.length > 0 &&
 rootState.products[0].price > 500 ? "table-striped" : ""
 }
 },
 mutations: {
 setEditButtonColor(currentState, primary) {
 currentState.primaryEditButton = primary;
 },
 setDeleteButtonColor(currentState, danger) {
 currentState.dangerDeleteButton = danger;
 }
 }
}
}

```

Powiązanie funkcji zdefiniowanych w module z komponentem sprawia, że nazwa żądanej przez Ciebie funkcji musi być poprzedzona nazwą, za pomocą której zarejestrowano moduł w magazynie danych (listing 20.26).

*Listing 20.26. Zastosowanie przestrzeni nazw modułu w pliku src/components/ProductDisplay.vue*

```

...
<script>

import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 striped: state => state.prefs.stripedTable
 }),
 ...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
 })
 },
 methods: {
 ...mapMutations({
 editProduct: "selectProduct",
 createNew: "selectProduct",
 setEditButtonColor: "prefs/setEditButtonColor",
 })
 }
}

```

```

 setDeleteButtonColor: "prefs/setDeleteButtonColor"
 }),
 ...mapActions({
 getProducts: "getProductsAction",
 deleteProduct: "deleteProductAction"
 })
},
created() {
 this.getProducts();
 this.setEditableButtonColor(false);
 this.setDeleteButtonColor(false);
}
}
</script>
...

```

Aby wybrać gettery, mutacje i akcje, korzystamy z przestrzeni nazw, po której występują ukośnik (/) i nazwa funkcji:

```

...
...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
})
...

```

Ten fragment wiąże gettery `tableClass`, `editClass` i `deleteClass` z przestrzenią nazw `prefs`.

## Podsumowanie

W tym rozdziale pokazałem, jak skorzystać z pakietu Vuex, aby utworzyć magazyn danych dla aplikacji Vue.js. Omówiłem zasady modyfikowania stanu za pomocą mutacji, łączenia wartości przy użyciu getterów, a także sposoby wykonania asynchronicznych zadań za pomocą akcji. Przedstawiłem możliwości Vuex w zakresie wiązania funkcji magazynu danych z komponentami i budowania bardziej elastycznej struktury magazynu danych przy użyciu przestrzeni nazw i modułów. W kolejnym rozdziale omówię mechanizm dynamicznych komponentów w Vue.js.

# ROZDZIAŁ 21.



## Komponenty dynamiczne

Proste aplikacje mogą prezentować użytkownikowi całą swoją zawartość w tym samym czasie. W bardziej złożonych projektach trzeba nieco zmienić takie podejście, wyświetlając różne komponenty w różnym czasie. W tym rozdziale omówię wbudowany w Vue.js mechanizm, który pozwala na dynamiczne wyświetlanie komponentów na podstawie wyborów dokonanych przez użytkownika. Komponenty są wtedy wczytywane tylko w razie potrzeby, co pozwala na ograniczenie ilości danych przetwarzanych przez użytkownika. Tabela 21.1 umieszcza dynamiczne komponenty w szerszym kontekście.

**Tabela 21.1.** Umiejscowienie magazynu danych Vuex w szerszym kontekście

Pytanie	Odpowiedź
Czym są komponenty dynamiczne?	Komponenty dynamiczne są wyświetlane w aplikacji, tylko kiedy są potrzebne.
Dlaczego są użyteczne?	Złożone aplikacje zawierają zbyt wiele funkcji, aby wyświetlać je wszystkim użytkownikowi w tym samym czasie. Możliwość zmiany wyświetlanych komponentów pozwala aplikacji na przedstawianie rozbudowanej treści bez przytłaczania użytkownika.
Jak się z nich korzysta?	Dynamiczny wybór komponentów jest możliwy za pomocą dyrektywy <code>is</code> .
Czy są jakieś pułapki lub ograniczenia?	Należy zadbać o to, aby dynamiczne komponenty nie czyniły założeń dotyczących swojego cyklu życia. Może być to problem zwłaszcza, gdy dynamiczne komponenty są dodawane do istniejącego projektu. Więcej na ten temat znajdziesz w podrozdziale „Przygotowywanie komponentów do dynamicznego cyklu życia”.
Czy są jakieś rozwiązania alternatywne?	Aplikacje nie muszą wyświetlać swoich komponentów dynamicznie. Proste aplikacje mogą wyświetlać całą swoją zawartość naraz, co pokazaliśmy w przykładowych aplikacjach we wcześniejszych rozdziałach.

- 
- **Wskazówka** Opisane w tym rozdziale mechanizmy są często używane w połączeniu z trasowaniem URL, które opisuję w rozdziale 22.
- 

Tabela 21.2 podsumowuje rozdział.

**Tabela 21.2.** Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Wybieraj komponenty dynamicznie.	Skorzystaj z atrybutu <code>is</code> i dyrektywy <code>v-bind</code> .	21.6 – 21.12
Wczytuj komponent tylko, gdy jest to niezbędne.	Zdefiniuj komponent asynchroniczny.	21.13 – 21.14
Wyłącz wskazówki wstępnego pobierania (ang. <i>prefetch hints</i> ).	Zmień konfigurację aplikacji.	21.15
Dostosuj komponenty asynchroniczne.	Skorzystaj z leniwego wczytywania opcji konfiguracji.	21.16 – 21.17

## Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę nad projektem *productapp* z rozdziału 20. Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj polecenie z listingu 21.1 w katalogu *productapp*.

### *Listing 21.1.* Uruchamianie usługi sieciowej

---

```
npm run json
```

---

Otwórz drugie okno wiersza poleceń, przejdź do katalogu *productapp* i wykonaj polecenie z listingu 21.2, aby uruchomić narzędzia deweloperskie Vue.js.

### *Listing 21.2.* Uruchamianie narzędzi deweloperskich

---

```
npm run serve
```

---

Po zakończeniu inicjalizacji otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć przykładową aplikację (rysunek 21.1).

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Pilka nożna	Pilka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>
4	Chorągiewki narożne	Pilka nożna	34.95	<button>Edytuj</button> <button>Usuń</button>
5	Stadion	Pilka nożna	79500	<button>Edytuj</button> <button>Usuń</button>
6	Myśląca czapeczka	Szachy	16	<button>Edytuj</button> <button>Usuń</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button> <button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button> <button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button> <button>Usuń</button>

**Utwórz nowy**

ID  
Nazwa  
Kategoria  
Cena  
**Utwórz** **Anuluj**

**Rysunek 21.1.** Przykładowa aplikacja po uruchomieniu

- 
- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/vue2wp.zip>.
- 

## Przygotowywanie komponentów do dynamicznego cyklu życia

Komponenty ProductDisplay i ProductEditor zostały napisane przy założeniu, że zostaną utworzone w momencieinicjalizacji aplikacji, a ich usunięcie nastąpi w momencie zakończenia aplikacji. Takie założenie przestaje być prawdziwe, gdy komponenty są wyświetlane dynamicznie, ponieważ Vue.js utworzy komponenty dopiero, gdy staną się potrzebne, a usunie je, gdy przestaną być widoczne dla użytkownika. W związku z tym obserwatorzy i funkcje obsługi zdarzeń mogą pominać ważne powiadomienia, a kosztowne operacje, takie jak pobieranie danych z usługi sieciowej, będą powtarzane przy każdym utworzeniu komponentu, mimo że aplikacja jest w stanie skorzystać z uprzednio pobranych danych.

### Pobieranie danych aplikacji

W listingu 21.3 usuwam instrukcję odpowiedzialną za pobranie początkowych danych z metody created komponentu ProductDisplay. Skoro będę wyświetlać komponenty dynamicznie, Vue.js utworzy nową instancję komponentu przy każdym wyświetleniu widoku tabeli produktów. Co za tym idzie, za każdym razem zostanie wykonany cykl życia aplikacji, łącznie z metodą created.

*Listing 21.3. Uniknięcie problemu zduplikowanych żądań w pliku src/components/ProductDisplay.vue*

```
...
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 useStripedTable: state => state.prefs.stripedTable
 }),
 ...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
 })
 },
 methods: {
 ...mapMutations({
 editProduct: "selectProduct",
 createNew: "selectProduct",
 setEditButtonColor: "prefs/setEditButtonColor",
 setDeleteButtonColor: "prefs/setDeleteButtonColor"
 }),
 ...mapActions({
 //getProducts: "getProductsAction",
 })
}

```

```

 deleteProduct: "deleteProductAction"
 })
},
created() {
 // this.getProducts();
 this.setEditableColor(false);
 this.setDeleteButtonColor(false);
}
}
</script>
...

```

Zadania, które powinny być wykonywane tylko raz, muszą być obsługiwane przez komponenty, które nie będą wyświetlane dynamicznie. W większości projektów Vue.js komponent nadzędny (główny) jest używany do ustalania widoczności innych komponentów aplikacji. Komponent ten będzie zawsze widoczny dla użytkownika. W listingu 21.4 do komponentu App dodałem kod odpowiedzialny za pobieranie początkowych danych z usługi sieciowej.

*Listing 21.4. Pobieranie danych z pliku src/App.vue*

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col"><error-display /></div>
 </div>
 <div class="row">
 <div class="col-8 m-3"><product-display /></div>
 <div class="col m-3"><product-editor /></div>
 </div>
 </div>
</template>
<script>
 import ProductDisplay from "./components/ProductDisplay";
 import ProductEditor from "./components/ProductEditor";
 import ErrorDisplay from "./components/ErrorDisplay";
 export default {
 name: 'App',
 components: { ProductDisplay, ProductEditor, ErrorDisplay },
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>

```

Skorzystałem z metody `dispatch` pakietu Vuex, aby wywołać akcję `getProductsAction`, którą zdefiniowałem w rozdziale 20. Skoro komponent App nie jest wyświetlany dynamicznie, mogę mieć pewność, że to zadanie zostanie wykonane tylko raz.

## Zarządzanie zdarzeniami obserwatora

Komponent `ProductEditor` korzysta z obserwatora magazynu danych, aby określić moment kliknięcia przycisku *Utwórz nowy* lub *Edytuj*. Gdy komponenty są tworzone dynamicznie, niezwykle trudno jest mieć pewność, że zmiany w stanie aplikacji zostaną obsłużone przed utworzeniem wymaganych komponentów. Może to doprowadzić do sytuacji, że komponent nie otrzyma zdarzeń, które miały go skonfigurować. W listingu 21.5 zmieniam komponent `ProductEditor`, aby skorzystać z istniejących wartości magazynu danych w metodzie `created`.

*Listing 21.5. Przygotowanie dynamicznego wyświetlania w pliku src/components/ProductEditor.vue*

```
...
<script>
let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 save() {
 this.$store.dispatch("saveProductAction", this.product);
 this.product = {};
 },
 cancel() {
 this.$store.commit("selectProduct");
 },
 selectProduct(selectedProduct) {
 if (selectedProduct == null) {
 this.editing = false;
 this.product = {};
 } else {
 this.editing = true;
 this.product = {};
 Object.assign(this.product, selectedProduct);
 }
 }
 },
 created() {
 unwatcher = this.$store.watch(state =>
 state.selectedProduct, this.selectProduct);
 this.selectProduct(this.$store.state.selectedProduct);
 },
 beforeDestroy() {
 unwatcher();
 }
}
</script>
...
```

Metoda created przetwarza bieżące wartości danych w magazynie i korzysta z metody watch, aby utworzyć obserwatora w celu obserwacji przyszłych zdarzeń. Wynikiem działania metody watch jest funkcja, która może zostać użyta do zatrzymania otrzymywania powiadomień. Z tej funkcji korzystam w metodzie cyklu życia beforeDestroy, aby upewnić się, że obserwator nie pozostanie aktywny po zniszczeniu komponentu.

## Dynamiczne wyświetlanie komponentów

Komponenty zostały zmodyfikowane, dzięki czemu mogą być tworzone i niszczone bez powodowania problemów. Teraz mogę zmienić sposób wyświetlania treści przez aplikację. W tym podrozdziale pokażę, jak korzystać bezpośrednio z Vue.js w celu dynamicznego wyboru komponentów. W ten sposób przygotujemy się do omówienia sposobu użycia popularnego pakietu *Vue-Router*.

## Przedstawianie różnych komponentów w elemencie HTML

Vue.js obsługuje specjalny atrybut w elementach HTML, który pozwala określić komponent użyty do zamiany treści w trakcie działania aplikacji. Tym atrybutem jest `is` i może on zostać użyty w dowolnym elemencie — choć z reguły jest stosowany w elemencie `component` (listing 21.6).

*Listing 21.6. Zastosowanie atrybutu `is` w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col"><error-display /></div>
 </div>
 <div class="row">
 <div class="col">
 <component is="ProductDisplay"></component>
 </div>
 </div>
 </div>
</template>
<script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";
export default {
 name: 'App',
 components: {
 ProductDisplay,
 ProductEditor,
 ErrorDisplay
 },
 created() {
 this.$store.dispatch("getProductsAction");
 }
}
</script>
```

- **Wskaźówka** Po zapisaniu zmian z listingu 21.6 zobaczysz ostrzeżenia lintera. Zostaną one rozstrzygnięte w kolejnym podrozdziale.

W czasie działania aplikacji Vue.js napotka atrybut `is` w elemencie `component` i oszacuje wartość atrybutu, aby określić komponent przeznaczony do wyświetlenia. W listingu atrybut `is` jest ustawiany na wartość `ProductDisplay`, która została użyta jako nazwa w instrukcji `import` w elemencie `script`. Nazwa ta była używana we właściwości `configuration` komponentu. Dzięki temu użytkownik zobaczy tabelę produktów, jak na rysunku 21.2.

## Wybór komponentów za pomocą wiązania danych

Atrybut `is` staje się niezwykle interesujący, gdy zastosuje się go z wiązaniem danych. W ten sposób komponent wyświetlany użytkownikowi może ulec zmianie w trakcie działania aplikacji. W listingu 21.7 do szablonu komponentu `App` dodajemy wiązanie danych, które ustawia wartość atrybutu `is` na podstawie wyboru użytkownika.

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>
4	Chorągiewki narożne	Piłka nożna	34.95	<button>Edytuj</button> <button>Usuń</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button> <button>Usuń</button>
6	Myśląca czapczka	Szachy	16	<button>Edytuj</button> <button>Usuń</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button> <button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button> <button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button> <button>Usuń</button>

[Utwórz nowy](#)

**Rysunek 21.2.** Wyświetlenie komponentu**Listing 21.7.** Zastosowanie wiązania danych w pliku src/App.vue

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group btn-group-toggle">
 <label class="btn btn-info"
 v-bind:class="{active: (selected == 'table') }">
 <input type="radio" v-model="selected" value="table" />
 Tabela
 </label>
 <label class="btn btn-info"
 v-bind:class="{active: (selected == 'editor') }">
 <input type="radio" v-model="selected" value="editor" />
 Edytor
 </label>
 </div>
 </div>
 </div>
 <div class="row">
 <div class="col">
 <component v-bind:is="selectedComponent"></component>
 </div>
 </div>
 </div>
</template>
<script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";

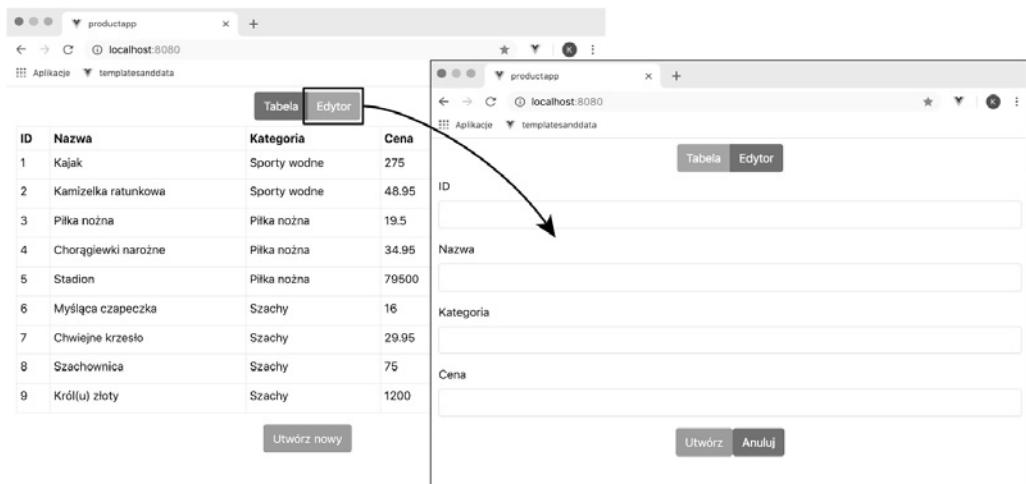
```

```

export default {
 name: 'App',
 components: {
 ProductDisplay,
 ProductEditor,
 ErrorDisplay
 },
 created() {
 this.$store.dispatch("getProductsAction");
 },
 data: function() {
 return {
 selected: "table"
 }
 },
 computed: {
 selectedComponent() {
 return this.selected == "table" ? ProductDisplay : ProductEditor;
 }
 }
}
</script>

```

Elementy input w tym przykładzie pozwalają użytkownikowi na wybór wyświetlanego komponentu przez zmianę właściwości danych selected za pomocą dyrektywy v-model. Wartość atrybutu is w elemencie component odzwierciedla wybraną wartość za pomocą dyrektywy v-bind, która odczytuje wartość właściwości obliczonej i korzysta z wartości selected, aby zidentyfikować wymagany przez użytkownika komponent, co daje efekt jak na rysunku 21.3.

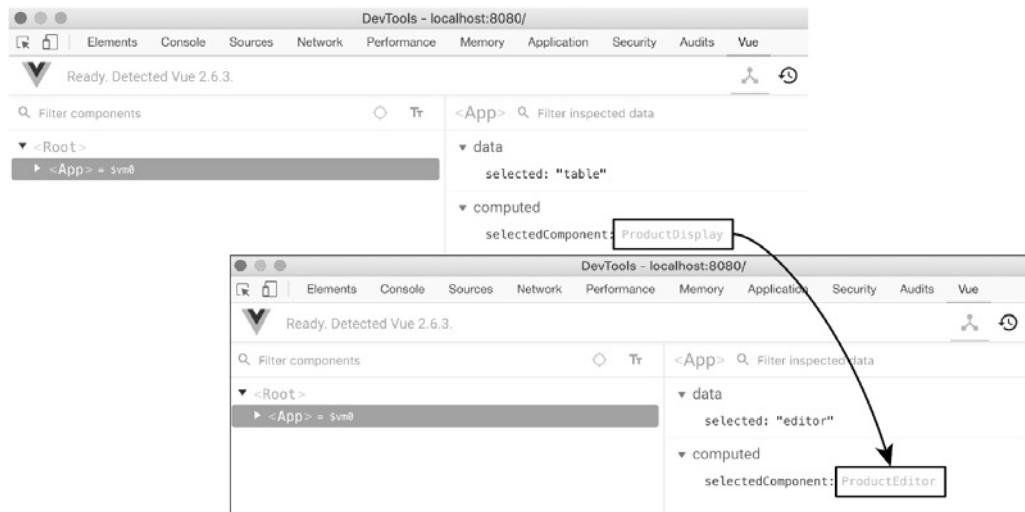


Rysunek 21.3. Wybór komponentu za pomocą wiązania danych

Użytkownik widzi grupę przycisków, za pomocą której można wybrać wyświetlany komponent — kliknięcie przycisku *Tabela* spowoduje wyświetlenie komponentu *ProductDisplay*, a przycisku *Edytor* — wyświetlenie komponentu *ProductEditor*.

## Omówienie cyklu życia komponentu

W tym momencie warto przeanalizować stan aplikacji, aby zobaczyć, w jaki sposób Vue.js obsługuje dynamiczne zmiany. Kliknij przyciski *Tabela* i *Edytor* w trakcie przeglądania komponentów aplikacji za pomocą narzędzia Vue Devtools, a przekonasz się, że komponenty są tworzone i niszczone w miarę potrzeby. Prowadzi to do sytuacji, w której komponent *App* w dowolnym momencie ma tylko jedno dziecko (rysunek 21.4).



Rysunek 21.4. Tworzenie i niszczenie komponentów

### Wielokrotne używanie komponentów dynamicznych

Vue.js umożliwia nieco inne podejście w zakresie zarządzania komponentami dynamicznymi, które nie wymaga ciągłego ich tworzenia i niszczenia. Jeśli otoczyesz element, który wyświetla komponent (ten z atrybutem *is*), elementem *keep-alive*, Vue.js zdezaktywuje komponent, zamiast zniszczyć go. Oto przykład zastosowania elementu *keep-alive*:

```
...
<div class="row">
 <div class="col">
 <keep-alive>
 <component v-bind:is="selectedComponent"></component>
 </keep-alive>
 </div>
</div>
...
```

Dzięki takiemu podejściu nie musisz martwić się o dostosowanie komponentów tak, aby uniknąć powtarzania jednorazowych zadań lub pominięcia ważnych zdarzeń. Wadą tego podejścia jest większe zużycie zasobów — nieaktywny komponent dalej korzysta z zasobów, a jeśli nie będzie ponownie wykorzystany w aplikacji, zajęcie zasobów okaże się bezproduktywne.

Moim zdaniem to Vue.js powinno decydować o tworzeniu i niszczeniu komponentów. Jeśli jednak chcesz skorzystać z komponentu *keep-alive*, możesz zapisać się na powiadomienia o aktywacji i dezaktywacji komponentu, implementując metody *activated* i *deactivated* (w rozdziale 17. znajdziesz więcej informacji na temat cyklu życia komponentu).

## Automatyczna nawigacja w aplikacji

Pozwolenie użytkownikowi na wybór wyświetlanego komponentu w aplikacji stanowi dobry przykład działania dynamicznych komponentów, niemniej nie jest to typowy sposób, w jaki działają aplikacje. Zdecydowanie wołalbym, aby to aplikacja sama wyświetlała odpowiednie treści na podstawie pewnych działań użytkownika. To komponenty aplikacji — wszystkie — powinny mieć możliwość zmiany aktualnie wyświetlanego komponentu. Kliknięcie przycisku *Edytuj* powinno przenieść użytkownika do komponentu edytora.

Zaczynam od dodania pliku *src/store/navigation.js* o treści z listingu 21.8.

*Listing 21.8. Zawartość pliku src/store/navigation.js*

```
export default {
 namespaced: true,
 state: {
 selected: "table"
 },
 mutations: {
 selectComponent(currentState, selection) {
 currentState.selected = selection;
 }
 }
}
```

Ten moduł Vuex rozszerza magazyn danych, dzięki czemu mogę poruszać się po aplikacji. W atrybutie `is` skorzystam z właściwości stanu `selected` i zmienię wartość, używając mutacji `selectComponent`. W listingu 21.9 importuję moduł do głównego magazynu danych.

*Listing 21.9. Dodawanie modułu w pliku src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import PrefsModule from "./preferences";
import NavModule from "./navigation";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
 modules: {
 prefs: PrefsModule,
 nav: NavModule
 },
 state: {
 products: [],
 selectedProduct: null
 },
 //...pozostałe funkcje magazynu danych zostały pominięte...
})
```

Kolejnym krokiem jest zastosowanie nowej właściwości stanu magazynu danych jako wartości atrybutu `is` w szablonie komponentu `App` (listing 21.10). Poza zastosowaniem magazynu danych usuwam przyciski, które pozwalają na jawną wybór wyświetlanego komponentu.

*Listing 21.10. Zastosowanie magazynu danych w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <!-- <div class="row">
 <div class="col text-center m-2">
```

```

<div class="btn-group btn-group-toggle">
 <label class="btn btn-info"
 v-bind:class="{active: (selected == 'table') }">
 <input type="radio" v-model="selected" value="table" />
 Tabela
 </label>
 <label class="btn btn-info"
 v-bind:class="{active: (selected == 'editor') }">
 <input type="radio" v-model="selected" value="editor" />
 Edytor
 </label>
</div>
</div> -->
<div class="row">
 <div class="col">
 <component v-bind:is="selectedComponent"></component>
 </div>
</div>
</div>
</template>
<script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";
import { mapState } from "vuex";
export default {
 name: 'App',
 components: {
 ProductDisplay,
 ProductEditor,
 ErrorDisplay
 },
 created() {
 this.$store.dispatch("getProductsAction");
 },
 // data:function() {
 // return {
 // selected: "table"
 // }
 // },
 computed: {
 ...mapState({
 selected: state => state.nav.selected
 }),
 selectedComponent() {
 return this.selected == "table" ? ProductDisplay : ProductEditor;
 }
 }
}
</script>

```

W tym momencie nie pozostało nam nic innego, jak wykonać mutację magazynu danych, aby zmienić komponent wyświetlany użytkownikowi. W listingu 21.11 modyfikuję komponent ProductDisplay tak, aby użytkownik widział edytor po kliknięciu przycisku *Utwórz nowy* lub *Edytuj*.

**Listing 21.11.** Dodawanie nawigacji do pliku *src/components/ProductDisplay.vue*

```

...
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 useStripedTable: state => state.prefs.stripedTable
 }),
 ...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
 })
 },
 methods: {
 editProduct(product) {
 this.selectProduct(product);
 this.selectComponent("editor");
 },
 createNew() {
 this.selectProduct();
 this.selectComponent("editor");
 },
 ...mapMutations({
 selectProduct: "selectProduct",
 selectComponent: "nav/selectComponent",
 // editProduct: "selectProduct",
 // createNew: "selectProduct",
 setEditButtonColor: "prefs/setEditButtonColor",
 setDeleteButtonColor: "prefs/setDeleteButtonColor"
 }),
 ...mapActions({
 deleteProduct: "deleteProductAction"
 })
 },
 created() {
 this.setEditButtonColor(false);
 this.setDeleteButtonColor(false);
 }
}
</script>
...

```

Zmieniłem definicje metod `editProduct` i `createNew`, dzięki czemu nie są one bezpośrednio powiązane z mutacją `selectProduct` magazynu danych. Zamiast tego utworzyłem metody lokalne, które wykonują mutacje `selectProduct` i `selectComponent`. Dzięki nim identyfikuję obiekt, który zostanie poddany edycji, a następnie wyświetlę edytor komponentów. W listingu 21.12 modyfikuję komponent `ProductEditor`, przez co po zakończeniu lub anulowaniu procesu edycji aplikacja powraca do widoku tabeli.

**Listing 21.12.** Dodawanie nawigacji w pliku src/components/ProductEditor.vue

```
...
<script>
let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$store.commit("nav/selectComponent", "table");
 this.product = {};
 },
 cancel() {
 this.$store.commit("selectProduct");
 this.$store.commit("nav/selectComponent", "table");
 },
 selectProduct(selectedProduct) {
 if (selectedProduct == null) {
 this.editing = false;
 this.product = {};
 } else {
 this.editing = true;
 this.product = {};
 Object.assign(this.product, selectedProduct);
 }
 }
 },
 created() {
 unwatcher = this.$store.watch(state =>
 state.selectedProduct, this.selectProduct);
 this.selectProduct(this.$store.state.selectedProduct);
 },
 beforeDestroy() {
 unwatcher();
 }
}
</script>
...
```

### Wyodrębnianie procesu nawigacji

Zwróć uwagę, że nawigacja jest obsługiwana w komponentach przykładowej aplikacji, a nie bezpośrednio w mutacjach magazynu danych, takich jak `selectProduct`. Choć intuicyjnie wykonywanie nawigacji w ramach zmiany stanu może wydawać się dobrym pomysłem, zakłada się w ten sposób, że komponenty będą wyświetlane w momencie wykonania mutacji lub akcji. Tak może się zdarzyć, gdy zaczynasz przygodę z dynamicznym wyświetlaniem komponentów, jednak w miarę rozwoju projektu jego złożoność rośnie, a użytkownik często może docierać do danej funkcji aplikacji na różne sposoby. Implementacja nawigacji w każdym komponencie może wydawać się skomplikowana, ale dzięki temu łatwiej jest dodawać nowe funkcje do aplikacji.

Ten komponent korzysta bezpośrednio z funkcji magazynu danych, bez potrzeby stosowania funkcji mapujących Vuex. Do metody save dodałem słowo kluczowe `async`, które pozwala na użycie słowa `await` w momencie zapisu produktu, dzięki czemu nawigacja nie zostanie wykonana do chwili zakończenia operacji HTTP.

Dzięki temu komponent będzie wyświetlany automatycznie w wyniku działań użytkownika, automatycznie przełączając się między tabelą a edytorem (rysunek 21.5).



Rysunek 21.5. Nawigacja między komponentami

## Stosowanie komponentów asynchronicznych

W większych aplikacjach często zdarzają się funkcje, które nie są niezbędne dla wszystkich użytkowników lub są stosowane dość rzadko — np. zaawansowane ustawienia lub narzędzia administracyjne. Domyślnie te nieużywane komponenty są dołączane do paczki z kodem JavaScript wysyłanej do przeglądarki. Jest to jednak niepotrzebne marnotrawstwo łącza, które dodatkowo opóźnia start aplikacji. Aby uniknąć tego problemu, Vue.js udostępnia mechanizm komponentów asynchronicznych, w przypadku których ładowanie komponentów jest odkładane do czasu ich faktycznego użycia — takie ładowanie jest określone mianem **leniwego** (ang. *lazy loading*). Aby zademonstrować przykład takiego komponentu, do katalogu `src/components` dodaję plik `DataSummary.vue` (listing 21.13).

Listing 21.13. Zawartość pliku `src/components/DataSummary.vue`

```
<template>
 <div>
 <h3 class="bg-success text-center text-white p-2">
 Podsumowanie
 </h3>
 <table class="table">
 <tr><th>Liczba produktów:</th><td> {{ products.length }} </td></tr>
 <tr><th>Liczba kategorii:</th><td> {{ categoryCount }} </td></tr>
 <tr>
 <th>Najwyższa cena:</th><td> {{ highestPrice | currency }} </td>
 </tr></table>
 </div>
</template>
<script>
import {
 mapState,
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 }
}</script>
```

```

categoryCount() {
 if (this.products.length > 0) {
 return this.products.map(p => p.category)
 .filter((cat, index, arr) => arr.indexOf(cat) ==
 index).length;
 } else {
 return 0;
 }
},
highestPrice() {
 if (this.products.length == 0) {
 return 0;
 } else {
 return Math.max(...this.products.map(p => p.price));
 }
},
filters: {
 currency(value) {
 return new Intl.NumberFormat("pl-PL", {
 style: "currency",
 currency: "PLN"
 }).format(value);
 }
}
}
</script>

```

Ten komponent wyświetla podsumowanie danych z magazynu. W tym rozdziale założyłem sobie, że nie chcę ładować tej funkcji, dopóki nie będzie rzeczywiście potrzebna. W listingu 21.14 zarejestrowałem nowy komponent tak, aby był ładowany leniwie.

*Listing 21.14. Leniwe ładowanie komponentu w pliku src/components/App.vue*

```

<template>
<div class="container-fluid">
 <div class="col">
 <div class="col text-center m-2">
 <button class="btn btn-primary"
 v-on:click="selectComponent('table')">
 Funkcje standardowe
 </button>
 <button class="btn btn-success"
 v-on:click="selectComponent('summary')">
 Funkcje zaawansowane
 </button>
 </div>
 </div>
 <div class="row">
 <div class="col">
 <component v-bind:is="selectedComponent"></component>
 </div>
 </div>
</div>
</template>
<script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";
const DataSummary = () => import("./components/DataSummary");

```

```

import {
 mapState,
 mapMutations
} from "vuex";
export default {
 name: 'App',
 components: {
 ProductDisplay,
 ProductEditor,
 ErrorDisplay,
 DataSummary
 },
 created() {
 this.$store.dispatch("getProductsAction");
 },
 methods: {
 ...mapMutations({
 selectComponent: "nav/selectComponent"
 })
 },
 computed: {
 ...mapState({
 selected: state => state.nav.selected
 }),
 selectedComponent() {
 switch (this.selected) {
 case "table":
 return ProductDisplay;
 case "editor":
 return ProductEditor;
 case "summary":
 return DataSummary;
 }
 }
 }
}
</script>

```

## Konsekwencje ładowania leniwego

Pojęcie *leniwy* odnosi się do faktu, że komponent nie zostanie wczytany z serwera HTTP, dopóki nie okaże się on rzeczywiście konieczny. Alternatywnym podejściem jest ładowanie wczesne, zwane też **zachłannym** (ang. *eager loading*), w przypadku którego komponenty są ładowane jako elementy głównej paczki aplikacji, nawet jeśli nie będą potrzebne.

Oba podejścia mają swoje wady. Ładowanie zachłanne wymaga większej ilości danych do pobrania i zabiera więcej czasu, sprawiając przy tym lepsze wrażenie, ponieważ cały kod i cała treść są dostępne od razu. Alternatywnie ładowanie leniwe zmniejsza rozmiar początkowego zestawu danych do pobrania, ale w razie wykorzystania dodatkowych funkcji konieczne jest wykonanie dodatkowych żądań HTTP.

Decyzję o wyborze ładowania leniwego lub zachłannego należy podjąć na podstawie oczekiwania co do sposobu użycia aplikacji. Warto jednak weryfikować swoje założenia już po wdrożeniu aplikacji. Jeśli zauważysz, że użytkownicy często wykonują akcje związane z leniwym ładowaniem komponentów, warto zastanowić się nad zmianą konfiguracji — nie ma sensu niepotrzebnie wydłużać czasu, który użytkownik musi tracić na leniwe załadowanie zasobów.

W listingu pojawiło się kilka zmian, ale to poniższa zmiana odpowiada za leniwe ładowanie komponentu:

```
...
const DataSummary = () => import("./components/DataSummary");
...
```

Zwykłe użycie słowa kluczowego `import` tworzy zależność statyczną, która powoduje dołączenie komponentu do paczki JavaScript przez webpacka. Powyższa postać instrukcji `import`, zapisana w postaci funkcji, w której komponent jest określony jako argument, tworzy zależność dynamiczną. W związku z tym webpack umieści komponent w odrębnej paczce. Efektem działania funkcji `import` jest obietnica (ang. *promise*), która zostanie wypełniona w momencie załadowania komponentu. Proces ładowania rozpocznie się tuż po wykonaniu funkcji `import`, dlatego ważne jest, aby zachować referencję do komponentu (w tym przykładzie `DataSummary`), czyli funkcję, która wykonuje operację importowania.

```
...
const DataSummary = () => import("./components/DataSummary");
...
```

Gdy obiekt `DataSummary` zostanie wybrany za pomocą atrybutu `is`, Vue.js wykryje funkcję, która oznacza komponent asynchroniczny. Funkcja zostanie wywołana, aby rozpocząć proces ładowania. Komponent zostanie wyświetlony po zakończeniu ładowania.

## Wyłączanie podpowiedzi wstępnego pobierania

Domyślnie projekt dostarcza przeglądarce podpowiedzi wstępnego pobierania (ang. *prefetch hints*), które wskazują, że pewien fragment treści aplikacji może być potrzebny w przyszłości. Założeniem tej funkcji jest pozostawienie przeglądarki decyzji o pobraniu treści, zanim będzie ona potrzebna. Ten mechanizm koliduje z ideą leniwego ładowania modułów Vue.js, które są wymagane przez niewielką liczbę użytkowników (gdy pobrany plik JavaScript nie będzie przez nich używany). Aby wyłączyć podpowiedzi wstępnego pobierania, należy do katalogu `productapp` dodać plik `vue.config.js` o treści z listingu 21.15.

**Listing 21.15.** Wyłączanie podpowiedzi wstępnego pobierania w pliku `productapp/vue.config.js`

```
module.exports = {
 chainWebpack: config => {
 config.plugins.delete('prefetch');
 }
}
```

Aby wdrożyć zmiany konfiguracji, zatrzymaj narzędzia deweloperskie i zrestartuj je, uruchamiając polecenie z listingu 21.16 w katalogu `productapp`.

**Listing 21.16.** Uruchamianie narzędzi deweloperskich

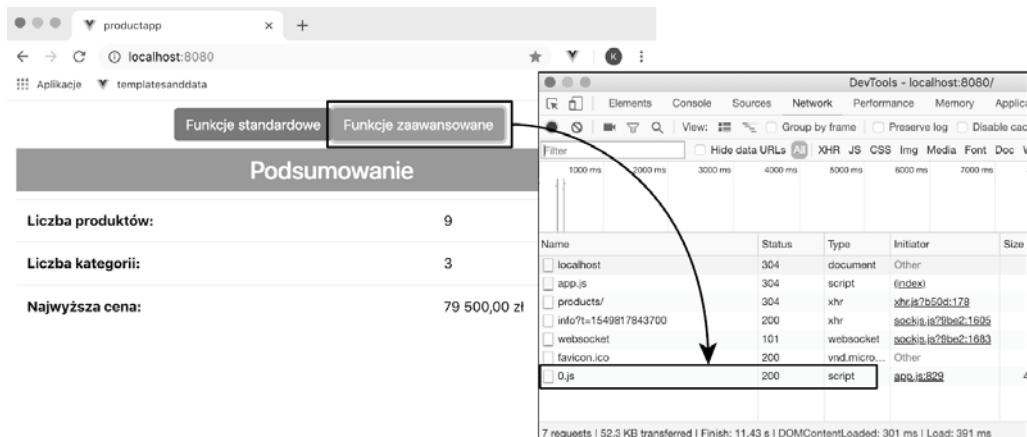
---

```
npm run serve
```

---

Aby przetestować komponent asynchroniczny, przejdź pod adres `http://localhost:8080` i kliknij przycisk *Funkcje zaawansowane*, który wyświetli komponent zdefiniowany w listingu 21.13, co widać na rysunku 21.6.

Nie sądzę, że zauważysz opóźnienie w ładowaniu, ponieważ przeglądarka i serwer HTTP są uruchomione na tym samym komputerze. Jeśli jednak przejdziesz do narzędzi deweloperskich F12 i wyświetlisz zakładkę *Network*, zobaczyesz, że po kliknięciu przycisku *Funkcje zaawansowane* zostanie wykonane dodatkowe żądanie HTTP. Będzie to plik, który zawiera komponent `DataSummary`. Na moim komputerze nosi on nazwę `0.js`, choć u Ciebie może być nieco inaczej.



Rysunek 21.6. Leniwe ładowanie komponentu

## Konfiguracja leniwego ładowania

Vue.js udostępnia szereg opcji, które mogą być użyte w celu dostrojenia procesu ładowania komponentów asynchronicznych. Są one opisane w tabeli 21.3.

Tabela 21.3. Opcje konfiguracji leniwego ładowania

Nazwa	Opis
component	Ta właściwość jest używana w celu zdefiniowania funkcji import, która wczytuje komponent asynchroniczny.
loading	Ta właściwość jest używana do określenia komponentu, który będzie wyświetlany użytkownikowi podczas procesu ładowania.
delay	Ta właściwość jest używana do określania opóźnienia (w milisekundach), jakie musi nastąpić, aby wyświetlić komponent loading. Domyślna wartość to 200.
error	Ta właściwość jest używana do określenia komponentu, który będzie wyświetlany, jeśli operacja ładowania nie powiedzie się lub zostanie przekroczony czas jej wykonania.
timeout	Ta właściwość jest używana do określenia maksymalnego czasu ładowania zasobów, wyrażonego w milisekundach. Domyślnie nie ma ograniczenia czasu ładowania.

Najbardziej użyteczną z tych opcji jest właściwość `loading`, która określa komponent do wyświetlenia w czasie ładowania asynchronicznego komponentu. Opcja `delay` jest również istotna, ponieważ określa czas, jaki może zajść ładowanie, zanim komponent ładowania nie zostanie wyświetlony. Dzięki niej możemy się upewnić, że użytkownik nie zobaczy komponentu ładowania w przypadku operacji, które zajmują niewiele czasu.

■ **Ostrzeżenie** Nie powinno się korzystać z właściwości `loading` w połączeniu z komponentem ładowanym leniwie, ponieważ może on nie zostać wczytany w momencie, w którym aplikacja zechce z niego skorzystać.

Aby przygotować się do użycia właściwości opisanych w tabeli 21.3, muszę utworzyć komponent, który będzie wyświetlony w czasie ładowania. W związku z tym do katalogu `src/components` dodaję plik `LoadingMessage.vue` o treści z listingu 21.17.

**Listing 21.17.** Zawartość pliku *src/components>LoadingMessage.vue*

```
<template>
 <h3 class="bg-info text-white text-center m-2 p-2">
 Komponent ładowany leniwie...
 </h3>
</template>
```

Komponent składa się wyłącznie z szablonu, który wyświetli użytkownikowi komunikat. W listingu 21.18 korzystam z komponentu, aby skonfigurować leniwe ładowanie komponentu DataSummary.

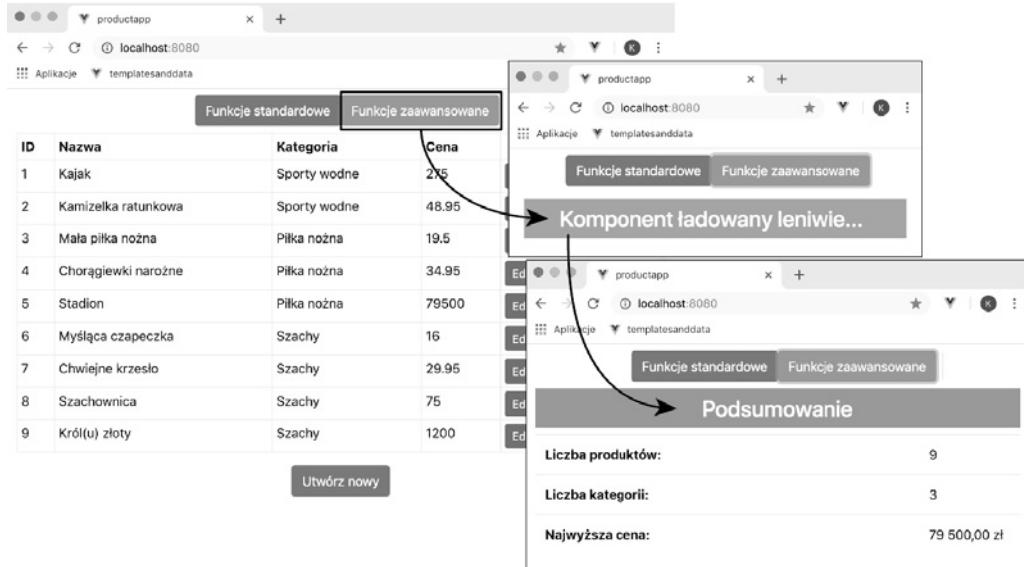
**Listing 21.18.** Konfiguracja leniwego ładowania w pliku *src/App.vue*

```
...
<script>
import ProductDisplay from "./components/ProductDisplay";
import ProductEditor from "./components/ProductEditor";
import ErrorDisplay from "./components/ErrorDisplay";
import LoadingMessage from "./components>LoadingMessage";
const DataSummary = () => ({
 component: import("./components/DataSummary"),
 loading: LoadingMessage,
 delay: 100
});
import {
 mapState,
 mapMutations
} from "vuex";
export default {
 name: 'App',
 components: {
 ProductDisplay,
 ProductEditor,
 ErrorDisplay,
 DataSummary
 },
 created() {
 this.$store.dispatch("getProductsAction");
 },
 methods: {
 ...mapMutations({
 selectComponent: "nav/selectComponent"
 })
 },
 computed: {
 ...mapState({
 selected: state => state.nav.selected
 }),
 selectedComponent() {
 switch (this.selected) {
 case "table":
 return ProductDisplay;
 case "editor":
 return ProductEditor;
 case "summary":
 return DataSummary;
 }
 }
 }
}
```

```
}
</script>
...

```

Właściwość component otrzymuje funkcję import, która wczyta komponent. Oprócz tego postanowiłem, że użytkownik zobaczy komponent LoadingMessage, jeśli operacja ładowania zajmie więcej niż 100 milisekund. Aby się o tym przekonać, przejdź pod adres <http://localhost:8080> i kliknij przycisk *Funkcje zaawansowane*. W ten sposób zobaczysz sekwencję ładowania pokazaną na rysunku 21.7.



Rysunek 21.7. Konfiguracja procesu leniwego ładowania

- **Uwaga** Przetestowanie omawianych funkcji podczas tworzenia aplikacji może być trudne, ponieważ komponent asynchroniczny zostanie wczytany zbyt szybko, aby zobaczyć komponent określony we właściwości component. W związku z tym korzystam z narzędzi deweloperskich przeglądarki Google Chrome, aby utworzyć profil sieciowy, który doda kilka sekund opóźnienia do moich żądań HTTP. Używam tego profilu przed wywołaniem leniwego ładowania.

### Łączenie komponentów we wspólną paczkę

Domyślnie każdy asynchroniczny komponent zostanie umieszczony we własnym pliku i zostanie wczytany tylko w miarę potrzeby. Alternatywnym podejściem jest zgrupowanie powiązanych komponentów, dzięki czemu zostaną one wczytane przy pierwszej okazji, gdy któryś z nich będzie potrzebny. W tym celu musimy dodać komentarz, który zostanie wykryty przez webpacka podczas budowania:

```
...
const DataSummary = () => ({
 component:
 import(/* webpackChunkName: "advanced" */ "./components/DataSummary"),
 loading: LoadingMessage,
 delay: 100
});
...

```

Komentarz ustawia wartość właściwości webpackChunkName. Webpack doda wszystkie komponenty asynchroniczne o tej samej wartości webpackChunkName do tej samej paczki. Nazwa nie zostanie użyta jako nazwa paczki — ta zostanie określona dynamicznie podczas procesu budowania.

---

## Podsumowanie

W tym rozdziale pokazałem, jak można wyświetlać komponenty dynamicznie. Zademonstrowałem sposób użycia atrybutu `is` w celu wyboru komponentu, co jest dobrym podejściem w prostszych projektach. Pokazałem też, jak wczytywać komponenty na żądanie, co może być dobre w przypadku komponentów, które nie są niezbędne dla wszystkich użytkowników. W kolejnym rozdziale wprowadzam trasowanie adresów URL, które wykorzystuje funkcje omówione w tym rozdziale.



## ROZDZIAŁ 22.



# Trasowanie URL

W tym rozdziale opisuję mechanizm **trasowania URL** (ang. *URL routing*), który opiera się na dynamicznych komponentach opisanych w rozdziale 21., ale wykorzystuje aktualny adres URL, aby wybrać komponenty wyświetlane w danej chwili użytkownikowi. Trasowanie URL to złożony temat, dlatego będę omawiał to zagadnienie również w rozdziałach 23. i 24. Tabela 22.1 umiejscowia trasowanie URL w szerszym kontekście.

Tabela 22.1. Umiejscowienie trasowania URL w szerszym kontekście

Pytanie	Odpowiedź
Czym jest trasowanie URL?	Trasowanie URL wybiera komponenty do wyświetlenia na podstawie aktualnego adresu URL.
Dlaczego jest użyteczne?	Zastosowanie adresu URL do wyboru komponentów pozwala na przejście do konkretnej części aplikacji. Dzięki trasowaniu złożone aplikacje mogą być budowane w sposób, który jest łatwiejszy do późniejszego zarządzania aniżeli przez wybór komponentów w kodzie.
Jak się z niego korzysta?	Należy dodać do projektu pakiet Vue Router. Element <code>router-view</code> jest używany do wyświetlenia komponentu na podstawie konfiguracji trasowania aplikacji.
Czy są jakieś pułapki lub ograniczenia?	Trudno jest znaleźć równowagę między spójnym wyrażaniem tras a tworzeniem tras, które są łatwe do zrozumienia i zarządzania.
Czy są jakieś rozwiązania alternatywne?	Trasowanie URL jest opcjonalne, a funkcje opisane w tym rozdziale są niezbędne tylko dla złożonych aplikacji.

Tabela 22.2 podsumowuje rozdział.

## Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę nad projektem `productapp` z rozdziału 21. Trasowanie URL wymaga zainstalowania w projekcie pakietu `vue-router`. Wykonaj polecenie z listingu 22.1 w katalogu `productapp`, aby zainstalować pakiet.

**Tabela 22.2.** Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Skonfiguruj trasowanie URL.	Utwórz obiekt <code>VueRouter</code> i przekaż do niego obiekt konfiguracji, który ma właściwość <code>routes</code> .	22.4 – 22.5
Wyświetl komponent trasowany.	Skorzystaj z elementu <code>router-view</code> .	22.6
Nawiguj z poziomu kodu.	Skorzystaj z metody <code>\$router.push</code> .	22.7
Nawiguj z poziomu szablonu.	Skorzystaj z elementu <code>router-link</code> .	22.8
Skonfiguruj tryb trasowania.	Skorzystaj z właściwości <code>mode</code> w trakcie tworzenia obiektu <code>VueRouter</code> .	22.9
Zdefiniuj trasę przechwytyującą wszystko ( <i>catchall</i> ).	Zdefiniuj trasę przy użyciu <code>*</code> jako ścieżki.	22.10
Zdefiniuj alias dla trasy.	Skorzystaj z właściwości <code>trasy</code> .	22.11
Pobierz szczegóły trasy w kodzie.	Skorzystaj z obiektu <code>\$route</code> .	22.12 – 22.14
Skonfiguruj adresy URL dopasowane przez trasę.	Dodaj segment dynamiczny, korzystaj z wyrażeń regularnych lub segmentów opcjonalnych.	22.15 – 22.20
Przypisz nazwę do trasy.	Skorzystaj z właściwości <code>name</code> .	22.21 – 22.23
Otrzymuj powiadomienia, gdy zmienia się aktywna trasa.	Zaimplementuj jedną lub więcej metod-strażników.	22.24

**Listing 22.1.** Instalacja pakietu do trasowania

```
npm install vue-router@3.0.1
```

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj polecenie z listingu 22.2 w katalogu `productapp`.

**Listing 22.2.** Uruchamianie usługi sieciowej

```
npm run json
```

Otwórz drugie okno wiersza poleceń, przejdź do katalogu `productapp` i wykonaj polecenie z listingu 22.3, aby uruchomić narzędzia deweloperskie Vue.js.

**Listing 22.3.** Uruchamianie narzędzi deweloperskich

```
npm run serve
```

Po zakończeniu inicjalizacji otwórz okno przeglądarki i przejdź pod adres `http://localhost:8080`, aby zobaczyć przykładową aplikację (rysunek 21.1).

		Funkcje standardowe	Funkcje zaawansowane	
ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>
4	Chorągiewki narożne	Piłka nożna	34.95	<button>Edytuj</button> <button>Usuń</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button> <button>Usuń</button>
6	Myśląca czapeczka	Szachy	16	<button>Edytuj</button> <button>Usuń</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button> <button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button> <button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button> <button>Usuń</button>

[Utwórz nowy](#)

Rysunek 22.1. Uruchomienie przykładowej aplikacji

## Rozpoczynamy pracę z trasowaniem URL

Zanim szczegółowo omówię proces używania i konfigurowania mechanizmu trasowania URL, przedstawię krótkie wprowadzenie do głównych funkcji, które pozwoli określić pewien kontekst dla bardziej zaawansowanych zagadnień. Pierwszy krok przy wyłączeniu trasowania URL polega na skonfigurowaniu zestawu tras, które stanowią mapowanie pomiędzy adresami URL, obsługiwanyimi przez aplikację, a wyświetlonymi komponentami. Zwyczajowo konfigurację trasowania umieszcza się w katalogu *router*. W związku z tym w przykładowym projekcie tworzę katalog *src/router* i dodaję do niego plik *index.js* o treści z listingu 22.4.

Listing 22.4. Zawartość pliku *src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 routes: [
 {
 path: "/",
 component: ProductDisplay
 },
 {
 path: "/edit",
 component: ProductEditor
 }
]
})
```

- 
- **Wskazówka** W trakcie tworzenia projektu, podczas wyboru pojedynczych funkcji do dodania, możesz zaznaczyć opcję *Router*. W ten sposób pakiet zostanie zainstalowany automatycznie, a do projektu zostanie dołączona podstawowa konfiguracja trasowania.
- 

Niezwyczajne jest prawidłowe zdefiniowanie podstawowej konfiguracji, dlatego fragment kodu z listingu 22.4 omówię bardzo szczegółowo, podobnie jak w rozdziale 20. przy omawianiu magazynu danych Vuex. Pierwszy zestaw instrukcji importuje z innych modułów funkcje wymagane przez konfigurację trasowania.

```
...
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
...
```

Pierwsze dwie instrukcje `import` wczytują pakiety związane z Vue.js i Vue Routerem. Pozostałe instrukcje dają dostęp do komponentów, które są wyświetlane w aplikacji. W typowym, rzeczywistym projekcie lista ta może być bardzo długa.

Kolejna instrukcja włącza mechanizm Vue Routera:

```
...
Vue.use(VueRouter);
...
```

Vue Router jest dostarczany w formie wtyczki Vue.js, która pozwala na rozszerzanie podstawowych funkcji Vue.js, co opisuję w rozdziale 26. Wtyczki są instalowane za pomocą metody `Vue.use`.

- 
- **Ostrzeżenie** Jeśli zapomnisz wywołać metodę `Vue.use`, Vue.js nie rozpozna elementów HTML `router-view` i `router-link`, używanych przez pakiet Vue Router.
- 

Kolejna instrukcja tworzy konfigurację trasowania i ustawia ją jako domyślny eksport z modułu w katalogu routes:

```
...
export default new VueRouter({
 ...
})
```

Słowo kluczowe `new` jest używane do utworzenia obiektu `VueRouter`, który przyjmuje obiekt konfiguracji. Konfiguracja w tym przykładzie to podstawowy zbiór instrukcji pozwalający wyświetlać dwa komponenty.

```
...
routes: [
 { path: "/", component: ProductDisplay },
 { path: "/edit", component: ProductEditor }
]
...
```

Właściwość `routes` definiuje powiązania pomiędzy adresami URL a komponentami. Powiązania informują Vue Router o konieczności wyświetlania komponentu `ProductDisplay` w przypadku użycia domyślnej trasy, a z kolei w przypadku trasy `/edit` konieczne jest wyświetlenie komponentu `ProductEditor`. Nie martw się trasami URL — już niebawem zobaczysz, jak stosuje się je w praktyce, co pozwoli Ci je lepiej zrozumieć.

## Dostęp do konfiguracji trasowania

Kolejny krok polega na dodaniu funkcji trasowania do aplikacji, dzięki czemu może być ona używana w komponentach (listing 22.5).

*Listing 22.5. Włączanie konfiguracji trasowania w pliku src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "../node_modules/bootstrap/dist/css/bootstrap.min.css";
import {
 RestDataSource
} from "./restDataSource";
import store from "./store";
import router from "./router";
Vue.config.productionTip = false
new Vue({
 render: h => h(App),
 data: {
 eventBus: new Vue()
 },
 store,
 router,
 provide: function() {
 return {
 eventBus: this.eventBus,
 restDataSource: new RestDataSource(this.eventBus)
 }
 }
}).$mount('#app')
```

Aby dodać trasowanie URL do aplikacji, korzystam z instrukcji `import`, która odwołuje się do modułu `router` (nie muszę określić jawnie pliku `index.js`, ponieważ jest to domyślna nazwa pliku poszukiwanego w momencie importowania). Instrukcja `import` wczyta konfigurację trasowania z pliku `index.js` w katalogu `router`. Dodaję właściwość `router` do obiektu konfiguracji Vue, dzięki czemu konfiguracja trasowania URL będzie dostępna dla innych komponentów aplikacji.

- **Ostrzeżenie** Jeśli zapomnisz dodać właściwość `router` pokazaną w listingu 22.5, pakiet trasowania URL nie będzie skonfigurowany prawidłowo, przez co napotkasz błędy w nadchodzących przykładach.

## Stosowanie systemu trasowania do wyświetlania komponentów

Po włączeniu systemu trasowania mogę skorzystać z niego w celu wyświetlania komponentów użytkownikowi. W listingu 22.6 korzystam z trasowania URL, aby zamienić istniejącą treść i kod, które wyświetlają komponenty dynamicznie.

*Listing 22.6. Zastosowanie trasowania URL w pliku src/App.vue*

```
<template>
<div class="container-fluid">
 <div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
 </div>
</div>
```

```

</template>
<script>
 // import ProductDisplay from "./components/ProductDisplay";
 // import ProductEditor from "./components/ProductEditor";
 // import ErrorDisplay from "./components/ErrorDisplay";

 // import { mapState } from "vuex";
 export default {
 name: 'App',
 // components: { ProductDisplay, ProductEditor, ErrorDisplay },
 created() {
 this.$store.dispatch("getProductsAction");
 },
 // computed: {
 // ...
 // ...mapState({
 // selected: state => state.nav.selected
 // }),
 // selectedComponent() {
 // return this.selected == "table" ? ProductDisplay : ProductEditor;
 // }
 // }
 }
</script>

```

Pakiet Vue Router korzysta z elementu router-view do wyświetlania treści, zastępując poprzedni element, w którym skorzystaliśmy z atrybutu `is`. Ze względu na fakt, że to pakiet Vue Router będzie odpowiedzialny za wybór komponentu wyświetlanego przez element router-view, jestem w stanie uprościć obiekt konfiguracji komponentu, usuwając właściwość `components`, właściwości obliczane i wszystkie instrukcje `import`. W wyniku tych zmian odpowiedzialność związana z wyświetlaniem treści przez element router-view spoczywa na pakiecie Vue Router, co daje efekt jak na rysunku 22.2 (przyciski *Utwórz nowy* i *Edytuj* jeszcze nie działają, ale za chwilę je obsłużymy).

ID	Nazwa	Kategoria	Cena		
1	Kajak	Sporty wodne	275	<button>Edytuj</button>	<button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button>	<button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button>	<button>Usuń</button>
4	Chorągiewki narożne	Piłka nożna	34.95	<button>Edytuj</button>	<button>Usuń</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button>	<button>Usuń</button>
6	Myśląca czapeczka	Szachy	16	<button>Edytuj</button>	<button>Usuń</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button>	<button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button>	<button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button>	<button>Usuń</button>

**Utwórz nowy**

Rysunek 22.2. Pakiet Vue Router w praktyce

■ **Wskazówka** W trakcie procesu dodawania lub zmiany konfiguracji trasowania nie zawsze otrzymasz odpowiedzi, jakich się spodziewałeś. W takiej sytuacji zawsze warto zacząć od odświeżenia przeglądarki w celu uzyskania aktualnej kopii aplikacji, co często rozwiązuje dany problem.

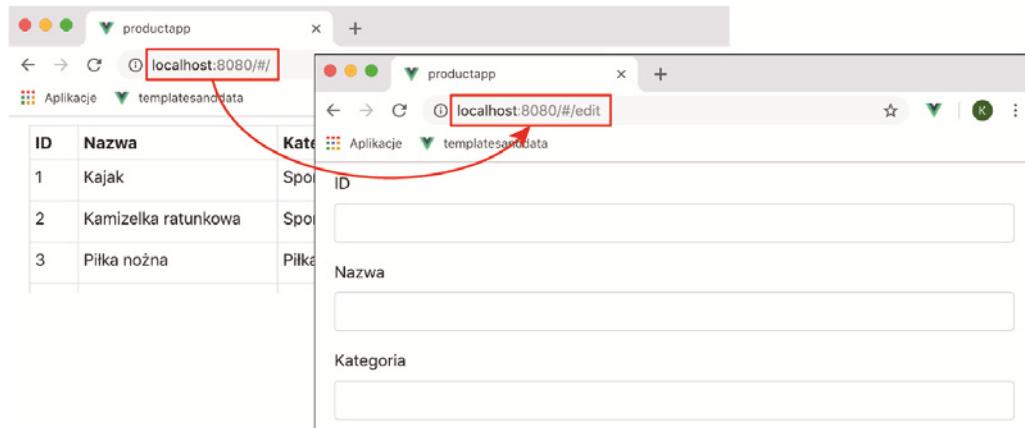
Choć aplikacja wygląda jak poprzednio, pojawiła się istotna różnica, widoczna w pasku adresu przeglądarki. Przejście pod adres `http://localhost:8080` spowoduje w istocie odwiedzenie następującego adresu:

`http://localhost:8080/#/`

Zwróć uwagę na ostatnią część adresu, czyli `#/`. Ostrożnie zmień adres i przejdź pod następujący URL:

`http://localhost:8080/#/edit`

Jest to ten sam adres co wyświetlany przed chwilą, ale zakończony słowem `edit`. Wciśnij `Enter`, a zobaczyś, że treść wyświetlana w przeglądarce ulegnie zmianie (rysunek 22.3).



Rysunek 22.3. Efekt zmiany adresu URL

Zamiast zastosować właściwości magazynu danych w celu wybrania wyświetlanego komponentu, Vue Router korzysta z adresu URL. Część adresu występująca po znaku `#` nawiązuje do konfiguracji zdefiniowanej w listingu 22.4.

```
...
routes: [
 { path: "/", component: ProductDisplay },
 { path: "/edit", component: ProductEditor}
]
...
```

Wartość określona we właściwości `path` odnosi się do części adresu URL występującej za znakiem `#`. Vue Router obserwuje aktualny adres URL i w momencie zmiany wybiera komponent do wyświetlenia w elemencie `router-view`, analizując elementy konfiguracji `routes`, a dokładniej — właściwość `path`. W efekcie zostanie wyświetlony komponent zdefiniowany we właściwości `component`, którego właściwość `path` jest zgodna z bieżącym adresem URL.

## Nawigowanie do innych adresów URL

Aby zmienić wyświetlany komponent, muszę zmienić adres URL przeglądarki, aby w ten sposób wyzwolić działanie pakietu Vue Router. Do nawigowania do nowych adresów URL służy specjalna funkcja, którą przedstawiam w listingu 22.7, aktualizując komponent ProductDisplay.

*Listing 22.7. Nawigowanie z poziomu kodu w pliku src/components/ProductDisplay.vue*

```
...
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 useStripedTable: state => state.prefs.stripedTable
 }),
 ...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
 })
 },
 methods: {
 editProduct(product) {
 this.selectProduct(product);
 //this.selectComponent("editor");
 this.$router.push("/edit");
 },
 createNew() {
 this.selectProduct();
 //this.selectComponent("editor");
 this.$router.push("/edit");
 },
 ...mapMutations({
 selectProduct: "selectProduct",
 //selectComponent: "nav/selectComponent",
 setEditButtonColor: "prefs/setEditButtonColor",
 setDeleteButtonColor: "prefs/setDeleteButtonColor"
 }),
 ...mapActions({
 deleteProduct: "deleteProductAction"
 })
 },
 created() {
 this.setEditButtonColor(false);
 this.setDeleteButtonColor(false);
 }
}
</script>
...
```

Wynikiem dodania właściwości router w listingu 22.7 jest udostępnienie mechanizmów Vue Routera wszystkim komponentom aplikacji za pomocą zmiennej \$router, analogicznie jak w przypadku magazynu danych Vuex i właściwości \$store. Właściwość \$router zwraca obiekt, który definiuje metody nawigacji opisane w tabeli 22.3.

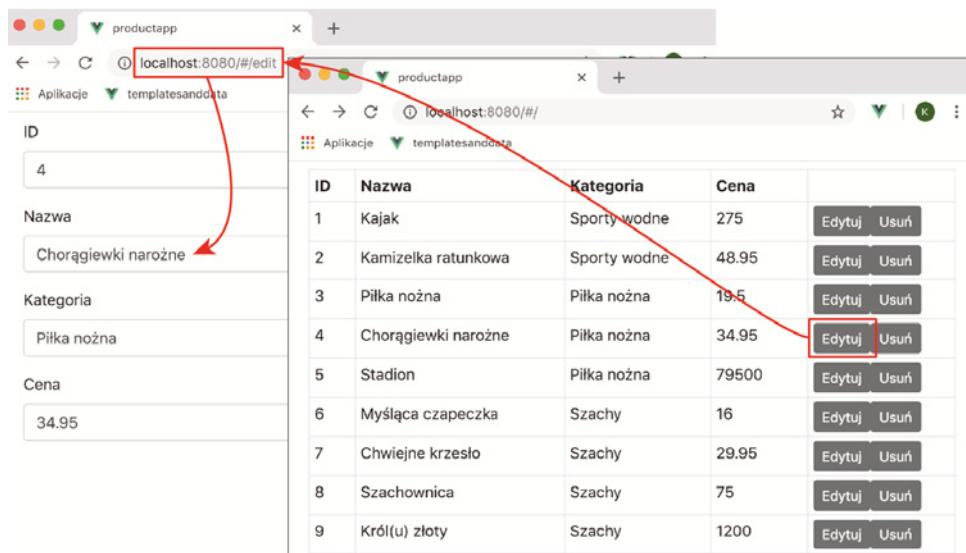
**Tabela 22.3. Metody nawigacji mechanizmu Vue Routera**

Nazwa	Opis
push(location)	Ta metoda powoduje przejście pod podany adres URL. Metoda akceptuje opcjonalne argumenty wywołania zwrotnego, wywoływanie w momencie zakończenia operacji lub w przypadku błędu.
replace(location)	Ta metoda wykonuje to samo zadanie co metoda push, ale nie pozostawia wpisu w historii przeglądarki (więcej informacji na ten temat już za chwilę).
back()	Ta metoda przechodzi do poprzedniego adresu URL w historii przeglądarki.
forward()	Ta metoda przechodzi do następnego adresu URL w historii przeglądarki.

Różnica między tymi metodami polega na tym, że przejście za pomocą metody push pozostawi wpis w historii przeglądarki — kliknięcie przycisku wstecz spowoduje powrót do poprzedniej trasy. Metoda replace zmienia adres URL bez zmiany w historii przeglądarki, przez co przejście wstecz spowoduje wyjście z aplikacji Vue.js. W listingu wyłączam metodę selectComponent, która zastosowała mutację do magazynu danych, i zamieniam ją na wywołanie metody push Vue Routera w celu przejścia do adresu URL /edit.

```
...
this.$router.push("/edit");
...
```

Teraz, po kliknięciu przycisku *Utwórz nowy* lub *Edytuj*, Vue Router nakaże przeglądarce przejść pod adres #/edit. Trasowanie URL to proces dwuetapowy. Metoda push zmienia adres URL, obserwowany stale przez Vue Routera, i wykorzystuje go do zmiany wyświetlanego komponentu. Innymi słowy, kliknięcie przycisku zmieni adres URL, a to spowoduje zmianę wyświetlanego komponentu (rysunek 22.4).



**Rysunek 22.4. Przechodzenie po widokach aplikacji**

## Nawigacja przy użyciu elementów HTML

Metody dostępne w obiekcie \$router to niejedyny sposób na poruszanie się w aplikacji. Vue Router obsługuje także własne elementy HTML, które wyzwalają nawigację w momencie ich kliknięcia. Metody z obiektu \$router i własny element HTML mogą się łączyć w komponencie (listing 22.8).

**Listing 22.8.** Dodawanie nawigacji według adresów URL w pliku src/components/ProductEditor.vue

```
<template>
 <div>
 <div class="form-group">
 <label>ID</label>
 <input class="form-control" v-model="product.id" />
 </div>
 <div class="form-group">
 <label>Nazwa</label>
 <input class="form-control" v-model="product.name" />
 </div>
 <div class="form-group">
 <label>Kategoria</label>
 <input class="form-control" v-model="product.category" />
 </div>
 <div class="form-group">
 <label>Cena</label>
 <input class="form-control" v-model.number="product.price" />
 </div>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="save">
 {{ editing ? "Zapisz" : "Utwórz" }}
 </button>
 <router-link to="/" class="btn btn-secondary">Anuluj</router-link>
 </div>
 </div>
</template>
<script>

let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 // this.$store.commit("nav/selectComponent", "table");
 this.$router.push("/");
 this.product = {};
 },
 // cancel() {
 // this.$store.commit("selectProduct");
 // this.$store.commit("nav/selectComponent", "table");
 // this.$router.push("/");
 // },
 selectProduct(selectedProduct) {
 if (selectedProduct == null) {

```

```

 this.editing = false;
 this.product = {};
 } else {
 this.editing = true;
 this.product = {};
 Object.assign(this.product, selectedProduct);
 }
}
},
created() {
 unwatcher = this.$store.watch(state =>
 state.selectedProduct, this.selectProduct);
 this.selectProduct(this.$store.state.selectedProduct);
},
beforeDestroy() {
 unwatcher();
}
}
</script>

```

W szablonie komponentu korzystam z elementu `router-link`, aby utworzyć element HTML, który wyzwoli nawigację w momencie kliknięcia. Adres URL, do którego przejdziemy, jest określany za pomocą atrybutu:

```
...
<router-link to="/" class="btn btn-secondary">Anuluj</router-link>
...
```

W tym przypadku określiłem trasę `/`, która stanowi pierwszą, domyślną trasę w konfiguracji. Po przetworzeniu i wyświetleniu szablonu element kotwicy (a) przyjmie następującą postać:

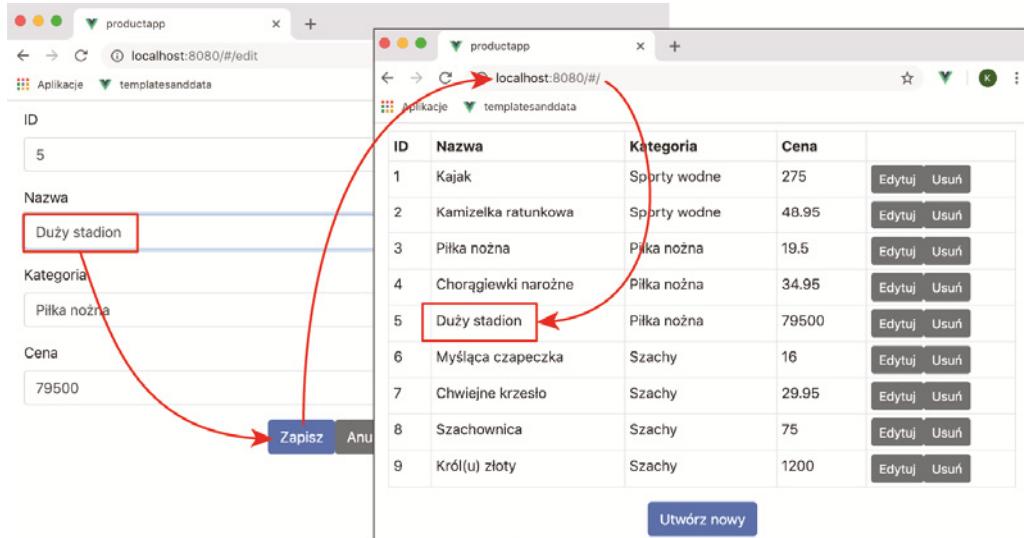
```
...
Anuluj
...
```

Atrybut `href` jest określony w taki sposób, aby przeglądarka przeszła pod adres względny wobec znaku `#` z adresem URL. Nie powinniśmy ręcznie uwzględniać go w przeglądarce, ponieważ istnieją inne metody implementacji nawigacji przy użyciu adresów URL, co opiszę za chwilę. Framework Bootstrap, z którego korzystam w tej książce, obsługuje stylowanie elementów w taki sposób, że wyglądają jak przyciski. Dzięki temu łączne może zastąpić przycisk bez wizualnej różnicy dla użytkownika. Skoro nawigacja URL jest obsługiwana bezpośrednio z poziomu elementu kotwicy, mogę usunąć metodę `cancel`.

Nie każdą operację związaną z nawigacją można wykonać z poziomu elementu HTML, ponieważ niektóre dodatkowe zadania muszą być wykonywane w odpowiedzi na działania użytkownika (tak jak pokazałem w metodzie `save`). Przyciski `Zapisz/Utwórz nowy` w szablonie komponentu nie mogą być zastąpione elementem `router-link`, ponieważ dane wprowadzone przez użytkownika muszą być wysłane do usługi sieciowej. W takiej sytuacji należy skorzystać z metody `$router.push`, co pokazano w listingu. Zmiany z listingu 22.8 pozwalają użytkownikowi przejść z edytora do widoku tabeli na dwa sposoby — przez kliknięcie przycisku `Zapisz/Utwórz nowy` w celu zapisania zmian lub przez kliknięcie przycisku `Anuluj` w celu ich anulowania (rysunek 22.5).

## Omówienie i konfiguracja dopasowania tras URL

Przykłady z poprzedniego podrozdziału zilustrowały w krótki sposób zasadę działania trasowania URL, włączając w to definiowanie tras i różne metody przemieszczania się po aplikacji. W kolejnych punktach zagłębię się nieco bardziej w ten temat, wyjaśniając, jak zmienić format adresów URL, aby stały się bardziej przyjazne dla użytkownika. Omówię także różne sposoby określania tras w celu dopasowania do nich adresów URL.



Rysunek 22.5. Przejście z powrotem do widoku produktów

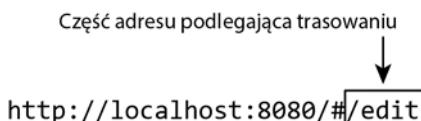
## Omówienie dopasowania i formatowania adresów URL

Vue Router analizuje bieżący adres URL i przetwarza listę tras w zbiorze danych konfiguracji do momentu, aż znajdzie dopasowanie. Aby dopasować trasę, adres URL musi zawierać tę samą liczbę segmentów. Każdy segment musi z kolei zawierać wartość określona w konfiguracji trasowania. Oto trasy, które zdefiniowałem w pliku `src/router/index.js` (listing 22.4):

```
...
routes: [
 { path: "/", component: ProductDisplay },
 { path: "/edit", component: ProductEditor }
]
...

```

Analizując trasy aplikacji, należy pamiętać, że są one dopasowywane według kolejności definicji, a system trasowania analizuje jedynie część całego adresu. Część adresu URL używanego przez system trasowania następująca po znaku # jest znana pod nazwą **fragmentu** lub **kotwicy nazwanej** (ang. *named anchor*). Adres URL `http://localhost:8080/#/edit` dopasuje trasę o ścieżce /edit (rysunek 22.6).



Rysunek 22.6. Fragment w adresie URL

Pierwotnie fragmenty były używane do wskazania konkretnych miejsc w dokumencie HTML, aby użytkownik mógł od razu przenieść się w docelowe miejsce na rozbudowanej stronie. W aplikacjach Vue.js fragmenty URL są używane do trasowania, ponieważ można zmieniać je bez wyzwolenia w przeglądarce procesu wysłania nowego żądania HTTP do serwera (a tym samym bez utraty stanu aplikacji).

## Stosowanie API historii HTML5 do trasowania

Fragmenty URL są obsługiwane przez wszystkie przeglądarki, jednak powstałe w wyniku całego procesu adresy URL mogą być mylące dla użytkowników. Jedną zalet trasowania URL jest możliwość przechodzenia do poszczególnych części aplikacji przez zmianę adresu URL. Proces ten może być dość podatny na błędy, jeśli znak # zostanie pominięty. W sytuacji gdy użytkownik przejdzie pod adres /edit zamiast #/edit, przeglądarka założy, że użytkownik chce przejść pod nowy adres URL, przez co wyśle nowe żądanie HTTP, co doprowadzi do opuszczenia aplikacji Vue.js.

Lepszym rozwiązaniem jest skonfigurowanie technologii Vue Router tak, aby korzystała z API historii HTML5, które pozwala na obsługę eleganckich i wyszukanych adresów URL. Niestety API to nie jest obsługiwane przez starsze przeglądarki. Na szczęście w przeglądarkach, które nie obsługują API historii, Vue Router automatycznie skorzysta z fragmentów. W listingu 22.9 aktualizuję konfigurację trasowania tak, aby Vue Router korzystał z API historii.

**Listing 22.9.** Włączanie API historii w pliku src/router/index.js

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/",
 component: ProductDisplay
 },
 {
 path: "/edit",
 component: ProductEditor
 }
]
})
```

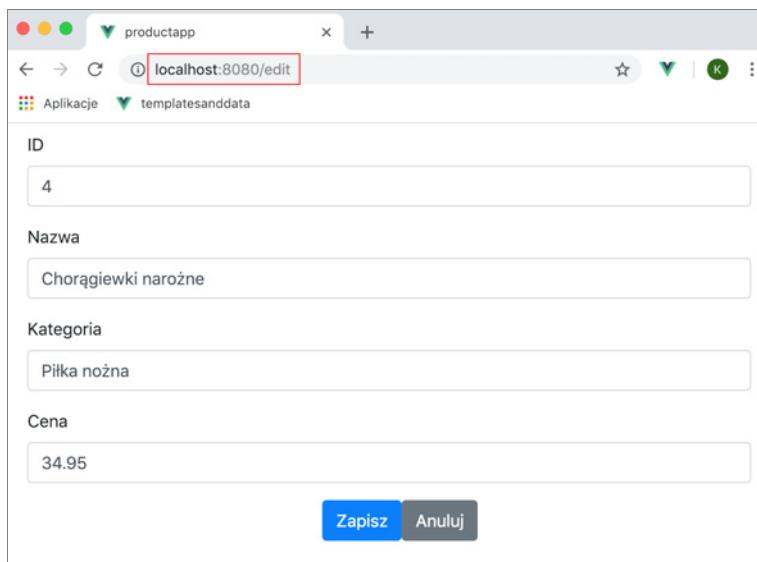
Dodana przed chwilą właściwość mode pozwala na określenie mechanizmu zastosowanego w trasowaniu URL (tabela 22.4).

**Tabela 22.4.** Wartości konfiguracji właściwości mode

Nazwa	Opis
hash	Ten tryb wprowadza do trasowania fragmenty URL, które dają najszerze wsparcie wśród przeglądarek, ale powodują powstawanie dziwnych adresów URL. Jest to wartość domyślna tej właściwości.
history	Ten tryb wprowadza API historii, które pozwala na tworzenie naturalnych adresów URL, jednak nie jest wspierane przez starsze przeglądarki.

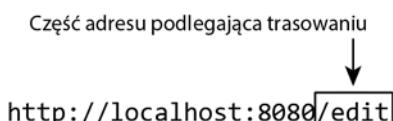
W listingu użyłem wartości history, dzięki której Vue Router skorzysta z API historii. Odśwież przeglądarkę i kliknij jeden z przycisków *Edytuj*, aby zobaczyć zmianę (rysunek 22.7).

- **Wskazówka** W wielu przykładach zawartych w tym rozdziale konieczne może być odświeżenie przeglądarki lub ręczne przejście pod adres <http://localhost:8080>.



Rysunek 22.7. Zastosowanie API historii do trasowania

Zamiast do adresu URL zawierającego fragment (np. <http://localhost:8080/#/edit>) aplikacja przechodzi do adresu <http://localhost:8080/edit>. API historii umożliwia przejście do innego, pełnego adresu URL bez wyzwalania odświeżenia przeglądarki, co pozwala zachować efekt przejścia do innej trasy w aplikacji (rysunek 22.8).



Rysunek 22.8. Efekt użycia API historii

- **Uwaga** W przypadku przeglądarek, które nie obsługują API historii (np. starszych wersji Internet Explorera), Vue Router automatycznie skorzysta z ustawienia hash, co spowoduje zastosowanie fragmentu URL w celu zdefiniowania trasy. Aby wyłączyć to zachowanie, możesz ustawić właściwość fallback na false w obiekcie konfiguracji trasowania.

## Definicja trasy przechwytyjącej wszystko (catchall)

Zastosowanie API historii wymaga wykonania dodatkowej pracy w celu zapewnienia właściwego wrażenia u użytkownika, który przejdzie bezpośrednio pod adres URL interpretowany przez aplikację, ale niezwiązany z żadnym dokumentem HTML. W przykładowej aplikacji użytkownik może wpisać adres <http://localhost:8080/edit> w pasku adresu, co spowoduje wysłanie żądania HTTP w sprawie dokumentu o nazwie *edit*. Na serwerze nie ma takiego dokumentu, ale serwer deweloperski webpack został skonfigurowany tak, aby na każde żądanie odpowiadał plikiem *index.html*.

Nie ma znaczenia, który adres URL znalazł się w żądaniu — jeśli nie ma dla niego treści, serwer zawsze zwróci zawartość pliku *index.html*, a nie kod 404 — *Nie znaleziono (Not Found)*. Jest to użyteczne, gdy użytkownik wysyła żądanie pod adres URL powiązany z jedną z tras aplikacji (np. */edit*), ale spowoduje wygenerowanie pustego dokumentu, gdy adres URL nie zostanie powiązany z żadną trasą.

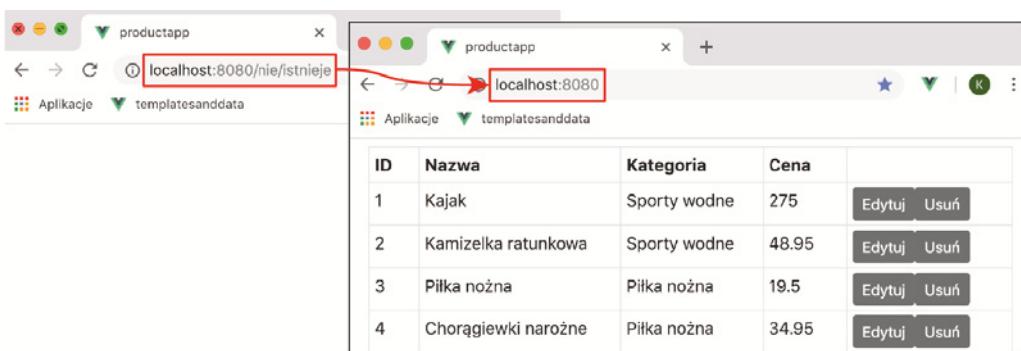
- Ostrzeżenie** Musisz upewnić się, że Twój produkcyjny serwer HTTP zwraca zawartość pliku `index.html`, jeśli korzystasz z API historii do trasowania. Zespół Vue.js przygotował pod adresem `http://router.vuejs.org/en/essentials/history-mode.html`/instrukcje dla najczęściej używanych serwerów produkcyjnych.

Aby rozwiązać ten problem, dodaję trasę przechwytyującą wszystko — `catchall` — która dopasuje wszystkie żądania i przekieruje je pod podany adres URL (listing 22.10).

**Listing 22.10.** Tworzenie trasy typu `catchall` w pliku `src/router/index.js`

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/",
 component: ProductDisplay
 },
 {
 path: "/edit",
 component: ProductEditor
 },
 {
 path: "*",
 redirect: "/"
 }
]
})
```

Właściwość `path` tej trasy zawiera gwiazdkę (\*), co oznacza, że dopasuje dowolny adres URL. Zamiast właściwości `component` trasa zawiera właściwość `redirect`, która poinstruktuje Vue.js o konieczności przekierowania na podany adres URL. Efekt tej funkcji jest połączony z mechanizmem awaryjnym serwera HTTP, dzięki czemu żądania, dla których nie ma żadnej treści, zostaną obsłużone przez zaserwowanie treści pliku `index.html`. Jeśli adres URL nie odpowiada żadnej trasie, przeglądarka zostanie przekierowana na adres /. Aby przetestować tę funkcję, otwórz okno przeglądarki i przejdź pod adres `http://localhost:8080/nie/istnieje`. Jak pokazuje rysunek 22.9, przeglądarka wyświetli aplikację, mimo że żądany adres URL nie odpowiada żadnej trasie aplikacji.



**Rysunek 22.9.** Efekt zastosowania trasy typu `catchall`

- **Wskazówka** Przekierowania mogą być definiowane także w formie funkcji, co pozwala na dołączenie aspektów żądanego przez użytkownika adresu URL do adresu URL przekierowania. Więcej informacji znajdziesz w rozdziale 23.

Po uruchomieniu aplikacji Vue.js Vue Router analizuje bieżący URL i rozpoczyna analizę kolejnych tras, aby znaleźć dopasowanie. Pierwsza i druga trasa nie pasują do aktualnego adresu URL, ponieważ adresem URL nie jest ani `/`, ani `/edit`. Vue Router osiąga ostatni adres, który przechwytuje wszystko, a zatem zostaje wdrożone przekierowanie na trasę `/`. Proces dopasowania ponownie się rozpoczyna, ale tym razem zadziała pierwsza z tras. Spowoduje to wyświetlenie komponentu `ProductDisplay`.

## **Stosowanie aliasu trasy**

Jednym z problemów w stosowaniu przekierowań jest fakt, że użytkownik może zauważać zmianę adresu URL w przeglądarce na URL przekierowany, co może być mylące. Alternatywą jest utworzenie aliasu trasy, który pozwoli na dopasowanie więcej niż jednego adresu URL bez konieczności przekierowania. W listingu 22.11 do konfiguracji trasowania dodajemy alias, aby zmienić sposób, w jaki są obsługiwane niedopasowane adresy URL.

**Listing 22.11.** Zastosowanie aliasu trasy w pliku `src/router/index.js`

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 { path: "/", component: ProductDisplay, alias: "/list" },
 { path: "/edit", component: ProductEditor } ,
 { path: "*", redirect: "/" }
]
})
```

Właściwość `alias` służy do utworzenia aliasu dla trasy, dzięki czemu jesteśmy w stanie dopasować wiele adresów URL bez konieczności wykonywania przekierowania. Utworzony przeze mnie alias wprowadza adres URL `/list` jako alias głównego adresu URL. Można go sprawdzić, przechodząc pod adres `http://localhost:8080/list`. Jak widać na rysunku 22.10, komponent `ProductDisplay` jest wyświetlany bez modyfikacji bieżącego adresu URL.

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>

**Rysunek 22.10.** Zastosowanie aliasu trasowania

## Pobieranie danych trasowania w komponentach

Poza właściwością \$router pakiet Vue Router dostarcza także właściwość \$route, która opisuje aktualną trasę i może być użyta do dostosowania treści lub zachowania komponentów. W ramach przykładu dodam nową trasę do przykładowej aplikacji (listing 22.12).

*Listing 22.12. Dodawanie trasy do pliku src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/",
 component: ProductDisplay,
 alias: "/list"
 },
 {
 path: "/edit",
 component: ProductEditor
 },
 {
 path: "/create",
 component: ProductEditor
 },
 {
 path: "*",
 redirect: "/"
 }
]
})
```

Teraz nowa trasa dopasowuje adres /create i korzysta z komponentu ProductEditor, co oznacza, że dwa różne adresy URL — /edit i /create — doprowadzą aplikację do mechanizmu edytora. W listingu 22.13 zamieniam przyciski w szablonie komponentu ProductDisplay, korzystając z elementów router-link, prowadzących do adresów /edit i /create.

*Listing 22.13. Ustalanie przeznaczenia tras w pliku src/components/ProductDisplay.vue*

```
...
<template>
 <div>
 <table class="table table-sm table-bordered" v-bind:class="tableClass">
 <tr>
 <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
 </tr>
 <tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <router-link to="/edit" v-bind:class="editClass"
 class="btn btn-sm">
 Edytuj
 </router-link>
 </td>
 </tr>
 </tbody>
 </table>
 </div>
</template>
```

```

<button class="btn btn-sm"
 v-bind:class="deleteClass"
 v-on:click="deleteProduct(p)">
 Usuń
</button>
</td>
</tr>
<tr v-if="products.length == 0">
 <td colspan="5" class="text-center">Brak danych</td>
</tr>
</tbody>
</table>
<div class="text-center">
 <router-link to="/create" class="btn btn-primary">
 Utwórz nowy
 </router-link>
</div>
</div>
</template>
...

```

Zwróć uwagę, że komponent „nie wie”, jaki będzie efekt przejścia pod adres `/edit` lub `/create`. To zachowanie przypomina zastosowanie magazynu danych do koordynowania komponentów, co opisałem w rozdziale 21., z tą różnicą, że zmiana w konfiguracji jest możliwa nawet tylko dzięki edycji konfiguracji trasowania. Teraz, gdy mamy do wyboru dwa różne adresy URL, które wyświetlają komponent `ProductEditor`, mogę skorzystać z właściwości `$route`, aby sprawdzić, który z nich został użyty, i zmienić treść wyświetlaną użytkownikowi, jak w listingu 22.14.

*Listing 22.14. Dostęp do informacji na temat trasy w pliku `src/components/ProductEditor.vue`*

```

<template>
<div>
 <h3 class="btn-primary text-center text-white p-2">
 {{ editing ? "Edytuj" : "Utwórz" }}
 </h3>
 <div class="form-group">
 <label>ID</label>
 <input class="form-control" v-model="product.id" />
 </div>
 <div class="form-group">
 <label>Nazwa</label>
 <input class="form-control" v-model="product.name" />
 </div>
 <div class="form-group">
 <label>Kategoria</label>
 <input class="form-control" v-model="product.category" />
 </div>
 <div class="form-group">
 <label>Cena</label>
 <input class="form-control" v-model.number="product.price" />
 </div>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="save">
 {{ editing ? "Zapisz" : "Utwórz" }}
 </button>
 <router-link to="/" class="btn btn-secondary">Anuluj</router-link>
 </div>
</div>

```

```

</template>
<script>

let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$router.push("/");
 this.product = {};
 },
 selectProduct(selectedProduct) {
 if (this.$route.path == "/create") {
 this.editing = false;
 this.product = {};
 } else {
 this.product = {};
 Object.assign(this.product, selectedProduct);
 this.editing = true;
 }
 }
 },
 created() {
 unwatcher = this.$store.watch(state =>
 state.selectedProduct, this.selectProduct);
 this.selectProduct(this.$store.state.selectedProduct);
 },
 beforeDestroy() {
 unwatcher();
 }
}
</script>

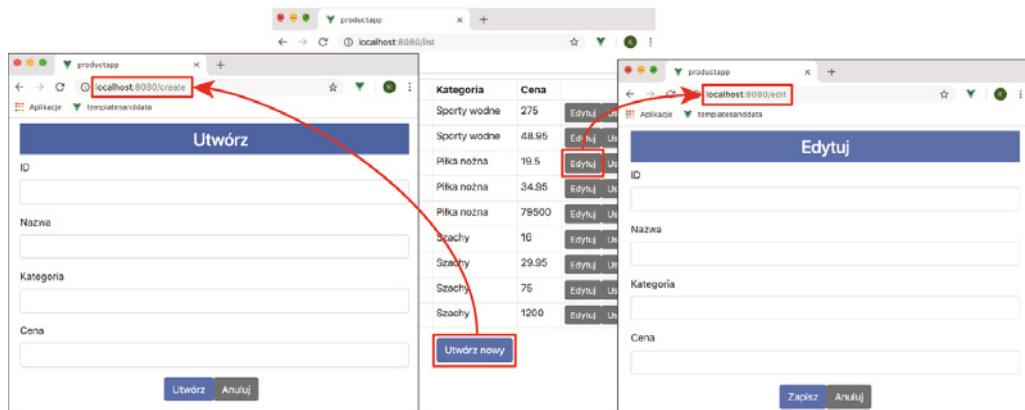
```

W momencie wywołania metody `selectProduct` komponent sprawdza obiekt `$route`, aby pobrać szczegóły na temat trasy. Właściwość `$route` otrzymuje obiekt, który opisuje aktualną trasę. Tabela 22.5 zawiera najbardziej przydatne właściwości tego obiektu.

**Tabela 22.5. Przydatne właściwości obiektu `$route`**

Nazwa	Opis
<code>name</code>	Ta właściwość zwraca nazwę trasy, co opisuję w punkcie „Tworzenie tras nazwanych”.
<code>path</code>	Ta właściwość zwraca ścieżkę adresu URL, np. <code>/edit/4</code> .
<code>params</code>	Ta właściwość zwraca obiekt słownika parametrów dopasowanych przez dynamiczną trasę (por. punkt „Dynamiczne dopasowywanie tras”).
<code>query</code>	Ta właściwość zwraca obiekt słownika zawierającego wartości łańcucha zapytania (ang. <i>query string</i> ). Na przykład dla adresu URL <code>/edit/4?validate=true</code> właściwość <code>query</code> zwróci obiekt zawierający właściwość <code>validate</code> o wartości <code>true</code> .

W tym listingu korzystam z właściwości path, aby określić, czy został użyty adres /create, czy też /edit. W zależności od tej informacji przebiega proces konfiguracji komponentu. Aby uwidoczyć tę różnicę, dodaję element h3, który wyświetla nagłówek za pomocą interpolacji tekstu. Aby sprawdzić zmiany, kliknij przycisk *Utwórz nowy* lub *Edytuj*, wyświetlane przez komponent ProductDisplay, lub przejdź bezpośrednio pod adres <http://localhost:8080/create> lub <http://localhost:8080/edit> (rysunek 22.11).



Rysunek 22.11. Reakcja na różne trasy w pojedynczym komponencie

## Dynamiczne dopasowywanie tras

Problem, który pojawił się wraz z wprowadzeniem przed chwilą zmian, polega na tym, że adres URL /edit wskazuje konieczność przejścia do komponentu ProductEditor, ale nie wiadomo, który obiekt należy poddać edycji.

Rozwiążaniem jest **segment dynamiczny**, który jest dodawany do trasy, gdy komponent musi pobrać dane z samego adresu URL. W listingu 22.15 rozszerzam konfigurację trasowania, dzięki czemu zawiera ona segment dynamiczny.

*Listing 22.15. Zastosowanie segmentu dynamicznego w pliku src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/",
 component: ProductDisplay,
 alias: "/list"
 },
 {
 path: "/edit/:id",
 component: ProductEditor
 },
 {
 path: "/create",
 component: ProductEditor
 },
 {
 path: "*",
 }
]
});
```

```

 redirect: "/"
 }
]
})
}

```

Zmienne segmentu są definiowane przez jego poprzedzenie dwukropkiem:

```

...
{ path: "/edit/:id", component: ProductEditor },
...

```

Właściwość path tej trasy zawiera dwa segmenty. Pierwszy dopasowuje adresy URL, które rozpoczynają się od ciągu /edit, jak w poprzednich przykładach. Drugi segment jest dynamiczny i dopasowuje dowolny segment URL, który zostanie przypisany do zmiennej id. Dzięki takiej konstrukcji zostaną przechwycone dwolne adresy URL o dwóch segmentach, w których to adresach pierwszy segment to /edit, np. /edit/10.

Aby obsługiwać nowy adres URL i jego dynamiczny segment, aktualizuję elementy router-link w szablonie komponentu ProductDisplay (listing 22.16).

**Listing 22.16.** Konfiguracja segmentu dynamicznego w pliku src/components/ProductDisplay.vue

```

...
<template>
 <div>
 <table class="table table-sm table-bordered" v-bind:class="tableClass">
 <tr>
 <th>ID</th><th>Nazwa</th><th>Kategoria</th><th>Cena</th><th></th>
 </tr>
 <tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <router-link v-bind:to="'/edit/' + p.id "
 v-bind:class="editClass" class="btn btn-sm">
 Edytuj
 </router-link>
 <button class="btn btn-sm"
 v-bind:class="deleteClass"
 v-on:click="deleteProduct(p)">
 Usuń
 </button>
 </td>
 </tr>
 <tr v-if="products.length == 0">
 <td colspan="5" class="text-center">Brak danych</td>
 </tr>
 </tbody>
 </table>
 <div class="text-center">
 <router-link to="/create" class="btn btn-primary">
 Utwórz nowy
 </router-link>
 </div>
 </div>
</template>
...

```

Kliknięcie przycisku *Edytuj* spowoduje przejście pod adres URL, który zawiera właściwość `id` powiązanego obiektu. Kliknięcie przycisku przy produkcie *Stadion* spowoduje przejście pod adres `/edit/5`.

Komponenty mogą korzystać ze zmiennych z segmentów dynamicznych za pomocą właściwości `$route.params`. W listingu 22.17 modyfikuję komponent `ProductEditor`, aby pobrać obiekt produktu z magazynu danych na podstawie segmentu dynamicznego `id`.

**Listing 22.17.** Zastosowanie wartości segmentu dynamicznego w pliku `src/components/ProductEditor.vue`

```
...
<script>
let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$router.push("/");
 this.product = {};
 },
 selectProduct() {
 if (this.$route.path == "/create") {
 this.editing = false;
 this.product = {};
 } else {
 let productId = this.$route.params.id;
 let selectedProduct = this.$store.state.products.find(p => p.id == productId);
 this.product = {};
 Object.assign(this.product, selectedProduct);
 this.editing = true;
 }
 }
 },
 created() {
 unwatcher = this.$store.watch(state => state.products,
 this.selectProduct);
 this.selectProduct();
 },
 beforeDestroy() {
 unwatcher();
 }
}
</script>
...
```

W metodzie `selectProduct` pobieram wartość segmentu `id` za pomocą obiektu `$route`, a następnie uzyskuję obiekt z magazynu danych.

Choć nie łatwo to zauważyc, zmieniłem także obiekt docelowy obserwatora magazynu danych. Komponent `ProductDisplay` nie korzysta już z magazynu danych do przechowywania wyboru użytkownika. W związku z tym obserwator może wydawać się zbędny. Tak jednak nie jest — zmieniamy obserwowany obiekt na tablicę obiektów-produktów.

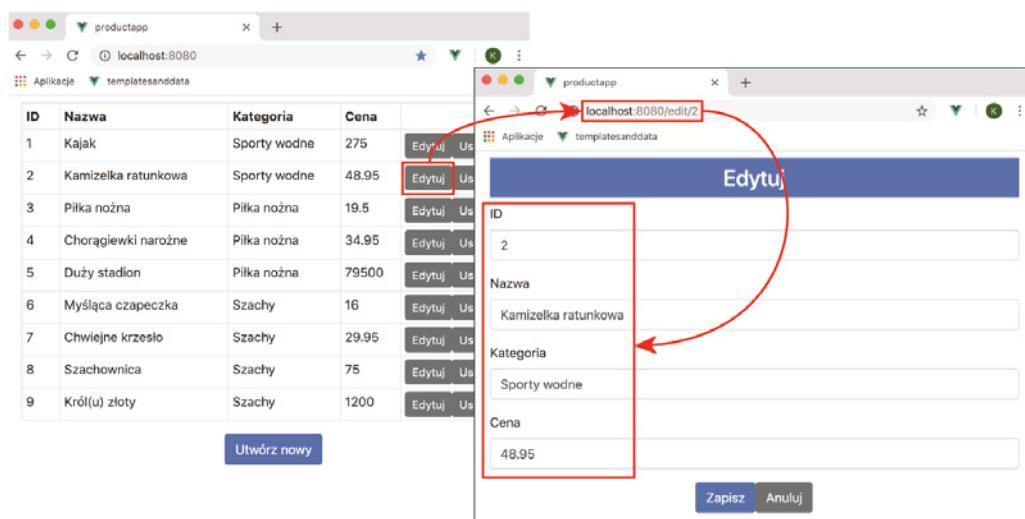
```
...
unwatcher = this.$store.watch(state => state.products, this.selectProduct);
...
```

Użytkownik może przejść bezpośrednio pod adres URL, który doprowadzi nas do edycji obiektu. W związku z tym komponent ProductEditor zostanie wyświetlony przed wypełnieniem magazynu danych treścią otrzymaną z serwera HTTP. Aby upewnić się, że użytkownik widzi niezbędne dane, korzystam z obserwatora magazynu danych, który wywołuje metodę selectProduct.

- **Wskazówka** Pakiet Vue Router umożliwia użycie bardziej eleganckiej metody oczekiwania na dane, opisanej szczegółowo w rozdziale 24.

Teraz, po kliknięciu jednego z przycisków w komponencie ProductDisplay, użytkownik trafi pod adres /edit/4, który można również wprowadzić bezpośrednio w pasku adresu przeglądarki.

Gdy adres URL zostanie dopasowany przez system trasowania, komponent ProductEditor odczyta wartość z segmentu dynamicznego, znajdzie odpowiedni obiekt w magazynie danych, a następnie wyświetli go do edycji (rysunek 22.12).



Rysunek 22.12. Zastosowanie segmentu dynamicznego

## Stosowanie wyrażeń regularnych do dopasowywania adresów URL

Segmente dynamiczne rozszerzają zakres adresów URL, do których trasa zostanie dopasowana. Efektem może być trasa dopasowująca adresy URL, które można obsłużyć później w konfiguracji trasowania. Aby zwiększyć precyzję działania tras, Vue Router obsługuje wyrażenia regularne dla segmentów dynamicznych, które to wyrażenia pozwalają na bardzo dokładną kontrolę dopasowania do adresów.

W listingu 22.18 przeglądám konfigurację trasowania, aby dodać segment dynamiczny z wyrażeniem regularnym i zastosować wyrażenie regularne do istniejącego segmentu id.

*Listing 22.18. Zastosowanie wyrażeń regularnych w pliku src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
```

```

mode: "history",
routes: [
 {
 path: "/",
 component: ProductDisplay,
 alias: "/list"
 },
 {
 path: "/:op(create|edit)/:id(\d+)",
 component: ProductEditor
 },
 // { path: "/create", component: ProductEditor },
 {
 path: "*",
 redirect: "/"
 }
]
})

```

Wyrażenia regularne są określane w nawiasach po nazwie segmentu dynamicznego. Nowy segment dynamiczny nosi nazwę op, a wyrażenie regularne, które wobec niego stosuję, pozwala na dopasowanie segmentów zawierających słowa *create* lub *edit*. Dzięki temu jestem w stanie połączyć dwie trasy w jedną i oznaczyć jako komentarz dedykowaną trasę */create*.

```

...
{ path: "/:op(create|edit)/:id(\d+)", component: ProductEditor },
...

```

Wyrażenie regularne, które zastosowałem wobec segmentu id, dopasuje segmenty składające się tylko z minimum jednej cyfry. Znak d w wyrażeniu musi zostać poprzedzony dwoma wstecznymi ukośnikami (\), aby zapobiec zinterpretowaniu tego znaku po prostu jako litery. Znak plus określa, że liczba cyfr musi być większa lub równa 1.

```

...
{ path: "/:op(create|edit)/:id(\d+)", component: ProductEditor },
...

```

Przedstawiona trasa dopasuje dwa segmenty URL, przy czym pierwszym segmentem URL może być słowo */create* lub */edit*, a drugim — po prostu liczba.

## Definiowanie segmentów opcjonalnych

Trasa przedstawiona w listingu 22.18 nie działa dokładnie tak, jak byśmy tego chcieli, ponieważ nie zostanie dopasowana trasa */create*, podana w komponencie *ProductDisplay* i używana w momencie kliknięcia przycisku *Utwórz nowy*. Segmente dynamiczne mogą być oznaczane jako opcjonalne za pomocą znaku zapytania, który pozwoli na podanie zera lub większej liczby wystąpień danego wyrażenia (w tym przypadku cyfry). W listingu 22.19 dodaję znak zapytania, aby uczynić segment id opcjonalnym.

**Listing 22.19.** Zastosowanie segmentu opcjonalnego w pliku *src/router/index.js*

```

import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/",
 component: ProductDisplay,

```

```

 alias: "/list"
 },
 {
 path: "/:op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "*",
 redirect: "/"
 }
]
})
})

```

Skoro id jest segmentem opcjonalnym, trasa dopasuje dowolny jednosegmentowy adres URL, np. `/edit` lub `/create`, a także adresy dwusegmentowe, gdzie pierwszym segmentem jest ciąg `/create` lub `/edit`, a drugim — jedna lub więcej cyfr.

Ta konfiguracja trasowania dopasuje adres taki jak `/create/10`, który w obecnej wersji komponentu `ProductEditor` zostanie zinterpretowany jako próba edycji obiektu o id równym 10. To pokazuje, jak ważne jest dostosowanie komponentów po wprowadzeniu zmian w konfiguracji trasowania. W listingu 22.20 modyfikuję komponent `ProductEditor`, aby uniknąć tego rodzaju problemu, i stosuję segment dynamiczny w postaci z listingu 22.19.

**Listing 22.20.** Zastosowanie nowego segmentu w pliku `src/components/ProductEditor.vue`

```

...
<script>
let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$router.push("/");
 this.product = {};
 },
 selectProduct() {
 if (this.$route.params.op == "create") {
 this.editing = false;
 this.product = {};
 } else {
 let productId = this.$route.params.id;
 let selectedProduct = this.$store.state.products.find(p => p.id == productId);
 this.product = {};
 Object.assign(this.product, selectedProduct);
 this.editing = true;
 }
 }
 },
 created() {
 unwatcher = this.$store.watch(state => state.products,
 this.selectProduct);
 this.selectProduct();
 },
}

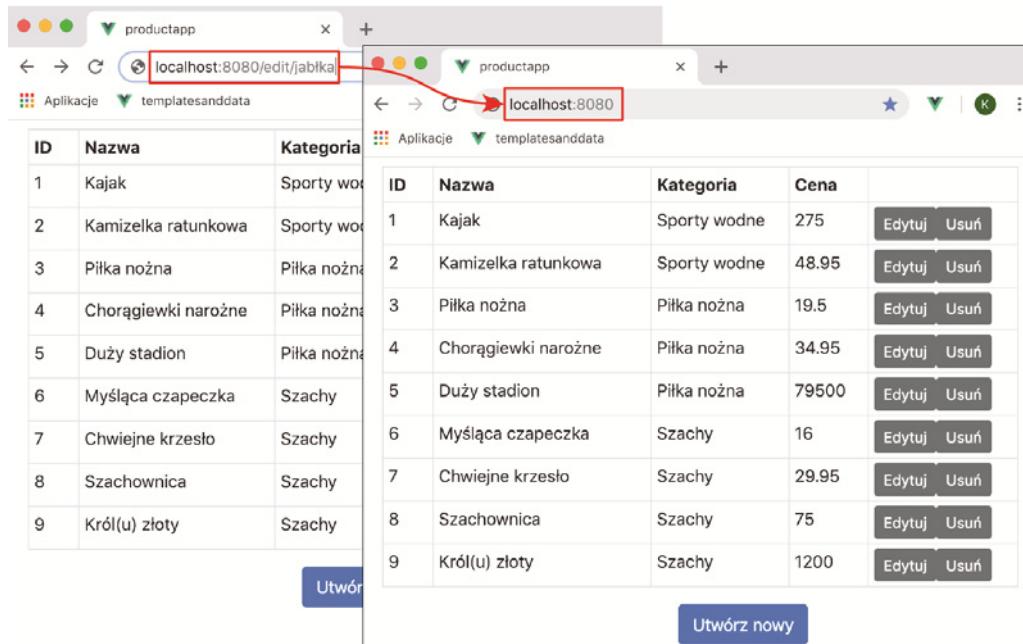
```

```

 beforeDestroy() {
 unwatcher();
 }
 }

```

Efektem zmian wprowadzonych w konfiguracji trasowania jest odrzucenie tras takich jak np. /edit/jabłka przez komponent ProductEditor. System trasowania odrzuci taką trasę i będzie kontynuował przetwarzanie tras aż do osiągnięcia trasy typu catchall, która przekieruje użytkownika pod główny adres aplikacji (rysunek 22.13).



Rysunek 22.13. Zastosowanie wyrażeń regularnych w trasie

## Tworzenie tras nazwanych

Jeśli nie chcesz osadzać adresów URL w kodzie i szablonach komponentu, możesz przypisać nazwy do tras i z nich korzystać. Zaletą tego podejścia jest możliwość łatwej zmiany adresów URL przez aplikację bez konieczności zmiany wszystkich elementów nawigacji i kodu w komponentach (por. rozdział 23.). Problemem w zastosowaniu tras nazwanych jest dziwna składnia. W listingu 22.21 dodaje nazwy do tras, które są związane z komponentami ProductDisplay i ProductEditor.

*Listing 22.21. Nazywanie tras w pliku src/router/index.js*

```

import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 { path: "/products/:id", component: ProductDisplay },
 { path: "/products/:id/edit", component: ProductEditor }
]
});

```

```

routes: [
 {
 name: "table",
 path: "/",
 component: ProductDisplay,
 alias: "/list"
 },
 {
 name: "editor",
 path: "/:op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "*",
 redirect: "/"
 }
]
})
)

```

Właściwość `name` służy do określenia nazwy trasy. W tym przypadku skorzystałem z nazw `table` i `editor`. W listingu 22.22 używam w nawigacji nazwy trasy zamiast adresu URL.

*Listing 22.22. Nawigacja za pomocą nazwy trasy w pliku src/components/ProductEditor.vue*

```

<template>
<div>
 <h3 class="btn-primary text-center text-white p-2">
 {{ editing ? "Edytuj" : "Utwórz"}}
 </h3>
 <div class="form-group">
 <label>ID</label>
 <input class="form-control" v-model="product.id" />
 </div>
 <div class="form-group">
 <label>Nazwa</label>
 <input class="form-control" v-model="product.name" />
 </div>
 <div class="form-group">
 <label>Kategoria</label>
 <input class="form-control" v-model="product.category" />
 </div>
 <div class="form-group">
 <label>Cena</label>
 <input class="form-control" v-model.number="product.price" />
 </div>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="save">
 {{ editing ? "Zapisz" : "Utwórz" }}
 </button>
 <router-link v-bind:to="{name: 'table'}" class="btn btn-secondary">
 Anuluj
 </router-link>
 </div>
</div>
</template>
<script>

let unwatcher;
export default {

```

```

data: function() {
 return {
 editing: false,
 product: {}
 }
},
methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$router.push({
 name: "table"
 });
 this.product = {};
 },
 selectProduct() {
 if (this.$route.params.op == "create") {
 this.editing = false;
 this.product = {};
 } else {
 let productId = this.$route.params.id;
 let selectedProduct = this.$store.state.products.find(p => p.id == productId);
 this.product = {};
 Object.assign(this.product, selectedProduct);
 this.editing = true;
 }
 }
},
created() {
 unwatcher = this.$store.watch(state => state.products,
 this.selectProduct);
 this.selectProduct();
},
beforeDestroy() {
 unwatcher();
}
}
</script>

```

Aby wykonać przejście z użyciem danej trasy za pomocą nazwy, obiekt, który ma właściwość name, jest przekazywany do metody \$router.push lub jest przypisywany do atrybutu do elementu router-link. Wartość właściwości name jest nazwą pożąданej trasy, a dyrektywa v-bind służy do zapewnienia, że Vue.js zinterpretuje wartość atrybutu jako wyrażenie języka JavaScript.

Jeżeli trasa zawiera segmenty dynamiczne, to obiekt zastosowany do nawigacji wprowadza właściwość params, za pomocą której określamy wartości segmentów. W listingu 22.23 zmieniam kod i element nawigacji, aby zastosować nazwy i dostarczyć wartości parametrów.

*Listing 22.23. Nawigacja przy użyciu parametrów w pliku src/components/ProductDisplay.vue*

```

...
<template>
<div>
 <table class="table table-sm table-bordered" v-bind:class="tableClass">
 <tr>
 <th>ID</th>
 <th>Nazwa</th>
 <th>Kategoria</th>
 <th>Cena</th>
 <th></th>

```

```

</tr>
<tbody>
 <tr v-for="p in products" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 <td>
 <router-link v-bind:to="{name: 'editor',
 params: { op: 'edit', id: p.id}}"
 v-bind:class="editClass" class="btn btn-sm">
 Edytuj
 </router-link>
 <button class="btn btn-sm"
 v-bind:class="deleteClass"
 v-on:click="deleteProduct(p)">
 Usuń
 </button>
 </td>
 </tr>
 <tr v-if="products.length == 0">
 <td colspan="5" class="text-center">Brak danych</td>
 </tr>
</tbody>
</table>
<div class="text-center">
 <router-link v-bind:to="{name: 'editor', params: { op: 'create'}}"
 class="btn btn-primary">
 Utwórz nowy
 </router-link>
</div>
</div>
</template>
...

```

Dyrektywa `v-bind` jest niezbędna w przypadku użycia nazwy i przekazywania parametrów za pomocą elementu `router-link`. W przeciwnym razie Vue.js nie będzie w stanie zinterpretować wartości atrybutu jako obiektu JavaScript i potraktuje ją jako adres URL.

## Obsługa zmian w nawigacji

Jeśli zmiana trasy wiąże się z przedstawieniem nowej treści, istniejący komponent zostanie zniszczony, a w zamian zostanie utworzona instancja nowego komponentu, zgodnie z cyklem życia opisanym w rozdziałach 17. i 21. Gdy zmiana trasy prowadzi do wyświetlenia tego samego komponentu, pakiet Vue Router po prostu korzysta ponownie z istniejącego komponentu, powiadamiając go o zmianie. Aby przedstawić ten problem, dodaję w komponencie `ProductEditor` nowe funkcje nawigacyjne, które pozwolą użytkownikowi na przemieszczanie się po obiektach przeznaczonych do edycji (listing 22.24).

**Listing 22.24.** Dodawanie funkcji nawigacyjnych w pliku `src/components/ProductEditor.vue`

```

<template>
 <div>
 <h3 class="btn-primary text-center text-white p-2">
 {{ editing ? "Edytuj" : "Utwórz" }}
 </h3>
 <div class="form-group">

```

```

 <label>ID</label>
 <input class="form-control" v-model="product.id" />
 </div>
 <div class="form-group">
 <label>Nazwa</label>
 <input class="form-control" v-model="product.name" />
 </div>
 <div class="form-group">
 <label>Kategoria</label>
 <input class="form-control" v-model="product.category" />
 </div>
 <div class="form-group">
 <label>Cena</label>
 <input class="form-control" v-model.number="product.price" />
 </div>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="save">
 {{ editing ? "Zapisz" : "Utwórz" }}
 </button>
 <router-link to="{name: 'table'}" class="btn btn-secondary">
 Anuluj
 </router-link>
 <router-link v-if="editing" v-bind:to="nextUrl" class="btn btn-info">
 Następny
 </router-link>
 </div>
</div>
</template>
<script>

let unwatcher;
export default {
 data: function() {
 return {
 editing: false,
 product: {}
 }
 },
 computed: {
 nextUrl() {
 if (this.product.id != null && this.$store.state.products != null) {
 let index = this.$store.state.products
 .findIndex(p => p.id == this.product.id);
 let target = index < this.$store.state.products.length - 1 ?
 index + 1 : 0
 return `/edit/${this.$store.state.products[target].id}`;
 }
 return "/edit";
 }
 },
 methods: {
 async save() {
 await this.$store.dispatch("saveProductAction", this.product);
 this.$router.push({
 name: "table"
 });
 this.product = {};
 },
 },
}

```

```

 selectProduct(route) {
 if (route.params.op == "create") {
 this.editing = false;
 this.product = {};
 } else {
 let productId = route.params.id;
 let selectedProduct = this.$store.state.products.find(p => p.id == productId);
 this.product = {};
 Object.assign(this.product, selectedProduct);
 this.editing = true;
 }
 },
 created() {
 unwatcher = this.$store.watch(state => state.products,
 () => this.selectProduct(this.$route));
 this.selectProduct(this.$route);
 },
 beforeDestroy() {
 unwatcher();
 },
 beforeRouteUpdate(to, from, next) {
 this.selectProduct(to);
 next();
 }
}
</script>

```

Dodałem element router-link, który jest wyświetlany w momencie edycji produktu. Jego celem jest przejście do następnego produktu w magazynie danych. Wyświetlany komponent nie ulega zmianie, co oznacza, że Vue Router nie zniszczy istniejącej instancji komponentu ProductEditor.

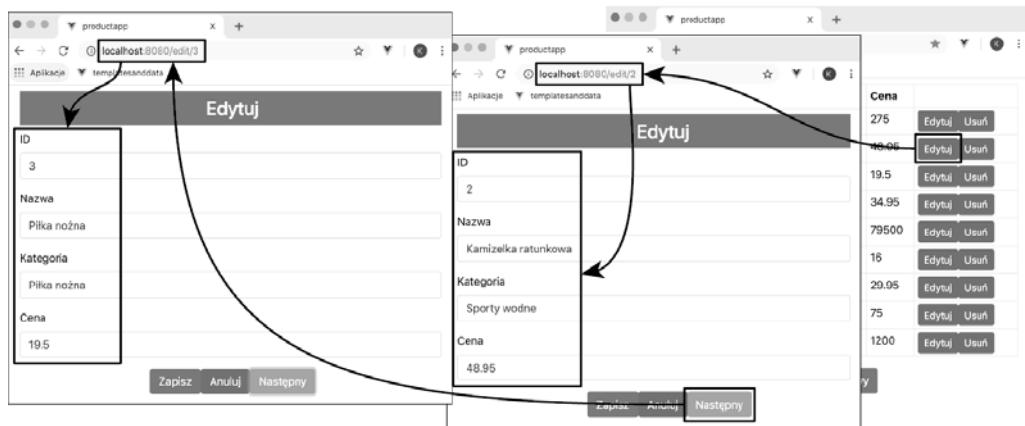
Komponenty mogą implementować metody w celu otrzymywania powiadomień systemu trasowania. Pełen zakres metod opisuję w rozdziale 24. W tym rozdziale to metoda beforeRouteUpdate jest istotna, ponieważ jest ona wywoływana w momencie zmiany trasy bez niszczenia komponentu. Metoda beforeRouteUpdate definiuje trzy parametry, opisane w tabeli 22.6.

**Tabela 22.6. Parametry metody beforeRouteUpdate**

Nazwa	Opis
to	Ten parametr otrzymuje obiekt opisujący trasę, do której wykonywane jest przejście.
from	Ten parametr otrzymuje obiekt opisujący bieżącą trasę, którą aplikacja zamierza opuścić.
next	Ten parametr otrzymuje funkcję, która musi być wywołana, aby pozwolić na przetworzenie powiadomienia przez inne komponenty. Można także skorzystać z niej w celu przejęcia kontroli nad procesem nawigacji, co opisuję w rozdziale 24.

Obiekty przekazane w parametrach to i from definiują ten sam zbiór właściwości co obiekt \$route opisany w tabeli 22.5. W listingu implementacja metody beforeRouteUpdate przekazuje obiekt z parametru to do metody selectProduct, dzięki czemu komponent może zmieniać swój stan. Pamiętaj, że aktywna trasa nie została jeszcze zmieniona, więc nie mogę skorzystać z obiektu \$route, aby zareagować na zmiany. Po przetworzeniu zmiany wywołuję funkcję next, która pozwala komponentom otrzymywać powiadomienia (może wydawać się to dziwne, ale funkcja next potrafi też zapobiec przejściu lub je zmodyfikować, co opisuję w rozdziale 24).

Aby przetestować powiadomienia trasowania przejdź na stronę <http://localhost:8080> i kliknij jeden z przycisków *Edytuj*. Następnie kliknij przycisk *Następny*, aby przejść do kolejnego produktu (rysunek 22.14).



Rysunek 22.14. Reagowanie na powiadomienia trasowania

## Podsumowanie

W tym rozdziale pokazałem, jak dynamicznie wybierać komponenty za pomocą trasowania URL. Omówiłem różne sposoby definiowania tras. Pokazałem, jak włączyć API historii HTML5, aby obsłużyć trasowanie bez fragmentów URL. Przedstawiłem metodę tworzenia trasy typu `catchAll`, a także omówiłem, jak nazywać trasy i tworzyć dla nich aliasy. Zademonstrowałem sposób użycia segmentów dynamicznych w celu pobrania danych z wybranego adresu URL i omówiłem metodę otrzymywania powiadomień w momencie zmiany trasy, która to metoda wyświetla ten sam komponent. W kolejnym rozdziale opiszę szczegółowo elementy HTML używane do zarządzania trasowaniem.

## ROZDZIAŁ 23.

# Elementy związane z trasowaniem URL

W tym rozdziale opiszę niektóre możliwości elementów `router-link` i `router-view`. Pokażę, jak zmienić sposób przetwarzania elementów `router-link`, jak skorzystać z różnych zdarzeń do nawigacji, a także jak reagować na aktywację tras przez zmianę stylów stosowanych wobec elementów nawigacji. Zademonstruję także to, jak skorzystać z więcej niż jednego elementu `router-view` w aplikacji i jak zarządzać elementami `router-view`, gdy są wyświetlane w tym samym szablonie. Tabela 23.1 umiejscawia ten rozdział w szerszym kontekście.

**Tabela 23.1.** Umiejscowienie elementów związanych z trasowaniem URL w szerszym kontekście

Pytanie	Odpowiedź
Czym są elementy związane z trasowaniem URL?	Element <code>router-link</code> udostępnia funkcje, które kontrolują generowany kod HTML, zdarzenia odpowiedzialne za wyzwalanie nawigacji, a także klasy, do których element zostanie dodany w celu wskazania aktywnej trasy.
Dlaczego są użyteczne?	Mechanizmy elementu <code>router-link</code> są użyteczne, gdy musisz przedstawić mechanizmy nawigacji w formie niestandardowych elementów lub gdy chcesz dostarczyć informację zwrotną z pomocą stylów pochodzących z framework'a CSS. Mechanizmy elementu <code>router-view</code> są użyteczne w złożonych aplikacjach, ponieważ pozwalają na tworzenie złożonej treści.
Jak się z nich korzysta?	Omawiane mechanizmy są wdrażane za pomocą atrybutów w elementach <code>router-link</code> i <code>router-view</code> ze wsparciem na poziomie tras aplikacji.
Czy są jakieś pułapki lub ograniczenia?	W czasie stosowania rozwiązań omawianych w tym rozdziale należy upewnić się, że istnieją właściwe trasy typu <code>catchall</code> i przekierowania, które pozwolą uniknąć wyświetlenia pustych okien lub niewłaściwej informacji zwrotnej za pomocą elementów nawigacji.
Czy są jakieś rozwiązania alternatywne?	Omawiane mechanizmy są opcjonalne — nie musisz z nich korzystać.

Tabela 23.2 podsumowuje rozdział.

**Tabela 23.2.** Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Skonfiguruj elementy używane do nawigacji.	Skorzystaj z atrybutów dostarczonych przez element router-link.	23.4 – 23.7
Ostyluj element, gdy ten pasuje do aktywnego adresu URL.	Zdefiniuj style, które wybiorą klasy router-link-active i router-link-exact-active.	23.8
Upewnij się, że element jest ostylowany tylko, gdy pasuje do aktualnej trasy.	Zastosuj atrybut exact.	23.9
Zmień klasy używane do oznaczenia aktywnej trasy.	Skorzystaj z atrybutów exact-active-class i active-class.	23.10
Utwórz trasy zagnieźdzone.	Zastosuj wiele elementów router-view i powiąż je z trasami zdefiniowanymi za pomocą właściwości children.	23.11 – 23.17

## Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę z projektem *productapp* z rozdziału 22. Aby przygotować się do tego rozdziału, oznaczam jako komentarz metodę `create` w komponentie `ProductDisplay` (listing 23.1). Instrukcje w tej metodzie zostały użyte w celu pokazania możliwości magazynu danych z rozdziału 20. i nie są konieczne w tym rozdziale, ponieważ spowodują przywrócenie wartości magazynu danych do stanu oryginalnego za każdym razem, gdy zostanie utworzona instancja komponentu `ProductDisplay`.

**Listing 23.1.** Wyłączanie metody w pliku `src/components/ProductDisplay.vue`

```
...
<script>
import {
 mapState,
 mapMutations,
 mapActions,
 mapGetters
} from "vuex";
export default {
 computed: {
 ...mapState(["products"]),
 ...mapState({
 useStripedTable: state => state.prefs.stripedTable
 }),
 ...mapGetters({
 tableClass: "prefs/tableClass",
 editClass: "prefs/editClass",
 deleteClass: "prefs/deleteClass"
 })
 },
 methods: {
 editProduct(product) {
 this.selectProduct(product);
 this.$router.push("/edit");
 },
 createNew() {
 this.selectProduct();
 this.$router.push("/edit");
 },
 }
},
```

```

 ...mapMutations({
 selectProduct: "selectProduct",
 setEditButtonColor: "prefs/setEditButtonColor",
 setDeleteButtonColor: "prefs/setDeleteButtonColor"
 }),
 ...mapActions({
 deleteProduct: "deleteProductAction"
 })
 },
 // created() {
 // this.setEditableColor(false);
 // this.setDeleteColor(false);
 // }
}
</script>
...

```

Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj polecenie z listingu 23.2 w katalogu *productapp*.

#### ***Listing 23.2. Uruchamianie usługi sieciowej***

---

```
npm run json
```

---

Otwórz drugie okno wiersza poleceń, przejdź do katalogu *productapp* i wykonaj polecenie z listingu 23.3, aby uruchomić narzędzia deweloperskie Vue.js.

#### ***Listing 23.3. Uruchamianie narzędzi deweloperskich***

---

```
npm run serve
```

---

Po zakończeniu inicjalizacji otwórz okno przeglądarki i przejdź pod adres *http://localhost:8080*, aby zobaczyć przykładową aplikację (rysunek 23.1).

---

■ **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/vue2wp.zip*.

---

## Obsługa elementów router-link

Element *router-link* jest bardziej elastyczny, niż na pierwszy rzut oka wygląda, bowiem obsługuje różne przydatne opcje pozwalające na dostosowanie elementu HTML i dostarczenie użytecznych informacji zwrotnych użytkownikowi. Aby przygotować się do kolejnych podrozdziałów, do szablonu komponentu App dodaję elementy *router-link* (listing 23.4).

#### ***Listing 23.4. Dodawanie elementów nawigacji w pliku src/App.vue***

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <router-link to="/list" class="m-1">Lista</router-link>
 <router-link to="/create" class="m-1">Utwórz</router-link>
 </div>
 </div>
 </div>

```

ID	Nazwa	Kategoria	Cena		
1	Kajak	Sporty wodne	275	<button>Edytuj</button>	<button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button>	<button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button>	<button>Usuń</button>
4	Chorągiewki narożne	Piłka nożna	34.95	<button>Edytuj</button>	<button>Usuń</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button>	<button>Usuń</button>
6	Mysiąca czapeczka	Szachy	16	<button>Edytuj</button>	<button>Usuń</button>
7	Chwiejne krzesło	Szachy	29.95	<button>Edytuj</button>	<button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button>	<button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button>	<button>Usuń</button>

[Utwórz nowy](#)

**Rysunek 23.1.** Przykładowa aplikacja po uruchomieniu

```

</div>
<div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
</div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>

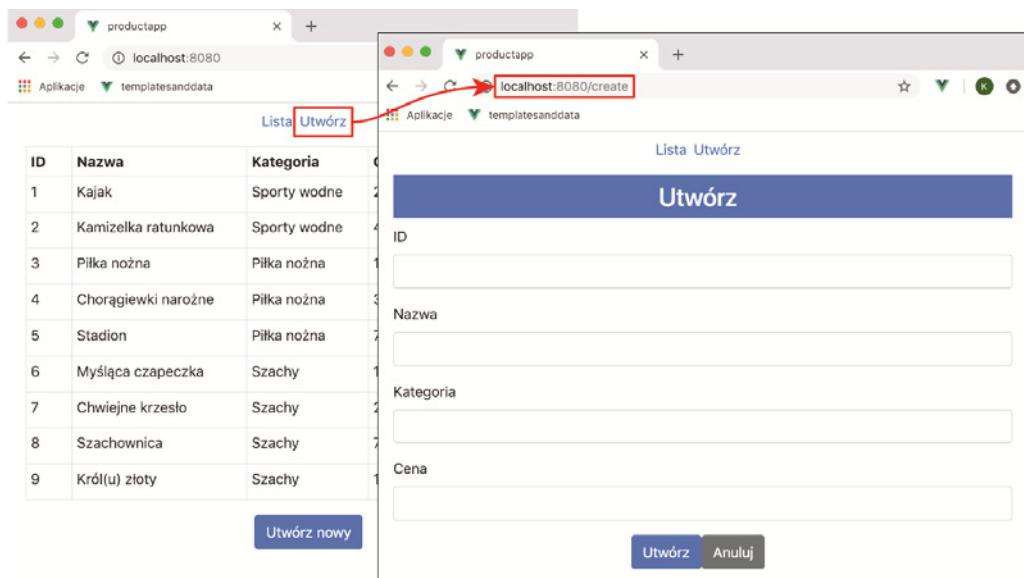
```

Przedstawione elementy router-link są przetwarzane jako część szablonu komponentu i zostały przekształcone w elementy kotwic (rysunek 23.2).

Element router-link jest konfigurowany przy użyciu atrybutów opisanych w tabeli 23.3, z których najbardziej przydatne omawiam w kolejnych punktach.

## Wybór rodzaju elementu

Domyślnie elementy router-link są przekształcane w kotwice (ang. *anchors*), czyli znaczniki a. Jeśli skorzystasz z narzędzia F12 do analizy obiektowego modelu dokumentu (DOM), zobaczyś efekt dodania elementów router-link jak w listingu 23.4.



Rysunek 23.2. Nawigacja za pomocą elementów router-link

Tabela 23.3. Atrybuty elementu router-link

Nazwa	Opis
tag	Ten atrybut określa rodzaj znacznika elementu HTML, który zostanie wygenerowany w momencie transformacji elementu router-link, co opisuję w punkcie „Wybór rodzaju elementu”.
event	Ten atrybut określa zdarzenie, które wyzwoli mechanizm nawigacji (por. punkt „Wybór zdarzenia nawigacji”).
exact	Ten atrybut określa, czy zostanie zastosowane częściowe dopasowanie adresów URL w operacji identyfikowania elementu, który jest związany z aktywną trasą (por. punkt „Stylowanie elementów łącza routera”).
active-class	Ten atrybut określa klasę, która zostanie przypisana do elementu w sytuacji, gdy aktywny adres URL rozpoczyna się od celu nawigacji elementu (por. punkt „Stylowanie elementów łącza routera”).
exact-active-class	Ten atrybut określa klasę, która zostanie przypisana do elementu w sytuacji, gdy aktywny adres URL będzie identyczny z celem nawigacji elementu (por. punkt „Stylowanie elementów łącza routera”).
to	Ten atrybut określa adres nawigacji i dodaje wpis do historii przeglądarki; jest równoważny z metodą nawigacji push, opisaną w rozdziale 22.
replace	Ten atrybut określa adres nawigacji, ale nie dodaje wpisu do historii przeglądarki; jest równoważny z metodą nawigacji replace, opisaną w rozdziale 22.
append	Ten atrybut określa adres względny URL, co jest użyteczne w przypadku, gdy nawigacja odbywa się na podstawie danych podanych przez użytkownika. W takiej sytuacji chcesz mieć pewność, że proces nawigacji będzie ograniczony do wybranej części aplikacji.

```
...
<div class="col text-center m-2">
 Lista
 Utwórz
</div>
...
```

- 
- **Wskazówka** Jeśli klikniesz jeden z elementów a, przekonasz się, że zostaną dodane klasy router-link-active i router-link-exact-active. Szczegóły objaśniam w punkcie „Stylowanie elementów łącza routera”.
- 

Atrybut tag służy do wyboru rodzaju elementu, który zostanie użyty do przekształcenia elementów router-link zamiast kotwic. Jest to użyteczne, gdy chcesz przedstawić elementy nawigacyjne w sposób, na który znaczniki a nie pozwolą, lub gdy chcesz zastosować style CSS, w przypadku których selektory nie dopasują elementów kotwic. W listingu 23.5 skorzystałem z atrybutu tag, aby wypełnić listę elementami nawigacyjnymi.

*Listing 23.5. Określanie rodzaju znacznika w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">

 <routerview tag="li" to="/list">Lista</routerview>
 <routerview tag="li" to="/create">Utwórz</routerview>

 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <routerview></routerview>
 </div>
 </div>
 </div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>
```

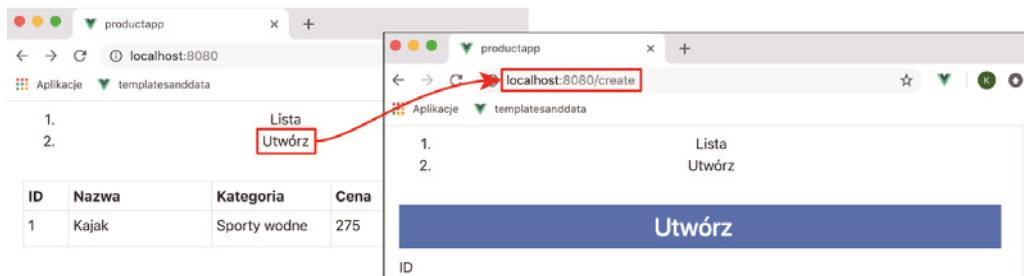
Atrybut tag określa, że elementy router-link powinny zostać zastąpione znacznikami li w momencie przetwarzania szablonu komponentu. Jeśli zapiszesz zmiany i skorzystasz z narzędzi F12, aby obejrzeć obiektowy model dokumentu, zobaczysz taki efekt:

```
...

 <li class="">Lista
 <li class="">Utwórz

...
```

Nawigacja nastąpi po kliknięciu jednego z elementów li (rysunek 23.3).



Rysunek 23.3. Zmiana rodzaju elementu nawigacji

## Wybór zdarzenia nawigacji

Domyślnym zdarzeniem nawigacji jest `click`, co oznacza, że nawigacja zostanie wykonana po kliknięciu elementu utworzonego przez element `router-link`. Atrybut `event` pozwala na określenie innego zdarzenia, dzięki czemu nawigacja może być realizowana na inne sposoby. W listingu 23.6 korzystam z atrybutu `event`, dzięki czemu nawigacja zostanie wykonana w momencie przesunięcia kurSORA myszy nad elementem nawigacji.

*Listing 23.6. Określanie zdarzenia nawigacji w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">

 <router-link tag="li" event="mouseenter" to="/list">
 Lista
 </router-link>
 <router-link tag="li" event="mouseenter" to="/create">
 Utwórz
 </router-link>

 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
 </div>
 </div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>
```

- 
- **Ostrzeżenie** Użytkownicy są przyzwyczajeni do klikania elementów nawigacyjnych, ponieważ tak działa zdecydowana większość aplikacji. Korzystaj z atrybutu `event` rozważnie, ponieważ łatwo zmylić swoich użytkowników i doprowadzić do nieoczekiwanych rezultatów.
-

Zdarzenie mouseenter jest wyzwalane, gdy kursor myszy wkroczy w obszar przeglądarki zajmowany przez element HTML. Nawigacja odbędzie się bez kliknięcia przycisku myszy przez użytkownika.

## Stylowanie elementów łącza routera

Stosując style do elementów router-link, musisz pamiętać, że stylujesz element, w który router-link zostanie przekształcony — nie zas sam element router-link. W listingu 23.7 do komponentu App dodaję atrybut style, używany do definiowania stylów dla elementów nawigacji.

*Listing 23.7. Stylowanie elementów nawigacji w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">

 <router-link tag="li" event="mouseenter" to="/list">
 Lista
 </router-link>
 <router-link tag="li" event="mouseenter" to="/create">
 Utwórz
 </router-link>

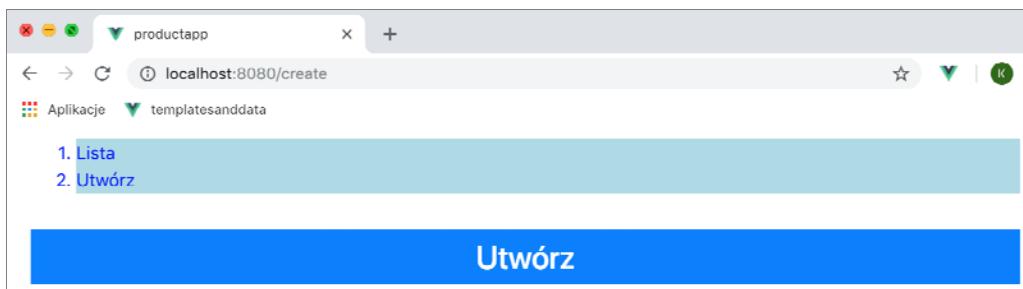
 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
 </div>
 </div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>
<style scoped>
 router-link { text-align: right; color: yellow; background-color: red; }
 li { text-align: left; color:blue; background-color: lightblue; }
</style>
```

Należy pamiętać, że to przeglądarka ewaluje selektory stylów. Operacja ta jest wykonywana po przekształceniu elementów router-link w element określony atrybutem tag. W związku z tym pierwszy styl zdefiniowany w listingu 23.7 nie dopasuje żadnych elementów, ponieważ po przetworzeniu szablonu komponentu nie będzie żadnych elementów router-link. Jest to powodem częstych pomyłek, zwłaszcza w przypadku użycia narzędzi, które zarządzają stylami CSS automatycznie. Selektor drugiego stylu dopasuje elementy po przekształceniu (rysunek 23.4).

---

■ **Wskazówka** Aby zobaczyć efekty zmian z rysunku 23.4, może być konieczne odświeżenie przeglądarki.

---



Rysunek 23.4. Stylowanie elementów nawigacji

## Reagowanie na aktywną trasę

Gdy aktywny adres URL pasuje do celu elementu nawigacji, pakiet Vue Router przypisuje do elementu klasy `router-link-active` i `router-link-exact-active`. W ten sposób możliwe jest zastosowanie stylów, które poinformują użytkownika o aktywnych elementach. W listingu 23.8 definiuję style, które korzystają z tych klas w swoich selektorach. Usunąłem atrybuty zdarzenia, dzięki czemu nawigacja nastąpi po kliknięciu. Dodałem elementy `router-link`, które prowadzą do adresów `/edit` i `/edit/1`.

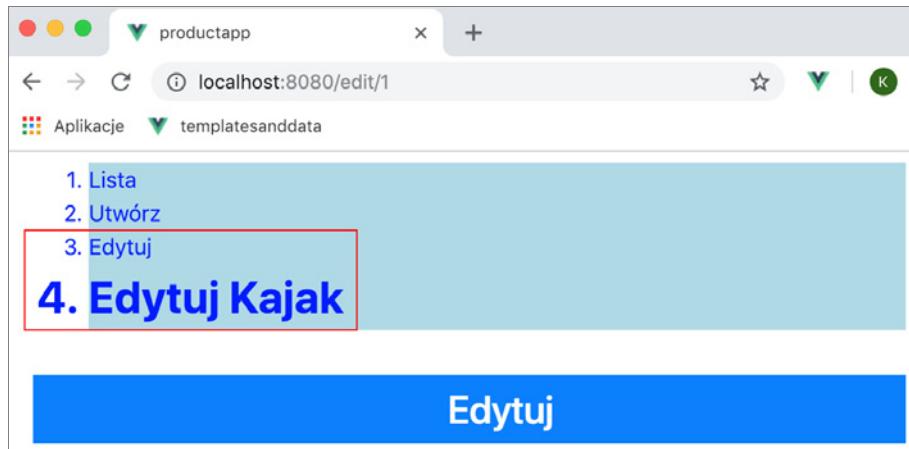
*Listing 23.8. Stylowanie aktywnego elementu nawigacji w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">

 <router-link tag="li" to="/list">Lista</router-link>
 <router-link tag="li" to="/create">Utwórz</router-link>
 <router-link tag="li" to="/edit">Edytuj</router-link>
 <router-link tag="li" to="/edit/1">Edytuj Kajak</router-link>

 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
 </div>
 </div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>
<style scoped>
li { text-align: left; color:blue; background-color: lightblue; }
.router-link-active { font-size: xx-large; }
.router-link-exact-active { font-weight: bolder; }
</style>
```

Klasy są dodawane automatyczne do elementów nawigacji i usuwane z nich w momencie zmiany trasy. Jeśli cel określony w atrybutie to pasuje dokładnie do bieżącego adresu URL, element otrzyma klasę router-link-exact-active, co w tym przypadku doprowadzi do pogrubienia. Element otrzyma klasę router-link-active, jeśli bieżący adres URL rozpoczyna się od celu określonego w atrybutie to. Różnicę widać po kliknięciu łącza *Edytuj Kajak*, które prowadzi pod adres /edit/1. Element *Edytuj*, którego celem jest /edit, jest związany z klasą router-link-active, ponieważ bieżący adres URL zaczyna się od celu: /edit/1 zaczyna się od ciągu /edit. Łącze *Edytuj Kajak* jest dodawane do obu klas, ponieważ bieżący adres URL zaczyna się od celu i pasuje do niego dokładnie. W związku z tym łącze *Edytuj* jest wyświetlane większą czcionką, a łącze *Edytuj Kajak* — również z wykorzystaniem pogrubienia (rysunek 23.5).



Rysunek 23.5. Reagowanie na klasy aktywnej trasy

Zastosowanie klasy router-link-active dla częściowych dopasowań adresów URL nie zawsze jest pomocne. Mechanizm ten można wyłączyć, dodając atrybut exact do elementu router-link (listing 23.9).

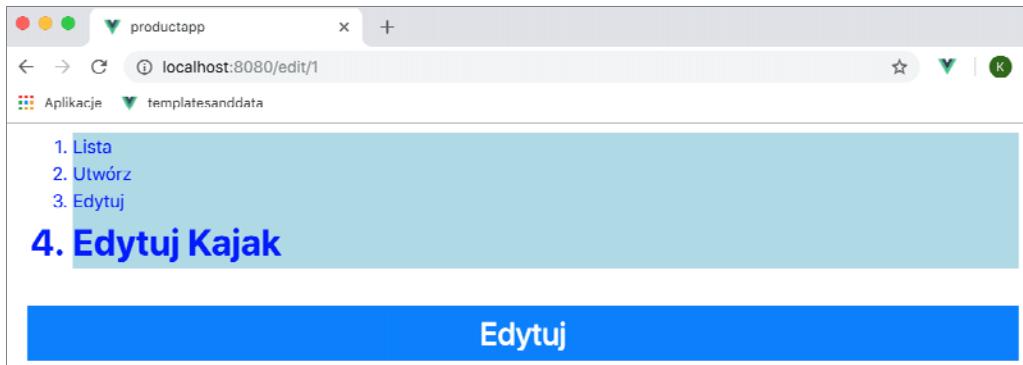
*Listing 23.9. Wyłączanie częściowego dopasowania adresów URL w pliku src/App.vue*

```
...
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">

 <routelink tag="li" to="/list">Lista</routelink>
 <routelink tag="li" to="/create">Utwórz</routelink>
 <routelink tag="li" to="/edit" exact>Edytuj</routelink>
 <routelink tag="li" to="/edit/1">Edytuj Kajak</routelink>

 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <routerview></routerview>
 </div>
 </div>
 </div>
</template>
...
```

Atrybut exact zadziała niezależnie od wartości do niego przypisanej. To właśnie obecność tego atrybutu (a nie jego wartość) spowoduje wyłączenie mechanizmu częściowego dopasowania. Atrybut zastosowany w przedstawionym listingu uniemożliwi dodanie klasy router-link-active do elementu *Edytuj* w momencie przejścia pod adres /edit/1 (rysunek 23.6).



Rysunek 23.6. Wyłączenie częściowego powiązania adresów URL

## Zmiana klas aktywnej trasy

Jeżeli korzystasz z frameworka CSS, takiego jak używany przez mnie Bootstrap, z pewnością zauważysz, że istnieją klasy przeznaczone do oznaczania elementów aktywnych. Klasy te różnią się od nazw używanych przez Vue Router. Za pomocą atrybutów active-class i exact-active-class można określić nazwy klas przypisanych do elementów, jeżeli ich cel odpowiada bieżącemu adresowi URL. W listingu 23.10 zamieniam listę elementów nawigacji na nieco bardziej tradycyjne przyciski i korzystam z atrybutu active-class-exact, aby określić nazwę klasy używanej do oznaczenia aktywnego przycisku. Usunąłem także element style, ponieważ nie potrzebuję już własnych stylów CSS.

*Listing 23.10. Określanie aktywnych nazw klas w pliku src/App.vue*

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/list"
 exact-active-class="btn-info"
 class="btn btn-primary">
 Lista
 </router-link>
 <router-link tag="button" to="/create"
 exact-active-class="btn-info"
 class="btn btn-primary">
 Utwórz
 </router-link>
 <router-link tag="button" to="/edit"
 exact-active-class="btn-info"
 class="btn btn-primary">
 Edytuj
 </router-link>
 <router-link tag="button" to="/edit/1"
 exact-active-class="btn-info"
 class="btn btn-primary">
```

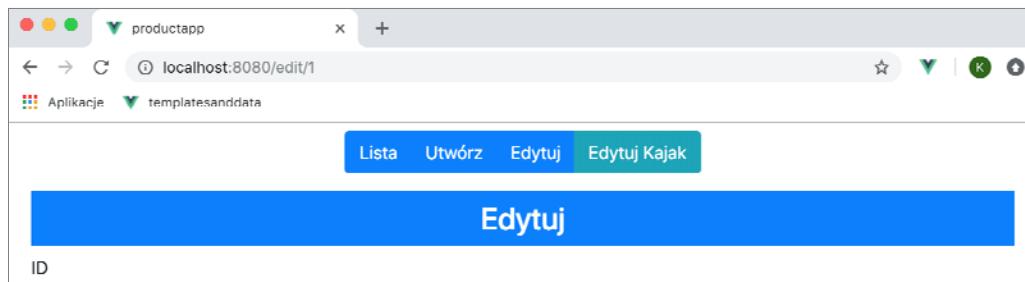
```

 Edytuj Kajak
 </router-link>
 </div>
 </div>
</div>
<div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
</div>
</div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>

```

- **Wskazówka** Możesz zmienić klasy używane do oznaczenia tras globalnie, korzystając z właściwości linkActiveClass i linkExactActiveClass w obiekcie konfiguracji trasowania, dzięki czemu zniknie konieczność określania tych klas w każdym elemencie.

Nowe elementy router-link otrzymują klasy btn i btn-primary, używane przez Bootstrapa do stylowania przycisków. Każdy z nich otrzyma także klasę btn-info, jeśli w danej chwili będzie reprezentował aktywną trasę (rysunek 23.7).



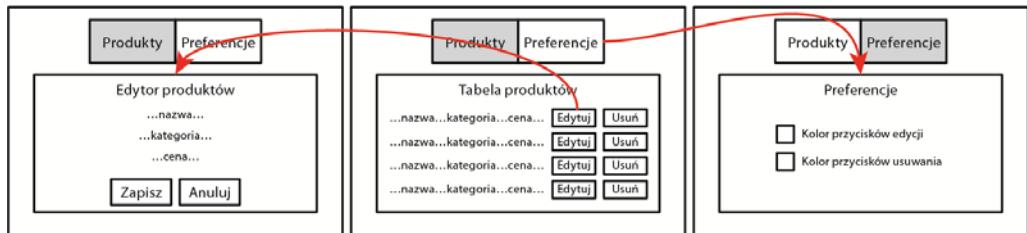
Rysunek 23.7. Zmiana klas aktywnej trasy

## Tworzenie tras zagnieżdzonych

Do tej pory w przykładach trasowania zakładałem, że aplikacja zawiera tylko jeden zbiór komponentów do wyświetlania i w momencie wyboru któregokolwiek z nich, zostanie wyświetlona ta sama treść. W złożonych aplikacjach jeden nadrzędny komponent może wyświetlać różne komponenty-dzieci. Aby obsłużyć to wymaganie, pakiet Vue Router wspiera trasy zagnieżdzione (ang. *nested routes*), nazywany też trasami-dziećmi (ang. *child routes*).

## Planowanie układu aplikacji

Korzystając z tras zagnieźdzonych, należy pamiętać o celu, który chcemy osiągnąć. W przykładowej aplikacji umieszczę nadrzędne elementy nawigacji, za pomocą których użytkownik dokona wyboru pomiędzy funkcjami związanymi z produktami a komponentem obsługi preferencji (ustawień) aplikacji. Rysunek 23.8 przedstawia strukturę aplikacji, którą chcę utworzyć.



Rysunek 23.8. Struktura aplikacji

Podjęcie decyzji w sprawie adresów URL, które obsłuży aplikacja, ułatwi utworzenie tras. Adresy URL, z których skorzystam w przykładowej aplikacji, są opisane w tabeli 23.4 i realizują ten sam schemat, który zastosowałem we wcześniejszych przykładach.

Tabela 23.4. Adresy URL w przykładowej aplikacji

Adres URL	Opis
/products/table	Ten adres URL jest powiązany z tabelą produktów.
/products/create	Ten adres URL jest powiązany z edytorem do tworzenia nowego produktu.
/products/edit/10	Ten adres URL jest powiązany z edytorem do modyfikacji wybranego produktu.
/preferences	Ten adres URL jest powiązany z widokiem ustawień aplikacji.

## Dodawanie komponentów do projektu

Aby utworzyć wymaganą strukturę, muszę zacząć od dodania komponentu. Najpierw dodaję do katalogu `src/components` plik `Preferences.vue` o treści z listingu 23.11.

Listing 23.11. Zawartość pliku `src/components/Preferences.vue`

```
<template>
 <div>
 <h4 class="bg-info text-white text-center p-2">Preferencje</h4>
 <div class="form-check">
 <input class="form-check-input" type="checkbox"
 v-bind:checked="primaryEdit" v-on:input="setPrimaryEdit">
 <label class="form-check-label">Główny kolor dla przycisków edycji</label>
 </div>
 <div class="form-check">
 <input class="form-check-input" type="checkbox"
 v-bind:checked="dangerDelete" v-on:input="setDangerDelete">
 <label class="form-check-label">Kolor ostrzeżenia dla przycisków usuwania</label>
 </div>
 </div>
</template>
```

```
<script>
import {
 mapState
} from "vuex";
export default {
 computed: {
 ...mapState({
 primaryEdit: state => state.prefs.primaryEditButton,
 dangerDelete: state => state.prefs.dangerDeleteButton
 })
 },
 methods: {
 setPrimaryEdit() {
 this.$store.commit("prefs/setEditButtonColor", !this.primaryEdit);
 },
 setDangerDelete() {
 this.$store.commit("prefs/setDeleteButtonColor", !this.dangerDelete);
 }
 }
}
</script>
```

Ten komponent wyświetla preferencje użytkownika. Prezentuje dwie właściwości stanu z magazynu danych za pomocą przycisków wyboru i aktualizuje ich wartości w przypadku zmiany stanu kontrolek przez użytkownika. Są to te same właściwości stanu z magazynu danych, które są używane do określenia koloru przycisków *Edytuj* i *Usuń* z komponentu *ProductDisplay*.

Następnie do katalogu *src/components* dodaję plik *Products.vue* o treści z listingu 23.12.

**Listing 23.12.** Zawartość pliku *src/components/Products.vue*

```
<template>
 <router-view></router-view>
</template>
```

Komponent zawiera jedynie element *template*, a z kolei ten składa się tylko z elementu *router-view*. Ten komponent wyświetli listę produktów lub edytor.

## Definiowanie tras

Dysponując niezbędnymi komponentami, mogę zdefiniować konfigurację trasowania aplikacji, aby zaimplementować adresy zdefiniowane w tabeli 23.4 (listing 23.13).

**Listing 23.13.** Definiowanie tras w pliku *src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: ProductDisplay
 },
 {
 path: "/product/:id/edit",
 component: ProductEditor
 }
]
})
```

```

 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
 },
 {
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
 },
 {
 path: "*",
 redirect: "/products/list"
 }
]
})
)

```

W powyższych trasach zawartych jest sporo informacji, dlatego omówię każdą z nich, objaśniając, w jaki sposób każda trasa wiąże się z omówioną przed chwilą strukturą aplikacji. Pierwsza trasa podąża schematem znany z poprzednich przykładów:

```

...
{ path: "/preferences", component: Preferences},
...

```

Ta trasa pozwoli wyświetlić komponent Preferences po przejściu na adres */preferences*. Nie zawiera ona żadnych segmentów dynamicznych, nazw, przekierowań lub innych specjalnych funkcji. Wybrany komponent zostanie wyświetlony w elemencie router-view, zdefiniowanym w szablonie komponentu App.

Następna trasa jest bardziej skomplikowana, więc podzielię ją na jeszcze mniejsze części. Pierwsza z nich jest całkiem prosta:

```

...
{ path: "/products", component: Products,
...

```

Właściwości path i component określają, że komponent Products zostanie wyświetlony, gdy adres URL przyjmie postać */products*. Podobnie jak w poprzedniej trasie, komponent Products zostanie wyświetlony w elemencie router-view w szablonie komponentu App. Szablon komponentu Products również zawiera element router-view, w ramach którego musimy wybrać komponent, więc właśnie dla tego wprowadzamy właściwość children:

```

...
{ path: "/products", component: Products,
 children: [
 { name: "table", path: "list", component: ProductDisplay},
 { name: "editor", path: ":op(create|edit)/:id(\d+)?",
 ...
 }
]
}

```

```

 component: ProductEditor},
 { path: "", redirect: "list" }
],
},
...

```

Właściwość `children` służy do zdefiniowania zbioru tras zastosowanych do elementu `router-view` w szablonie komponentu `Products`. Wartość właściwości `path` każdej z tras dzieci jest łączona z właściwością `path` rodzica, aby dopasować adres URL i wybrać komponent. W związku z tym komponent `ProductDisplay` jest wybierany dla adresu `/products/list`. Ścieżki dzieci mogą zawierać segmenty dynamiczne i wyrażenia regularne, co widać w przypadku komponentu `ProductEditor` i adresów `/create` oraz `/edit/:id`.

Ostatnią trasą w sekcji `children` jest trasa typu `catchall`, która przekieruje dowolny adres niedopasowany do wcześniejszych tras na adres `/products/list`. Zwrót uwagi, że ścieżką tej trasy jest pusty lańcuch (a nie gwiazdka), ponieważ chcę dopasować adresy URL, które nie mają wartości dla tego segmentu.

## Obsługa starych adresów URL

Gdy zestaw adresów URL w istniejącej aplikacji zmienia się, trzeba się upewnić, że adresy URL w komponentach również zostały zmodyfikowane lub zostały utworzone przekierowania lub aliasy pomiędzy starymi a nowymi adresami. Funkcja edycji (dostępna wcześniej za pomocą ścieżki `/edit/:id`) jest dostępna za pomocą ścieżki `/products/edit/:id`. Aby upewnić się, że stare adresy URL wciąż będą działać, w listingu 23.13 dodałem ścieżkę przekierowania.

```

...
{ path: "/edit/:id", redirect: to => `/products/edit/${to.params.id}` },
...

```

W rozdziale 22. utworzyłem przekierowanie za pomocą stałego, niezmiennego adresu URL. W tej sytuacji muszę zmienić podejście, ponieważ do nowej trasy muszę także przekazać wartość segmentu dynamicznego `id`. Jak widać w listingu, przekierowania mogą być wyrażane w formie funkcji, która otrzymuje dopasowaną trasę, a zwraca — adres URL przekierowania. W tym przykładzie funkcja przekierowania otrzymuje trasę i tworzy adres URL przekierowania zawierający wartość `id`.

## Tworzenie elementów nawigacji

W listingu 23.14 zamieniam przyciski nawigacji w szablonie komponentu `App` na te, które uwzględniają adresy URL opisane w tabeli 23.4.

*Listing 23.14. Nawigacja do nowych adresów URL w pliku src/App.vue*

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/products" active-class="btn-info"
 class="btn btn-primary">
 Produkty
 </router-link>
 <router-link tag="button" to="/preferences"
 active-class="btn-info" class="btn btn-primary">
 Preferencje
 </router-link>
 </div>
 </div>
 </div>
 </div>

```

```

<div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
</div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>

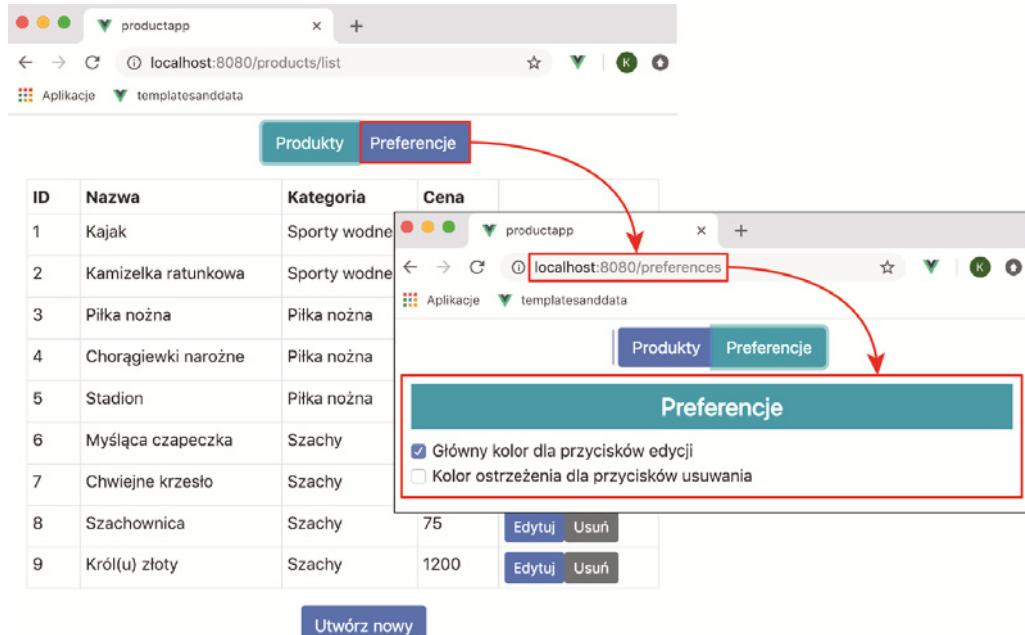
```

Przedstawione elementy router-link zostaną przekształcone w elementy button, które prowadzą do adresów URL /products i /preferences. Są one ostylowane za pomocą klas btn i btn-primary, czyli podstawowych klas przycisków z framework'a Bootstrap.

Aby oznaczyć przycisk reprezentujący aktywną trasę, korzystam z atrybutu active-class i dodaję do przycisku klasę btn-info, która stosuje inny kolor do przycisku Bootstrapa.

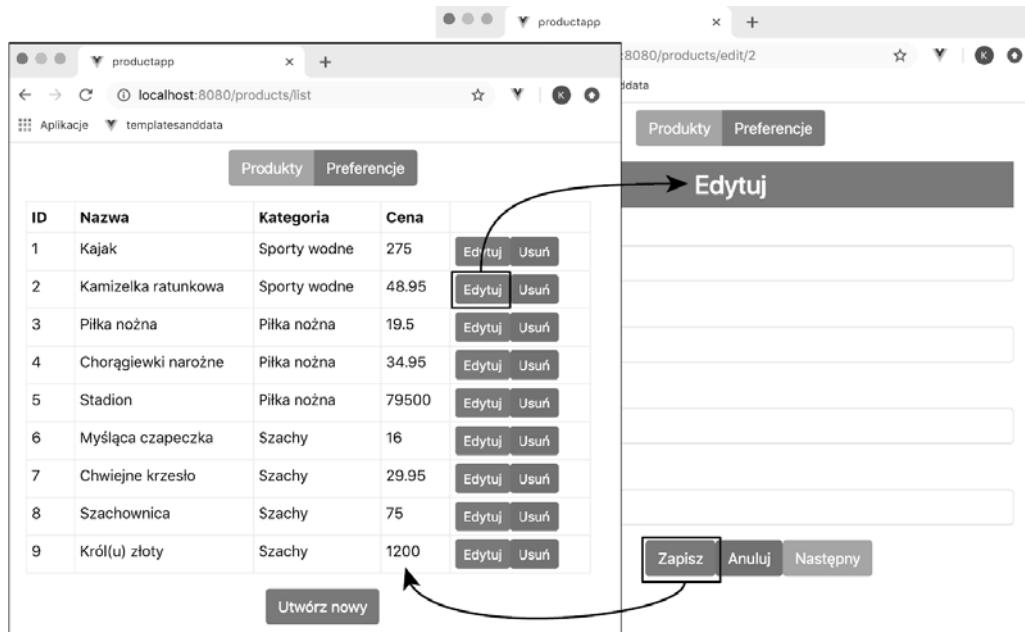
## Testowanie klas zagnieżdżonych

Wszystkie zmiany niezbędne do obsługi zagnieżdżonych elementów router-view zostały wprowadzone. Aby zobaczyć te zmiany, przejdź na stronę <http://localhost:8080>, po czym skorzystaj z przycisków *Produkty* i *Preferencje*, aby zmienić zawartość elementu router-view w komponencie szablonu App (rysunek 23.9).



Rysunek 23.9. Wybór komponentów dla nadzawanego elementu router-view

Aby obejrzeć zagnieźdzone elementy router-view w działaniu, kliknij przycisk *Produkty*, a następnie jeden z przycisków *Edytuj* z tabeli. Komponent wyświetlony przez element router-view w komponencie Products zostanie zamieniony na edytor, a do tabeli powrócisz, klikając przycisk *Zapisz* lub *Anuluj* (rysunek 23.10).



Rysunek 23.10. Wybór komponentów w zagnieźdzonym elemencie router-view

Zwróć uwagę, że elementy router-link i kod, który używa tras nazwanych, automatycznie skorzystają z adresów, które będą działać z nowymi trasami. Gdy definiowałem trasy w listingu 23.13, zastosowałem nazwy zdefiniowane w rozdziale 22. do odpowiednich tras w nowej konfiguracji.

```
...
{
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
},
...
```

Jeśli przetwarzaniu podlegają elementy `router-link` korzystające z tras nazwanych, wynikiem jest adres URL, który wskazuje określona trasę — oznacza to, że nazwy będą działać nawet, jeśli same trasy się zmienią. Widać to na przykład w przypadku przycisków `Edituj` z widoku tabeli, tworzonych z użyciem elementu `router-link`:

```
...
<router-link v-bind:to="{name: 'editor', params: { op: 'edit', id: p.id}}"
 v-bind:class="editClass" class="btn btn-sm">
 Edytuj
</router-link>
...
```

Atrybut `to` zawiera nazwę, która wskazuje na trasę edytora. Po przetworzeniu elementu otrzymujemy element kotwicy, którego cel jest związyany z trasą nazwaną:

```
...

 Edytuj

...
```

Obsługa tras nazwanych wymaga zastosowania dziwnej mieszanki kodu JavaScript i HTML, ale w rezultacie otrzymujemy znacznie bardziej elastyczną aplikację, która dostosowuje się do zmian wprowadzonych w konfiguracji trasowania bez potrzeby wprowadzania zmian w komponentach.

## Obsługa nazwanych elementów `router-view`

Niektóre komponenty muszą korzystać z wielu elementów `router-view` w tym samym szablonie, dzięki czemu możliwy jest dynamiczny wybór dwóch lub większej liczby komponentów. Atrybut `name` pozwala w takiej sytuacji rozróżnić elementy `router-view` za pomocą dodatkowej nazwy. Nazwy te są używane w trasach, aby wybrać komponenty do wyświetlenia.

Aby pokazać zasadę działania tego mechanizmu, tworzę w pliku `SideBySide.vue` w katalogu `src/components` nowy komponent o treści z listingu 23.15.

**Listing 23.15.** Zawartość pliku `src/components/SideBySide.vue`

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <h3 class="bg-secondary text-white text-center p-2">Widok lewy</h3>
 <router-view name="left" class="border border-secondary p-2" />
 </div>
 <div class="col text-center m-2">
 <h3 class="bg-secondary text-white text-center p-2">Widok prawy</h3>
 <router-view name="right" class="border border-secondary p-2" />
 </div>
 </div>
 </div>
</template>
```

Ten komponent zawiera element szablonu, w którym umieszczone są dwa elementy `router-view`. Rozróżniam je za pomocą nazwy — jeden nosi nazwę `left`, a drugi `right`. Klasy Bootstrapa w połączeniu z elementami struktury HTML pozwolą na wyświetlenie zawartości elementów `router-view` obok siebie. Aby obsługiwać nowe elementy `router-view`, dodam obsługę nowych adresów URL, które opisuję w tabeli 23.5.

**Tabela 23.5.** Adresy URL zastosowane do obsługi elementów nazwanych widoków routera

Adres URL	Opis
/named/tableleft	Ten adres URL pozwoli wyświetlić tabelę produktów w elemencie left, a edytora — w elemencie right.
/named/tableright	Ten adres URL pozwoli wyświetlić tabelę produktów w elemencie right, a edytora — w elemencie left.

Aby obsłużyć te adresy URL, dodaję elementy nawigacji pokazane w listingu 23.16 do szablonu komponentu App.

**Listing 23.16.** Dodawanie elementów nawigacji w pliku src/App.vue

```
<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group">
 <routerview tag="button" to="/products"
 active-class="btn-info" class="btn btn-primary">
 Produkty
 </routerview>
 <routerview tag="button" to="/preferences"
 active-class="btn-info" class="btn btn-primary">
 Preferencje
 </routerview>
 <routerview to="/named/tableleft" class="btn btn-primary"
 active-class="btn-info">
 Tabela po lewej
 </routerview>
 <routerview to="/named/tableright" class="btn btn-primary"
 active-class="btn-info">
 Tabela po prawej
 </routerview>
 </div>
 </div>
 </div>
 <div class="row">
 <div class="col m-2">
 <routerview></routerview>
 </div>
 </div>
 </div>
</template>
<script>
 export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>
```

Aby dokończyć obsługę nowych adresów URL i wskazać nazwane elementy router-view, dodaję trasy pokazane w listingu 23.17.

**Listing 23.17.** Dodawanie tras w pliku *src/router/index.js*

```

import Vue from "vue";
import VueRouter from "vue-router";
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
import SideBySide from "../components/SideBySide";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
 },
 {
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
 },
 {
 path: "/named",
 component: SideBySide,
 children: [
 {
 path: "tableleft",
 components: {
 left: ProductDisplay,
 right: ProductEditor
 }
 },
 {
 path: "tableright",
 components: {
 left: ProductEditor,
 right: ProductDisplay
 }
 }
]
 },
 {
 path: "*",
 }
]
});

```

```

 redirect: "/products"
 }
]
})
}

```

Właściwość `components` służy do definiowania tras, które wskazują na nazwane elementy `router-view`. Ta właściwość otrzymuje obiekt, którego właściwościami są nazwy elementów `router-view`, a wartościami są komponenty, które powinny być wyświetlane:

```

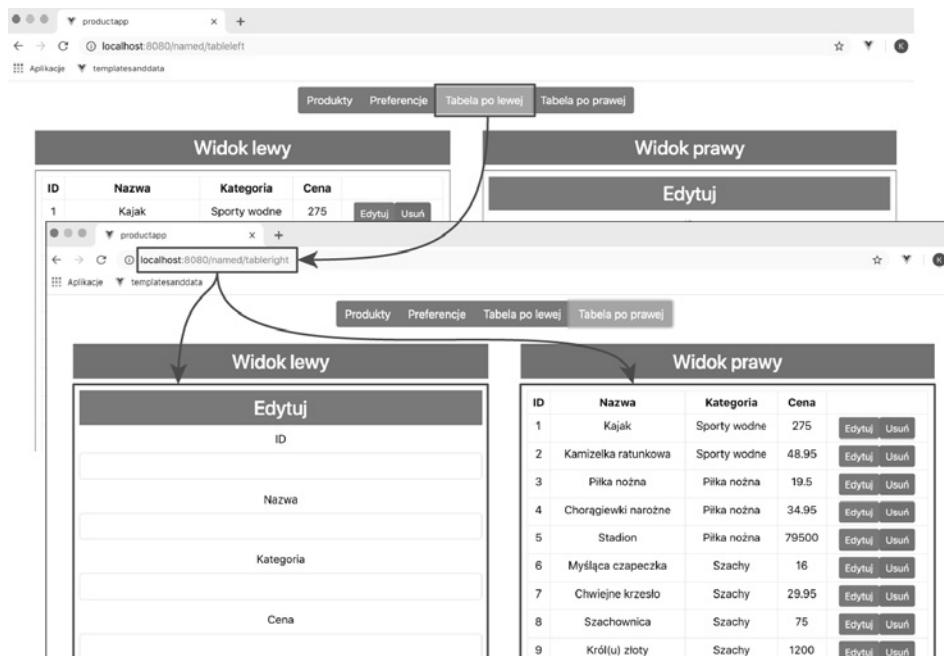
...
{
 path: "tableleft",
 components: {
 left: ProductDisplay,
 right: ProductEditor
 }
},
...

```

Właściwość `components` powoduje wyświetlenie komponentu `ProductDisplay` w elemencie `router-view` o nazwie `left`, a komponentu `ProductEditor` — w elemencie `router-view` o nazwie `right`.

- **Uwaga** Właściwość, która określa elementy nazwane, to `components` (liczba mnoga), a nie `component` (liczba pojedyncza), którą stosowaliśmy w innych trasach z listingu 23.17.

Aby przetestować nowy mechanizm, przejdź pod adres `http://localhost:8080` i kliknij przyciski *Tabela po lewej* i *Tabela po prawej*, które wskazują na trasy zdefiniowane w listingu 23.17. Efekt będzie taki jak na rysunku 23.11.



Rysunek 23.11. Przykład użycia nazwanych elementów `router-view`

## Podsumowanie

W tym rozdziale opisałem niektóre zaawansowane funkcje dostępne dzięki zastosowaniu trasowania URL w aplikacjach Vue.js. Omówiłem metody konfiguracji elementów router-link, pozwalające na generowanie różnych elementów HTML. Pokazałem, jak obsługiwać różne zdarzenia, a także jak stylować elementy nawigacji, aby wyświetlić informację zwrotną dla użytkownika. Omówiłem element router-view, pokazując, jak korzystać z tras zagnieżdżanych, dzięki czemu aplikacja może zawierać więcej niż jeden element, a także pokazałem, jak nazywać te elementy, gdy są zdefiniowane w jednym szablonie. W kolejnym rozdziale opisuję zaawansowane funkcje trasowania URL.



# ROZDZIAŁ 24.



## Zaawansowane trasowanie URL

W tym rozdziale opisuję bardziej zaawansowane funkcje trasowania. Rozpocznę od pokazania, jak utworzyć skomplikowaną strukturę konfiguracji trasowania, podzieloną na wiele plików, jak korzystać ze strażników trasowania w celu kontroli procesu nawigacji, a także jak leniwie wczytywać komponenty wybrane przez trasę. Na zakończenie rozdziału pokażę, jak współpracować z komponentami, które nie zostały napisane z myślą o współpracy z pakietem Vue Router. Tabela 24.1 podsumowuje rozdział.

**Tabela 24.1.** Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Zgrupuj powiązane trasy.	Zdefiniuj odrębne moduły dla każdej grupy tras.	24.3 – 24.5
Obserwuj i przechwytyj proces nawigacji.	Skorzystaj ze strażnika trasы.	24.6, 24.9 – 24.11
Zmień cel nawigacji.	Zdefiniuj zastępczy adres URL w funkcji next strażnika trasы.	24.7 – 24.8
Uzyskaj dostęp do komponentu przed zakończeniem nawigacji.	Skorzystaj z funkcji wywołania zwrotnego w metodzie strażnika beforeRouteEnter.	24.12 – 24.15
Wczytuj leniwie komponenty.	Skorzystaj z dynamicznych zależności dla komponentów.	24.16 – 24.21
Skorzystaj z komponentów, które nie mają związku z systemem trasowania.	Skorzystaj z propów przy definicji tras.	24.22 – 24.23

## Przygotowania do tego rozdziału

W tym rozdziale kontynuuję pracę z projektem *productapp* z rozdziału 23. Nie są wymagane żadne zmiany. Aby uruchomić REST-ową usługę sieciową, otwórz okno wiersza poleceń i wykonaj polecenie z listingu 24.1 w katalogu *productapp*.

**Listing 24.1.** Uruchamianie usługi sieciowej

```
npm run json
```

Otwórz drugie okno wiersza poleceń, przejdź do katalogu *productapp* i wykonaj polecenie z listingu 24.2, aby uruchomić narzędzia deweloperskie Vue.js.

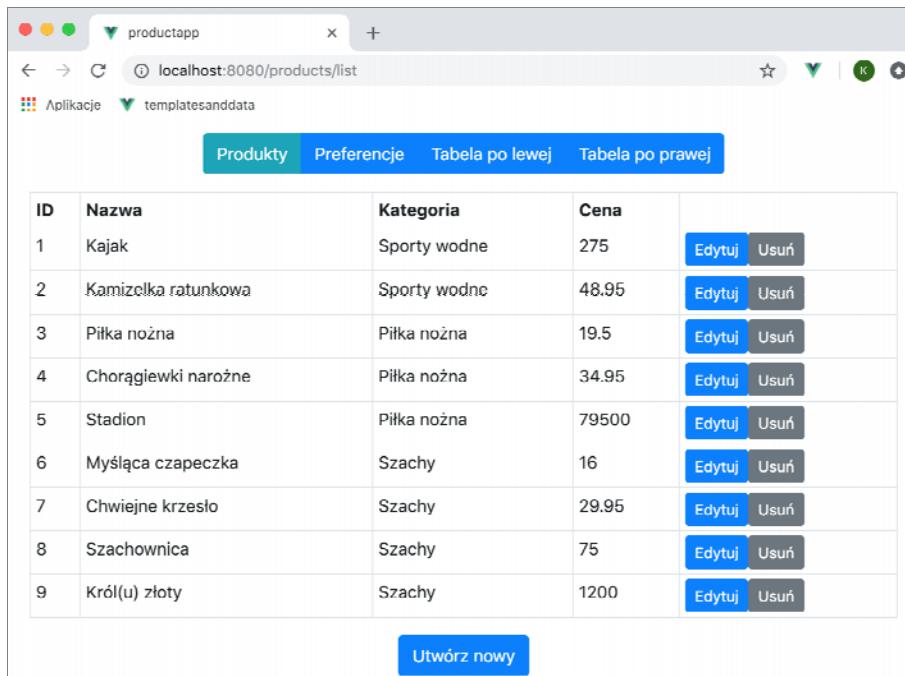
#### *Listing 24.2. Uruchamianie narzędzi deweloperskich*

---

```
npm run serve
```

---

Po zakończeniu inicjalizacji otwórz okno przeglądarki i przejdź pod adres <http://localhost:8080>, aby zobaczyć przykładową aplikację (rysunek 24.1).



ID	Nazwa	Kategoria	Cena	Edytuj	Usuń
1	Kajak	Sporty wodne	275	Edytuj	Usuń
2	Kamizelka ratunkowa	Sporty wodne	48.95	Edytuj	Usuń
3	Piłka nożna	Piłka nożna	19.5	Edytuj	Usuń
4	Chorągiewki narożne	Piłka nożna	34.95	Edytuj	Usuń
5	Stadion	Piłka nożna	79500	Edytuj	Usuń
6	Myśląca czapczka	Szachy	16	Edytuj	Usuń
7	Chwiejne krzesło	Szachy	29.95	Edytuj	Usuń
8	Szachownica	Szachy	75	Edytuj	Usuń
9	Król(u) złoty	Szachy	1200	Edytuj	Usuń

Rysunek 24.1. Przykładowa aplikacja po uruchomieniu

- **Wskazówka** Przykładowy projekt do tego rozdziału — podobnie jak do wszystkich innych — można pobrać z serwera FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przykłady/vue2wp.zip>.

## Stosowanie odrębnych plików dla powiązanych tras

W miarę wzrostu liczby adresów URL obsługiwanych przez aplikację śledzenie tras i wspieranych przez nie elementów w aplikacji staje się coraz bardziej skomplikowane. Dodatkowe pliki JavaScript mogą posłużyć do zgrupowania powiązanych tras, które wówczas można zimportować do głównej konfiguracji trasowania.

Nie ma szybkiej metody grupowania tras — podejście, które przedstawiam w tym rozdziale, polega na oddzieleniu tras, które obsługują prezentację funkcji obok siebie za pomocą elementów typu *router-view*, od tras, które obsługują podstawowe mechanizmy przedstawiane w pozostałej części aplikacji. Do katalogu *src/router* dodaję plik *basicRoutes.js* o zawartości z listingu 24.3.

**Listing 24.3.** Zawartość pliku *src/router/basicRoutes.js*

```
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
export default [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
 },
 {
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
 },
]
```

Ten plik eksportuje tablicę tras, które obsługują podstawowe adresy URL wspierane przez aplikację. Następnie dodaję w katalogu *src/router* plik *sideBySideRoutes.js* o treści z listingu 24.4.

**Listing 24.4.** Zawartość pliku *src/router/sideBySideRoutes.js*

```
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import SideBySide from "../components/SideBySide";
export default {
 path: "/named",
 component: SideBySide,
 children: [
 {
 path: "tableleft",
 components: {
 left: ProductDisplay,
 right: ProductEditor
 }
 },
 {
 path: "tableright",
 components: {
 left: ProductEditor,
 right: ProductDisplay
 }
 }
]
};
```

```

 }
]
}

```

Umieszczanie grup tras w odrębnych plikach pozwala mi uprościć plik *index.js* (listing 24.5), który przedstawia sposób importowania tras w plikach *basicRoutes.js* i *sideBySideRoutes.js*.

**Listing 24.5. Importowanie tras w pliku src/router/index.js**

```

import Vue from "vue";
import VueRouter from "vue-router";
// import ProductDisplay from "../components/ProductDisplay";

// import ProductEditor from "../components/ProductEditor";
// import Preferences from "../components/Preferences";
// import Products from "../components/Products";
// import SideBySide from "../components/SideBySide";
import BasicRoutes from "./basicRoutes";
import SideBySideRoutes from "./sideBySideRoutes";
Vue.use(VueRouter);
export default new VueRouter({
 mode: "history",
 routes: [
 ...BasicRoutes,
 SideBySideRoutes,
 {
 path: "*",
 redirect: "/products"
 }
]
})

```

Zawartość każdego z plików importuję za pomocą instrukcji `import`, nadając treści nazwę, do której odwołuję się we właściwości `routes`. Plik *basicRoutes.js* eksportuje tablicę, która musi być odpakowana za pomocą operatora rozwińcia, opisanego w rozdziale 4. Plik *sideBySideRoutes.js* eksportuje pojedynczy obiekt i może być używany bez operatora rozwińcia. Aplikacja nie zmienia się pod kątem funkcjonalnym, ale konfiguracja trasowania zostaje podzielona, co ułatwia zarządzanie powiązanymi trasami.

## Ochrona tras

Jednym z problemów związanych z umożliwieniem przemieszczania się użytkownika po aplikacji za pomocą adresów URL jest ryzyko, że użytkownik zawędruje tam, gdzie nie powinien — np. do panelu administratora bez uprzedniego zalogowania się lub do formularza edycji przed pobraniem danych z serwera. Strażnicy nawigacji chronią dostęp do tras i mogą reagować na próby niewłaściwej nawigacji przez przekierowanie na inny adres URL lub całkowite anulowanie procesu nawigacji. Strażnicy nawigacji mogą być stosowani na różne sposoby, opisane w kolejnych punktach.

## Definiowanie globalnych strażników nawigacji

Globalni strażnicy nawigacji to metody zdefiniowane jako część konfiguracji trasowania. Są one używane do kontroli dostępu wszystkich tras w aplikacji. Istnieją trzy globalne metody strażników nawigacji, które są używane do rejestrowania funkcji związanych z kontrolą nawigacji (tabela 24.2).

**Tabela 24.2.** Globalne metody strażników nawigacji

Nazwa	Opis
beforeEach	Ta metoda jest wywoływana przed zmianą aktywnej trasy.
afterEach	Ta metoda jest wywoywana po zmianie aktywnej trasy.
beforeResolve	Ta metoda jest podobna do metody beforeEach, ale jest wywoywana po sprawdzeniu wszystkich strażników tras i komponentów. Więcej na ten temat znajdziesz w punkcie „Omówienie kolejności strażników”.

Każda z tych metod jest wywoywana z pomocą obiektu `VueRouter` i przyjmuje funkcję, która zostanie wykonana w czasie nawigacji. Najłatwiej wyjaśnić to na przykładzie. W listingu 24.6 korzystam z metody `beforeEach`, aby zapobiec wykonaniu nawigacji z tras, których ścieżka zaczyna się od adresów `/named` i `/preferences`.

**Listing 24.6.** Ochrona trasy w pliku `src/router/index.js`

```
import Vue from "vue";
import VueRouter from "vue-router";
import BasicRoutes from "./basicRoutes";
import SideBySideRoutes from "./sideBySideRoutes";
Vue.use(VueRouter);
const router = new VueRouter({
 mode: "history",
 routes: [
 ...BasicRoutes,
 SideBySideRoutes,
 {
 path: "*",
 redirect: "/products"
 }
]
});
export default router;
router.beforeEach((to, from, next) => {
 if (to.path == "/preferences" && from.path.startsWith("/named")) {
 next(false);
 } else {
 next();
 }
});
```

Globalni strażnicy tras stanowią najbardziej elastyczny rodzaj strażników, ponieważ można z nich skorzystać do przechwycenia całej nawigacji. Konfiguracja może jednak wydać się nieco dziwna, co pokazuje listing 24.6. Obiekt `VueRouter` jest tworzony za pomocą słowa kluczowego `new`. Funkcja, którą chcesz wykonać, jest przekazywana do odpowiedniej metody z tabeli 24.2. W tym przykładzie przekazałem funkcję do metody `beforeEach`, co oznacza, że będzie ona wykonywana przed każdą operacją nawigacji.

Funkcja przyjmuje trzy argumenty. Pierwsze dwa stanowią obiekty reprezentujące trasę, do której przechodzimy, i trasę, którą opuszczamy. Obiekty zawierają właściwości opisane w rozdziale 22. Korzystam z ich właściwości `path`, aby sprawdzić, czy trasą źródłową jest trasa `/named`, a trasą docelową — `/preferences`.

Trzeci argument stanowi funkcja wykonywana w celu akceptowania, przekierowania lub anulowania nawigacji. Przekazuje ona żądania nawigacji do następnego strażnika trasy w celu dalszego przetwarzania. W związku z tym argument ten nosi na ogół nazwę `next` (następny). Efekt działania nawigacji zależy od tego, co przekażemy w formie argumentu do funkcji (tabela 24.3).

**Tabela 24.3.** Przykłady użycia funkcji next w strażnikach nawigacji

Przykład użycia	Opis
next()	Przy braku przekazanych argumentów do funkcji nawigacja będzie odbywać się dalej.
next(false)	Gdy funkcja otrzyma wartość false, nawigacja zostanie przerwana.
next(url)	Gdy funkcja otrzyma tekst, zostanie on zinterpretowany jako adres URL i stanie się nowym celem nawigacji.
next(object)	Gdy funkcja otrzyma obiekt, zostanie on zinterpretowany jako nowy cel nawigacji. Jest to przydatne w przypadku wyboru tras za pomocą nazwy (por. punkt „Przekierowanie do trasy nazwanej”).
next(callback)	To specjalna wersja funkcji next, z której można skorzystać tylko w jednej sytuacji, opisanej w punkcie „Dostęp do komponentu w metodzie beforeRouteEnter” — w innych przypadkach ten wariant nie jest obsługiwany.

W listingu przekazuję wartość false do funkcji next, jeżeli aktualny adres URL rozpoczyna się od ciągu /named, a adres docelowy to /preferences. W ten sposób anuluje żądanie nawigacji. W innym przypadku wywołuję funkcję next bez argumentu, co oznacza automatyczną akceptację dalszego procesu nawigacji.

- **Ostrzeżenie** Musisz pamiętać o wywołaniu funkcji next w swoich strażnikach. W aplikacji może występować wielu strażników i nie zostaną oni wywołani, jeśli zapomnisz o wywołaniu funkcji next — w ten sposób możesz doprowadzić do powstania nieoczekiwanych zachowań.

## Przekierowanie żądania nawigacji pod inny adres URL

Alternatywą dla przerwania procesu nawigacji jest przekierowanie użytkownika pod inny adres URL. W listingu 24.7 dodaję nową funkcję strażnika, która przechwytuje żądania kierowane pod adres URL /named/tableright i przekierowuje je na adres /products.

**Listing 24.7.** Tworzenie nowego strażnika w pliku src/router/index.js

```
import Vue from "vue";
import VueRouter from "vue-router";
import BasicRoutes from "./basicRoutes";
import SideBySideRoutes from "./sideBySideRoutes";
Vue.use(VueRouter);
const router = new VueRouter({
 mode: "history",
 routes: [
 ...BasicRoutes,
 SideBySideRoutes,
 {
 path: "*",
 redirect: "/products"
 }
]
});
export default router;
router.beforeEach((to, from, next) => {
 if (to.path == "/preferences" && from.path.startsWith("/named")) {
 next(false);
 } else {
```

```

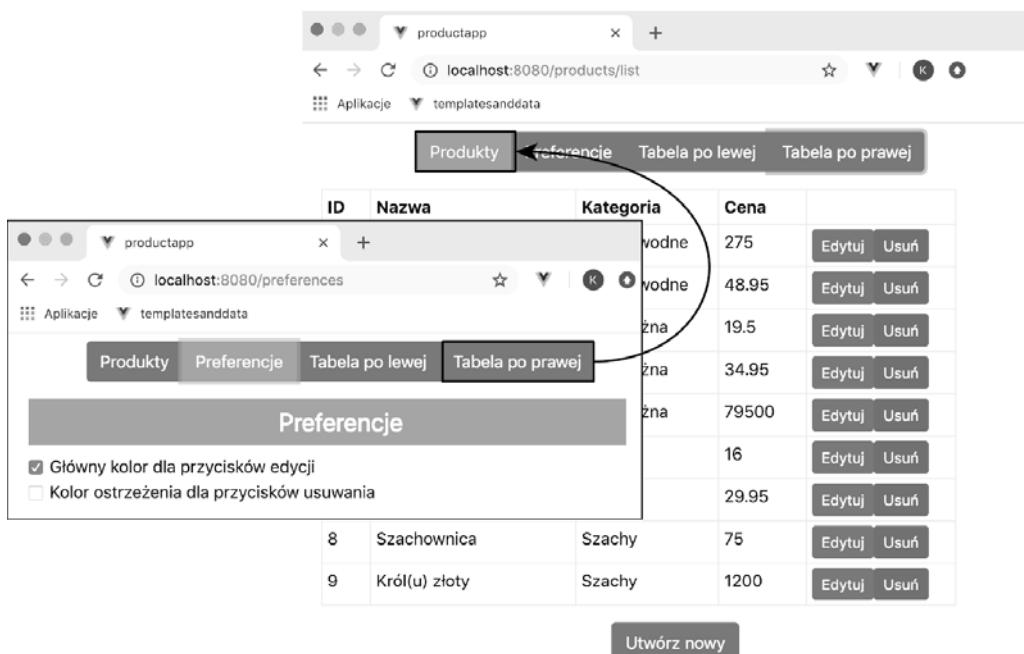
 next();
 }
});

router.forEach((to, from, next) => {
 if (to.path == "/named/tableright") {
 next("/products");
 } else {
 next();
 }
});

```

- **Wskazówka** Jeżeli istnieje wiele globalnych funkcji strażników, są one wykonywane w kolejności, w jakiej zostały przekazane do metody beforeEach lub beforeAfter.

Ten warunek mogłem zaimplementować w istniejącej funkcji strażnika, ale chciałem przy tym pokazać sposób obsługi wielu strażników, co jest użyteczne w grupowaniu powiązanych mechanizmów w złożonych aplikacjach. Aby zobaczyć efekt przekierowania, przejdź na stronę <http://localhost:8080/preferences> i kliknij przycisk *Tabela po prawej*. Zamiast pokazywać komponenty obok siebie, aplikacja przejdzie do adresu */products*, jak na rysunku 24.2.



Rysunek 24.2. Przekierowanie przy użyciu strażnika trasy

- **Wskazówka** Gdy wykonujesz przekierowanie w strażniku trasy, zostanie utworzone nowe żądanie nawigacji, co doprowadzi do ponownego wykonania funkcji strażników. Z tego względu trzeba uważać, aby nie utworzyć pętli przekierowań, w której dwie funkcje strażników będą nawzajem kierować aplikację pomiędzy dwoma adresami URL.

## Przekierowanie do trasy nazwanej

Jeśli chcesz przekierować żądanie nawigacji do trasy nazwanej, możesz przekazać obiekt o właściwości name do funkcji next (listing 24.8).

*Listing 24.8. Przekierowanie do trasy nazwanej w pliku src/router/index.js*

```
import Vue from "vue";
import VueRouter from "vue-router";
import BasicRoutes from "./basicRoutes";
import SideBySideRoutes from "./sideBySideRoutes";
Vue.use(VueRouter);
const router = new VueRouter({
 mode: "history",
 routes: [
 ...BasicRoutes,
 SideBySideRoutes,
 {
 path: "*",
 redirect: "/products"
 }
]
});
export default router;
router.beforeEach((to, from, next) => {
 if (to.path == "/preferences" && from.path.startsWith("/named")) {
 next(false);
 } else {
 next();
 }
});
router.beforeEach((to, from, next) => {
 if (to.path == "/named/tableright") {
 next({
 name: "editor",
 params: {
 op: "edit",
 id: 1
 }
 });
 } else {
 next();
 }
});
```

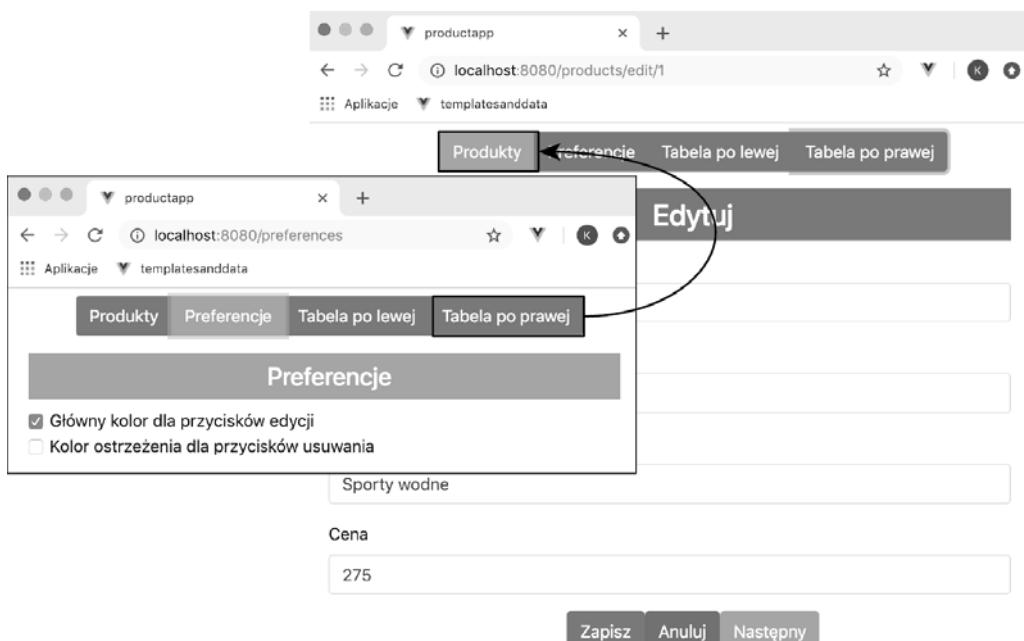
W tym listingu korzystam z funkcji next w celu przekierowania nawigacji do trasy editor (rysunek 24.3).

## Definiowanie strażników dla konkretnych tras

Pojedyncze trasy również mogą implementować strażników, co pozwala bardziej naturalnie zarządzać nawigacją. Jedyną metodą, z której można korzystać w strażnikach przeznaczonych dla pojedynczych tras, jest beforeEnter. Stosuję ją w listingu 24.9, aby ochronić dwie trasy.

*Listing 24.9. Ochrona pojedynczych tras w pliku src/router/sideBySideRoutes.js*

```
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import SideBySide from "../components/SideBySide";
```



Rysunek 24.3. Przekierowanie nawigacji do trasy nazwanej

```
export default {
 path: "/named",
 component: SideBySide,
 children: [
 {
 path: "tableleft",
 components: {
 left: ProductDisplay,
 right: ProductEditor
 }
 },
 {
 path: "tableright",
 components: {
 left: ProductEditor,
 right: ProductDisplay
 },
 beforeEnter: (to, from, next) => {
 next("/products/list");
 }
 },
 beforeEnter: (to, from, next) => {
 if (to.path == "/named/tableleft") {
 next("/preferences");
 } else {
 next();
 }
 }
 }
}
```

Korzystając z tras zagnieźdzonych, możesz chronić zarówno rodzica, jak i pojedyncze dzieci. W listingu dodaję do tras rodzica strażnika, który przekierowuje żądania z adresu `/named/tableleft` na `/preferences`. Aby sprawdzić ten mechanizm w praktyce, kliknij *Tabela po lewej* (rysunek 24.4).

ID	Nazwa	Kategoria	Cena	
1	Kajak	Sporty wodne	275	<button>Edytuj</button> <button>Usuń</button>
2	Kamizelka ratunkowa	Sporty wodne	48.95	<button>Edytuj</button> <button>Usuń</button>
3	Piłka nożna	Piłka nożna	19.5	<button>Edytuj</button> <button>Usuń</button>
4	Chorągiewki narożne	Piłka nożna	34.95	<button>Edytuj</button> <button>Usuń</button>
5	Stadion	Piłka nożna	79500	<button>Edytuj</button> <button>Usuń</button>
6	Myśląca czapczka	Szachy	16	<button>Edytuj</button> <button>Usuń</button>
7	Chwiejące krzesło	Szachy	29.95	<button>Edytuj</button> <button>Usuń</button>
8	Szachownica	Szachy	75	<button>Edytuj</button> <button>Usuń</button>
9	Król(u) złoty	Szachy	1200	<button>Edytuj</button> <button>Usuń</button>

[Utwórz nowy](#)

Rysunek 24.4. Ochrona pojedynczej trasy

## Omówienie kolejności strażników

W listingu 24.9 dodałem do jednej z tras-dzieci strażnika, który przekierowuje żądania z adresu `/named/tableright` na `/products/list`. Jeśli jednak klikniesz przycisk *Tabela po prawej*, przekonasz się, że strażnik nie działa zgodnie z wcześniejszymi założeniami.

Dzieje się tak, ponieważ globalni strażnicy tras są wykonywani przed indywidualnymi. Jeden ze strażników globalnych zajmuje się przetwarzaniem żądań kierowanych pod adres `/named/tableright`. Gdy strażnik wykonuje przekierowanie, przetwarzanie aktualnej trasy jest przerywane i zaczyna się nowy proces nawigacji. Oznacza to, że żaden ze strażników, który zostałby wykonany po tym, który wykonał przekierowanie, nie miałby szansy na analizę żądania.

Jak wspomniałem w tabeli 24.2, metoda `beforeResolve` jest wykonywana po przejściu przez wszystkie inne rodzaje strażników. Daje ona możliwość zdefiniowania globalnego strażnika, będącego ostatnim miejscem na wykonanie działania przed rozpoczęciem przetwarzania strażników indywidualnych tras. W listingu 24.10 skorzystałem z metody `beforeResolve`, aby zmienić funkcję strażnika, która blokowała strażnika tras zdefiniowanego w listingu 24.9.

### Listing 24.10. Zmiana strażnika w pliku `src/router/index.js`

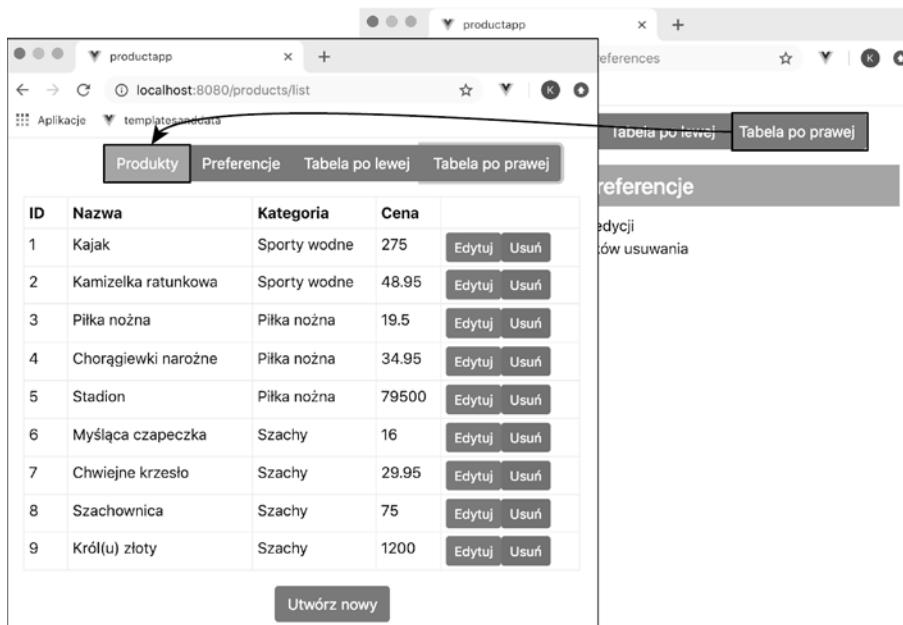
```
import Vue from "vue";
import VueRouter from "vue-router";
import BasicRoutes from "./basicRoutes";
import SideBySideRoutes from "./sideBySideRoutes";
Vue.use(VueRouter);
const router = new VueRouter({
 mode: "history",
 routes: [
 ...BasicRoutes,
 SideBySideRoutes,
 {
 path: "**",
 beforeResolve(to, from, next) {
 if (to.name === "table-right") {
 next({ name: "table-left" });
 } else {
 next();
 }
 }
 }
]
});
```

```

 redirect: "/products"
 }
]
});
export default router;
router.beforeEach((to, from, next) => {
 if (to.path == "/preferences" && from.path.startsWith("/named")) {
 next(false);
 } else {
 next();
 }
});
router.beforeResolve((to, from, next) => {
 if (to.path == "/named/tableright") {
 next({
 name: "editor",
 params: {
 op: "edit",
 id: 1
 }
 });
 } else {
 next();
 }
})
)

```

W rzeczywistych projektach taka zmiana byłaby użyteczna tylko, jeżeli istniałby pewien zbiór adresów URL, dla których strażnik indywidualny nie przerwałby procesu nawigacji. W tym przypadku widać efekt po kliknięciu przycisku *Tabela po prawej*. Aplikacja rozpoczęte nawigację pod adresem */named/tableright*, który zostanie przechwycony przez strażnika indywidualnej trasy. To właśnie on przekieruje aplikację na adres */products/list* (rysunek 24.5).



Rysunek 24.5. Efekt uporządkowania strażników tras

## Definiowanie strażników tras dla komponentów

W rozdziale 22. skorzystałem z metody `beforeRouteUpdate` w komponencie `ProductEditor`, aby zareagować na zmiany trasy:

```
...
beforeRouteUpdate(to, from, next) {
 this.selectProduct(to);
 next();
}
...
```

Jest to jedna z metod strażników tras, którą mogą implementować komponenty w celu ochrony tras odpowiedzialnych za wyświetlanie tych komponentów. Zastosowanie metody `beforeRouteUpdate` w celu otrzymania powiadomienia o zmianie to przydatna technika, jednak ta metoda to jedynie przykład metod ochronnych komponentów, opisanych w tabeli 24.4.

**Tabela 24.4. Metody ochronne komponentów**

Nazwa	Opis
<code>beforeRouteEnter</code>	Ta metoda jest wywoływana przed wyborem komponentu przez docelową trasę. Służy do kontroli dostępu do trasy, zanim komponent zostanie utworzony. Aby uzyskać dostęp do komponentu w tej metodzie, należy skorzystać ze specjalnego mechanizmu opisanego w punkcie „Dostęp do komponentu w metodzie <code>beforeRouteEnter</code> ”.
<code>beforeRouteUpdate</code>	Ta metoda jest wywoływana, gdy trasa, która wybrała aktywny komponent, ulega zmianie, a dany komponent został również wybrany przez nową trasę.
<code>beforeRouteLeave</code>	Ta metoda jest wywoywana, gdy aplikacja zamierza opuścić aktualną trasę, która wybrała aktywny komponent.

Te metody otrzymują te same trzy argumenty co pozostały strażnicy. Można je wykorzystać do akceptowania, przekierowania i anulowania nawigacji. W listingu 24.11 zaimplementowałem metodę `beforeRouteLeave`, aby poprosić użytkownika o zgodę na wykonanie procesu nawigacji w momencie zmiany trasy.

**Listing 24.11. Ochrona trasy w pliku src/components/Preferences.vue**

```
<template>
<div>
 <div v-if="displayWarning" class="text-center m-2">
 <h5 class="bg-danger text-white p-2">
 Czy masz pewność?
 </h5>
 <button class="btn btn-danger" v-on:click="doNavigation">
 Tak
 </button>
 <button class="btn btn-danger" v-on:click="cancelNavigation">
 Anuluj
 </button>
 </div>
 <h4 class="bg-info text-white text-center p-2">Preferencje</h4>
 <div class="form-check">
 <input class="form-check-input" type="checkbox"
 v-bind:checked="primaryEdit" v-on:input="setPrimaryEdit">
 <label class="form-check-label">Główny kolor dla przycisków edycji</label>
 </div>
</div>
```

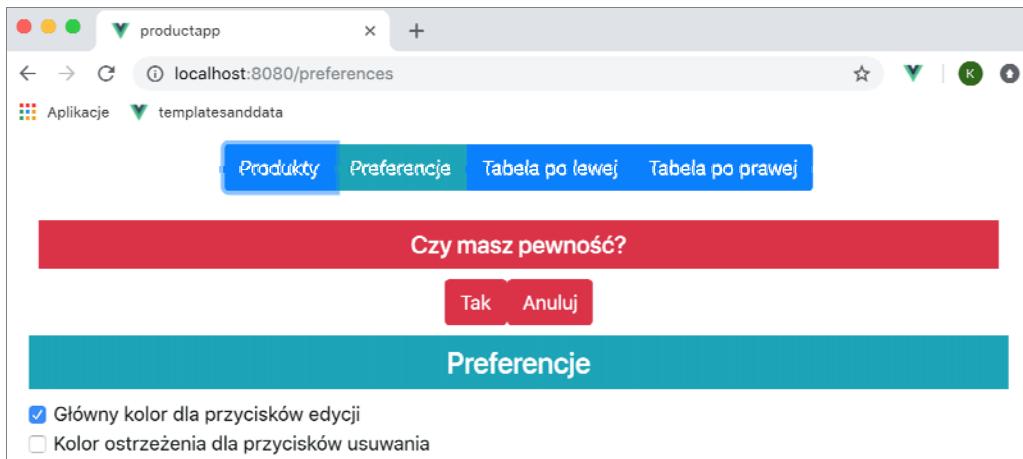
```

<div class="form-check">
 <input class="form-check-input" type="checkbox"
 v-bind:checked="dangerDelete" v-on:input="setDangerDelete">
 <label class="form-check-label">Kolor ostrzeżenia dla przycisków usuwania</label>
</div>
</div>
</template>
<script>

import {
 mapState
} from "vuex";
export default {
 data: function() {
 return {
 displayWarning: false,
 navigationApproved: false,
 targetRoute: null
 }
 },
 computed: {
 ...mapState({
 primaryEdit: state => state.prefs.primaryEditButton,
 dangerDelete: state => state.prefs.dangerDeleteButton
 })
 },
 methods: {
 setPrimaryEdit() {
 this.$store.commit("prefs/setEditButtonColor", !this.primaryEdit);
 },
 setDangerDelete() {
 this.$store.commit("prefs/setDeleteButtonColor", !this.dangerDelete);
 },
 doNavigation() {
 this.navigationApproved = true;
 this.$router.push(this.targetRoute.path);
 },
 cancelNavigation() {
 this.navigationApproved = false;
 this.displayWarning = false;
 }
 },
 beforeRouteLeave(to, from, next) {
 if (this.navigationApproved) {
 next();
 } else {
 this.targetRoute = to;
 this.displayWarning = true;
 next(false);
 }
 }
}
</script>

```

Metoda `beforeRouteLeave` jest wykonywana, gdy aplikacja zamierza przejść do trasy, która wyświetli inny komponent. W tym przykładzie proszę użytkownika o potwierdzenie i powstrzymuję proces nawigacji, dopóki go nie otrzymam. Aby obejrzeć efekt, kliknij przyciski *Preferencje* i *Produkty*. Strażnik trasy powstrzyma nawigację, dopóki nie klikniesz przycisku *Tak* (rysunek 24.6).



Rysunek 24.6. Zastosowanie strażnika trasy w komponencie

## Dostęp do komponentu w metodzie beforeRouteEnter

Metoda beforeRouteEnter jest wykonywana przed utworzeniem komponentu, co zapewnia, że proces nawigacji może być anulowany przed rozpoczęciem cyklu życia komponentu. Może to spowodować problem, jeśli chcesz wykorzystać tę metodę do wykonania zadania wymagającego dostępu do właściwości i metod komponentu (np. pobrania danych z usługi sieciowej). Aby rozwiązać ten problem, funkcja next przekazana do metody beforeRouteEnter przyjmuje funkcję wywołania zwrotnego, która jest wywoływana po utworzeniu komponentu. W ten sposób metoda beforeRouteEnter ma dostęp do komponentu, ale tylko, gdy możliwość anulowania lub przekierowania procesu nawigacji została wykluczona. W ramach przykładu utworzyłem komponent, który korzysta z metody beforeRouteEnter, aby uzyskać dostęp do metody zdefiniowanej przez komponent w pliku *src/component/FilteredData.vue* o treści z listingu 24.12.

*Listing 24.12. Zawartość pliku src/components/FilteredData.vue*

```
<template>
 <div>
 <h3 class="bg-primary text-center text-white p-2">
 Dane dla kategorii {{ category }}
 </h3>
 <div class="text-center m-2">
 <label>Kategoria:</label>
 <select v-model="category">
 <option>Wszystkie</option>
 <option>Sporty wodne</option>
 <option>Piłka nożna</option>
 <option>Szachy</option>
 </select>
 </div>
 <h3 v-if="loading" class="bg-info text-white text-center p-2">
 Wczytywanie danych...
 </h3>
 <table v-else class="table table-sm table-bordered">
 <tr>
 <th>ID</th>
 <th>Nazwa</th>
 <th>Kategoria</th>
 <th>Cena</th>
```

```

 </tr>
 <tbody>
 <tr v-for="p in data" v-bind:key="p.id">
 <td>{{ p.id }}</td>
 <td>{{ p.name }}</td>
 <td>{{ p.category }}</td>
 <td>{{ p.price }}</td>
 </tr>
 </tbody>
</table>
</div>
</template>
<script>
import Axios from "axios";
const baseUrl = "http://localhost:3500/products/";
export default {
 data: function() {
 return {
 loading: true,
 data: [],
 category: "Wszystkie"
 }
 },
 methods: {
 async getData(route) {
 if (route.params != null && route.params.category != null) {
 this.category = route.params.category;
 } else {
 this.category = "Wszystkie";
 }
 let url = baseUrl +
 (this.category == "Wszystkie" ? "" : `?category=${this.category}`);
 this.data.push(...(await Axios.get(url)).data);
 this.loading = false;
 }
 },
 watch: {
 category() {
 this.$router.push(`/filter/${this.category}`);
 }
 },
 async beforeRouteEnter(to, from, next) {
 next(async component => await component.getData(to));
 },
 async beforeRouteUpdate(to, from, next) {
 this.data.splice(0, this.data.length);
 await this.getData(to);
 next();
 }
}
</script>

```

Ten komponent pozwala na filtrowanie danych produktów według kategorii, pobierając najświeższe dane wprost z usługi sieciowej za pomocą biblioteki Axios opisanej w rozdziale 19. Dla uproszczenia skorzystałem z Axiosa bezpośrednio, przechowując dane w komponencie, zamiast modyfikować repozytorium i magazyn danych.

Komponent przedstawia element select, który służy do wyboru kategorii do filtrowania danych. W momencie użycia elementu select właściwość category jest modyfikowana, co powoduje rozpoczęcie

procesu nawigacji przez obserwatora. Celem staje się adres URL zawierający nazwę kategorii. Dzięki temu wybór kategorii *Piłka nożna* spowoduje przejście pod adres */filter/Piłka%20nożna*.

Komponent implementuje dwie metody strażników tras w komponentach. Obie metody są używane do wywołania asynchronicznej metody `getData` komponentu, która przyjmuje obiekt trasy i korzysta z niego w celu pobrania właściwych danych z usługi sieciowej. Pakiet `json-server`, dodany w rozdziale 19., obsługuje filtrowanie danych, dlatego żądanie o adresie `http://localhost:3500/products?category=Piłka%20nożna` spowoduje pobranie tylko tych obiektów, których kategoria to *Piłka nożna*. Jedna z metod strażników tras jest całkiem prosta:

```
...
async beforeRouteUpdate(to, from, next) {
 this.data.splice(0, this.data.length);
 await this.getData(to);
 next();
}
...
```

Ten sposób pracy z komponentami, z którego korzystam w całej książce, polega na użyciu słowa `this` w celu odwołania do obiektu komponentu. Wywołanie `this.getData` jest więc używane do wykonania metody `getData` zdefiniowanej w komponencie. Osiągnięcie tego samego efektu w przypadku metody `beforeRouteEnter` wymaga nieco innego podejścia.

```
...
async beforeRouteEnter(to, from, next) {
 next(component => component.getData(to));
},
...
```

Metoda `beforeRouteEnter` jest wykonywana przed utworzeniem komponentu (i przed początkiem jego zwykłego cyklu życia), co oznacza, że nie jesteśmy w stanie skorzystać ze słowa `this`, aby uzyskać dostęp do komponentu. Zamiast tego metoda `next` może otrzymać funkcję, która zostanie wykonana po utworzeniu komponentu i otrzyma go w formie argumentu. W listingu korzystam z funkcji `next`, aby wykonać metodę `getData` po utworzeniu komponentu.

To podejście do pobierania danych może wydawać się nazbyt złożone — wystarczyłoby skorzystać z metody `created` cyklu życia komponentu, opisanej w rozdziale 17. Metoda `beforeRouteEnter` jest jednak szczególnie użyteczna, ponieważ dzięki niej jesteśmy w stanie anulować proces nawigacji przed utworzeniem komponentu — coś, czego nie zrobimy w metodzie `created`, wywoływanej już po zakończeniu nawigacji i utworzeniu komponentu. W ramach przykładu do metody `beforeRouteEnter` dodaję nowy warunek, który spowoduje przekierowanie użytkownika, jeśli ten wybrał kategorię inną niż *Wszystkie* (listing 24.13).

**Listing 24.13.** Ochrona komponentu w pliku `src/components/FilteredData.vue`

```
...
async beforeRouteEnter(to, from, next) {
 if (to.params.category != "Wszystkie") {
 next("/filter/Wszystkie");
 } else {
 next(async component => await component.getData(to));
 }
},
...
```

Metoda przekierowuje wszelkie próby przejścia na adres URL, o ile kategoria nie jest równa wartości *Wszystkie*, odpowiedzialnej za pobranie wszystkich danych z serwera. Po wyświetleniu komponentu nawigacja do pozostałych komponentów jest możliwa, ponieważ zmiany te są chronione przez metodę `beforeRouteUpdate`.

Ten przykład łatwiej zrozumieć, gdy widać go w działaniu. Aby dodać obsługę nowego komponentu, dodaję instrukcję z listingu 24.14 do zbioru podstawowych tras.

**Listing 24.14.** Dodawanie trasy do pliku *src/router/basicRoutes.js*

```

import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
import FilteredData from "../components/FilteredData";
export default [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
 },
 {
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
 },
 {
 path: "/filter/:category",
 component: FilteredData
 }
]

```

Aby ułatwić nawigację do nowej trasy, dodaję element nawigacji do nadrzędnego szablonu komponentu App (listing 24.15).

**Listing 24.15.** Dodawanie elementu nawigacji w pliku *src/App.vue*

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/products"
 active-class="btn-info" class="btn btn-primary">
 Produkty
 </router-link>
 <router-link tag="button" to="/preferences"
 active-class="btn-info" class="btn btn-primary">
 Preferencje
 </router-link>
 <router-link to="/named/tableleft" class="btn btn-primary">

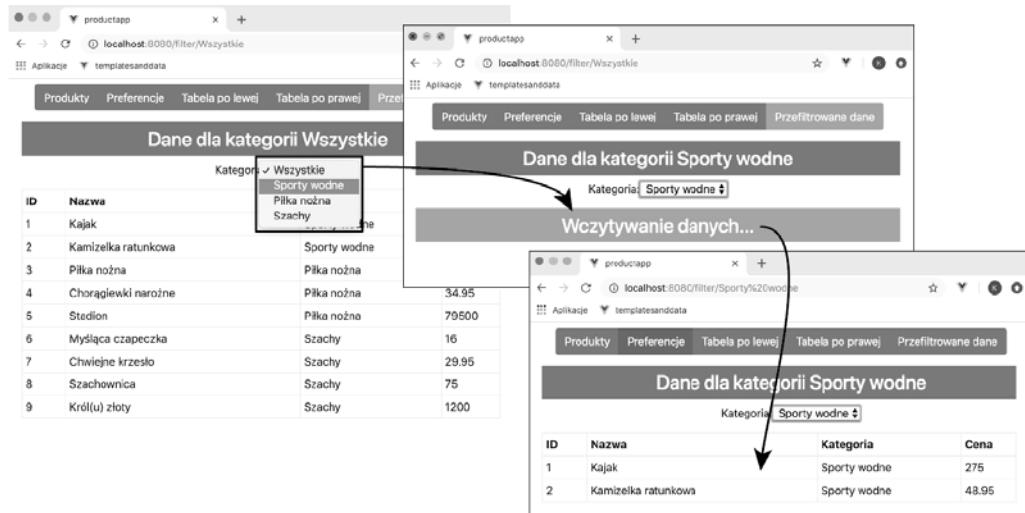
```

```

 active-class="btn-info">
 Tabela po lewej
 </router-link>
 <router-link to="/named/tableright" class="btn btn-primary"
 active-class="btn-info">
 Tabela po prawej
 </router-link>
 <router-link to="/filter/Wszystkie" class="btn btn-primary"
 active-class="btn-info">
 Przefiltrowane dane
 </router-link>
 </div>
</div>
<div class="row">
 <div class="col m-2">
 <router-view></router-view>
 </div>
</div>
</div>
</template>
<script>
export default {
 name: 'App',
 created() {
 this.$store.dispatch("getProductsAction");
 }
}
</script>

```

Wynikiem tej zmiany jest nowy przycisk, który przeniesie użytkownika do trasy związanej z wyborem komponentu `FilteredData`. Metoda strażnika trasy pozwoli na nawigację tylko, gdy wartość segmentu kategorii to `Wszystkie` — pozostałe żądania zostaną przekierowane. Po wyświetleniu komponentu wybór innego komponentu za pomocą elementu `select` spowoduje przejście do adresu URL chronionego przez metodę `beforeRouteUpdate`, która zareaguje na zmianę trasy pobraniem danych ze wskazanej kategorii (rysunek 24.7).



Rysunek 24.7. Dostęp do obiektu komponentu w strażniku trasy

## Praca ze strażnikami tras w komponentach

Praca ze strażnikami tras w komponentach może być trudna. Narzędzia deweloperskie Vue.js aktualizują aplikację na bieżąco, ale dzieje się to w sposób, który nie wyzwala prawidłowo metod strażników tras, przez co aby przekonać się o ich działaniu, konieczne staje się odświeżenie okna przeglądarki.

Na podobnej zasadzie, gdy przeglądarka i usługa sieciowa są włączone na tym samym komputerze, odpowiedź z usługi będzie niezwykle szybka, przez co trudno będzie zobaczyć komponent wyświetlany w trakcie pobierania danych. Jeśli chcesz spowolnić proces pobierania danych, aby przekonać się, jak ów komponent wygląda, dodaj następujące instrukcje przed wykonaniem żądania HTTP:

```
...
await new Promise(resolve => setTimeout(resolve, 3000));
...
```

W przypadku komponentu z listingu 24.15 instrukcja ta została wstawiona do metody `getData` tuż przed instrukcją, która wysyła żądanie HTTP za pomocą Axiosa. W ten sposób wprowadzona została trzysekundowa pauza przed wysłaniem żądania.

## Ładowanie komponentów na żądanie

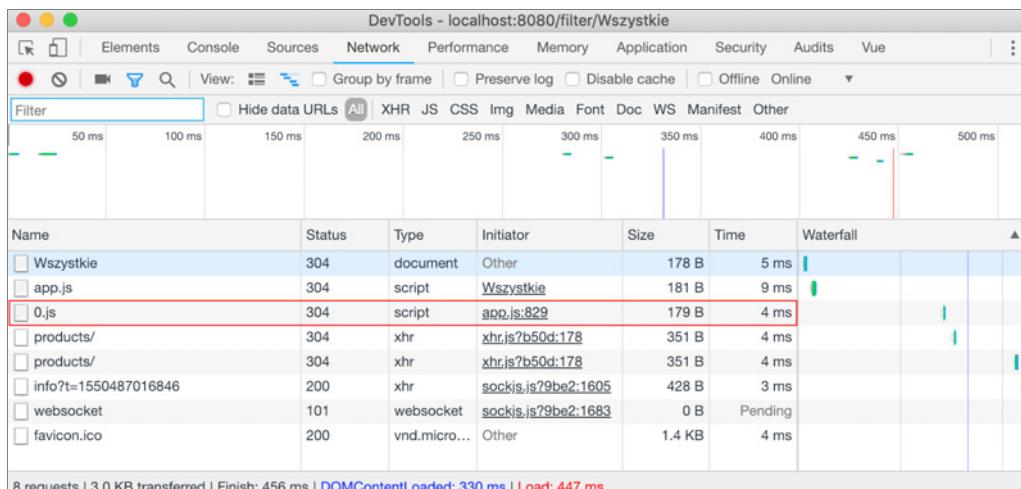
Komponenty wymagane przez trasę mogą być wykluczone z paczki JavaScript generowanej przez aplikację. Wczytywanie może następować tylko w razie potrzeby, z zastosowaniem tych samych funkcji, które opisałem w rozdziale 21. Przy korzystaniu z trasowania adresów URL dostępne są tylko niektóre funkcje leniwego ładowania. Opcje takie jak `loading` czy `delay` są ignorowane (w kolejnym punkcie opisuję za to metodę wyświetlania wiadomości ładowania za pomocą strażników tras). W listingu 24.16 zmieniam instrukcję, która importuje komponent `FilteredData`, dzięki czemu jest ona ładowana leniwie.

*Listing 24.16. Leniwe ładowanie komponentu w pliku src/router/basicRoutes.js*

```
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
const FilteredData = () => import("../components/FilteredData");
export default [
 { path: "/preferences", component: Preferences },
 {
 path: "/products", component: Products,
 children: [{ name: "table", path: "list", component: ProductDisplay },
 {
 name: "editor", path: ":op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 { path: "", redirect: "list" }]
 },
 { path: "/edit/:id", redirect: to => `/products/edit/${to.params.id}` },
 { path: "/filter/:category", component: FilteredData }
]
```

Stosuję tutaj funkcję `import`, z której skorzystałem w rozdziale 21., z tym samym efektem. Komponent `FilteredData` jest wykluczany z głównej paczki JavaScript i umieszczany w swojej własnej paczce, która zostanie wczytana przy pierwszym użyciu.

Aby upewnić się, że komponent `FilteredData` jest wczytywany tylko w razie potrzeby, przejdź na stronę `http://localhost:8080` i otwórz narzędzia przeglądarki F12, a następnie przejdź do zakładki `Network`. W głównym oknie przeglądarki kliknij przycisk `Dane przefiltrowane`, a zobacysz, że do serwera zostanie wysłane żądanie HTTP z prośbą o plik `0.js`, jak na rysunku 24.8 (możesz zobaczyć inną nazwę pliku, jednak nie jest to istotne).



Rysunek 24.8. Leniwe ładowanie komponentu

Podczas procesu budowania zostały utworzone dwie odrębne paczki z kodem JavaScript. Plik `app.js` zawiera główną część aplikacji, a plik `0.js` zawiera jedynie komponent `FilteredData` (pozostałe żądania pokazywane przez narzędzia F12 to początkowe żądanie HTTP, żądania danych z usługi sieciowej, a także połączenie z serwerem używane przez narzędzia deweloperskie do aktualizacji przeglądarki).

- **Wskazówka** Leniwe ładowanie w przypadku trasowania URL domyślnie dostarcza wskazówki dotyczące wstępnego pobierania. W rozdziale 21. znajdziesz opis i konfigurację związane z wyłączeniem tej funkcji.

## Wyświetlanie komponentu z komunikatem ładowania

W trakcie pisania tej książki funkcje dostępne w rozdziale 21., przeznaczone do wyświetlania komponentu ładowania lub błędu, nie są obsługiwane przez pakiet Vue Router. Aby utworzyć podobną funkcję, zamierzam połączyć właściwość magazynu danych ze strażnikami tras — w ten sposób komunikat zostanie wyświetlony użytkownikowi w trakcie leniwego ładowania komponentu. Na początek dodam do magazynu danych właściwość, która wskaże, kiedy komponent jest ładowany. Dodam także mutację w celu zmiany jej wartości (listing 24.17).

*Listing 24.17. Dodawanie właściwości danych i mutacji w pliku src/store/index.js*

```
import Vue from "vue";
import Vuex from "vuex";
import Axios from "axios";
import PrefsModule from "./preferences";
import NavModule from "./navigation";
Vue.use(Vuex);
const baseUrl = "http://localhost:3500/products/";
export default new Vuex.Store({
```

```

modules: {
 prefs: PrefsModule,
 nav: NavModule
},
state: {
 products: [],
 selectedProduct: null,
 componentLoading: false
},
mutations: {
 setComponentLoading(currentState, value) {
 currentState.componentLoading = value;
 },
 saveProduct(currentState, product) {
 let index = currentState.products.findIndex(p => p.id == product.id);
 if (index == -1) {
 currentState.products.push(product);
 } else {
 Vue.set(currentState.products, index, product);
 }
 },
 //...pozostałe funkcje magazynu danych pominięto...
}
})
)

```

Aby wskazać moment ładowania komponentu, dodaję elementy pokazane w listingu 24.18 do szablonu komponentu aplikacji, wraz z powiązaniem do właściwości magazynu danych utworzonej w listingu 24.17.

**Listing 24.18.** Wyświetlanie komunikatu ładowania w pliku src/App.vue

```

<template>
 <div class="container-fluid">
 <div class="row">
 <div class="col text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/products"
 active-class="btn-info" class="btn btn-primary">
 Produkty
 </router-link>
 <router-link tag="button" to="/preferences"
 active-class="btn-info" class="btn btn-primary">
 Preferencje
 </router-link>
 <router-link to="/named/tableleft" class="btn btn-primary"
 active-class="btn-info">
 Tabela po lewej
 </router-link>
 <router-link to="/named/tableright" class="btn btn-primary"
 active-class="btn-info">
 Tabela po prawej
 </router-link>
 <router-link to="/filter/Wszystkie" class="btn btn-primary"
 active-class="btn-info">
 Dane przefiltrowane
 </router-link>
 </div>
 </div>
 </div>
 </div>

```

```

</div>
<div class="row">
 <div class="col m-2">
 <h3 class="bg-warning text-white text-center p-2"
 v-if="componentLoading">
 Komponent ładowania...
 </h3>
 <router-view></router-view>
 </div>
</div>
</template>
<script>
 import { mapState } from "vuex";
 export default {
 name: 'App',
 computed: {
 ...mapState(["componentLoading"]),
 },
 created() {
 this.$store.dispatch("getProductsAction");
 }
 }
</script>

```

Aby ustawić wartość właściwości magazynu danych i wyświetlić komunikat użytkownikowi, dodajemy do trasy strażnika, którego celem jest komponent ładowany leniwie (listing 24.19).

**Listing 24.19.** Dodawanie strażnika trasy w pliku *src/router/basicRoutes.js*

```

import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
const FilteredData = () => import("../components/FilteredData");
import dataStore from "../store";
export default [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
 }
]

```

```

},
{
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
},
{
 path: "/filter/:category",
 component: FilteredData,
 beforeEnter: (to, from, next) => {
 dataStore.commit("setComponentLoading", true);
 next();
 }
}
]

```

Instrukcja `import` w tym przykładzie daje dostęp do magazynu danych. Atrybut `$store`, z którego korzystam w przykładach z rozdziału 20., jest dostępny tylko w komponentach, dlatego dostęp do magazynu danych w pozostałej części aplikacji wymaga użycia instrukcji `import`. Metoda strażnika w listingu 24.19 wykorzystuje mutację `setComponentLoading` do wprowadzenia zmian w magazynie danych i wywołania funkcji `next`.

## Obsługa błędów ładowania

Funkcja leniwego ładowania w przypadku trasowania adresów URL nie obsługuje komponentu błędu. Aby poradzić sobie z błędami w trakcie ładowania komponentu lub w przypadku wykonywania strażników tras, możesz skorzystać z metody `onError` zdefiniowanej w obiekcie `VueRouter` — w ten sposób zarejestrowana zostanie funkcja wywołania zwrotnego. Zostanie ona wykonana, jeżeli zaistnieje jakikolwiek problem.

Metoda strażnika zdefiniowana w listingu 24.19 wskazuje początek procesu ładowania, niemniej muszę także wiedzieć, kiedy ów proces się zakończy, aby ukryć ten komunikat. W listingu 24.20 aktualizuję strażnika tras w komponencie, który będzie ładowany leniwie.

*Listing 24.20. Aktualizacja strażnika tras w pliku src/components/FilteredData.vue*

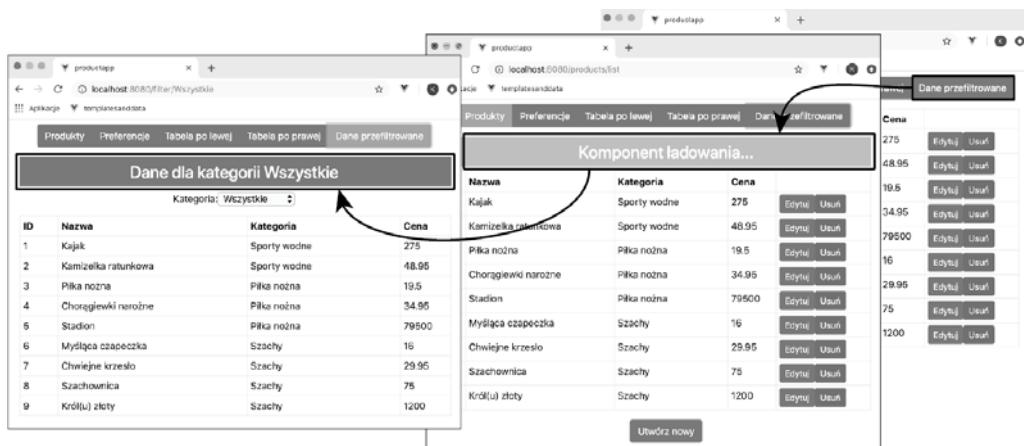
```

...
async beforeRouteEnter(to, from, next) {
 if (to.params.category != "Wszystkie") {
 next("/filter/Wszystkie");
 } else {
 next(async component => {
 component.$store.commit("setComponentLoading", false);
 await component.getData(to)
 });
 }
},
...

```

■ **Ostrzeżenie** Umieszczenie obu instrukcji mutacji w metodzie strażnika `beforeRouteGuard` może wydawać się dobrym pomysłem. Takie rozwiązanie jednak nie zadziała, ponieważ kod komponentu właśnie jest ładowany, a metoda strażnika tras nie zostanie wykonana, dopóki proces ładowania się nie zakończy.

Do funkcji wywołania zwrotnego dodałem instrukcję, która zostanie wykonana po zatwierdzeniu procesu nawigacji i utworzeniu komponentu. Korzystając z parametru `component`, aktualizuję magazyn danych i stosuję mutację, która informuje o zakończeniu procesu ładowania, co daje efekt jak na rysunku 24.9.



Rysunek 24.9. Wyświetlanie komunikatu w trakcie leniwego ładowania komponentu

## Ukrywanie odchodzącego komponentu w trakcie ładowania

Jeśli przeanalizujesz rysunek 24.9, zobaczyysz, że komponent, który zamierzamy usunąć, jest wyświetlany w trakcie procesu ładowania. Choć w wielu projektach jest to zachowanie dopuszczalne, to w innych tak nie będzie, dlatego warto zastosować dyrektywę `v-show`, aby ukryć stary komponent w trakcie oczekiwania na wczytanie nowego (listing 24.21).

Listing 24.21. Ukrywanie elementu w pliku `src/App.vue`

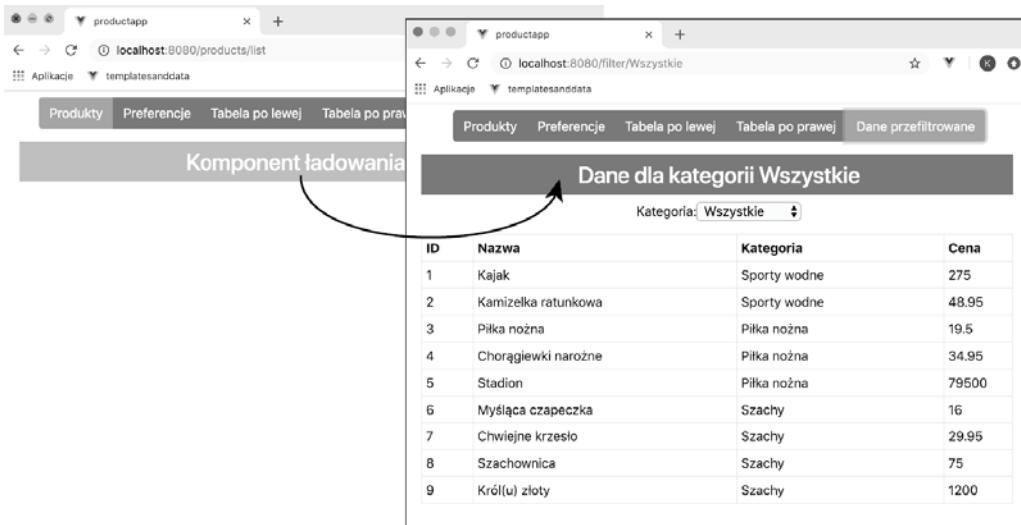
```
...
<div class="row">
 <div class="col m-2">
 <h3 class="bg-warning text-white text-center p-2"
 v-if="componentLoading">
 Komponent ładowania...
 </h3>
 <router-view v-show="!componentLoading"></router-view>
 </div>
</div>
...
```

Jak objaśniłem w rozdziale 12., dyrektywa `v-show` ukrywa element bez jego usuwania. Jest to niezwykle ważne, ponieważ w przypadku użycia dyrektyw `v-if` lub `v-else` element `router-view` zostanie usunięty z obiektowego modelu dokumentu (DOM), a załadowany komponent nie zostanie zainicjalizowany i wyświetlony użytkownikowi. Dzięki dyrektywie `v-show` element `router-view` pozostanie w dokumencie i będzie dostępny jako docelowy do wyświetlenia leniwie załadowanego komponentu (rysunek 24.10).

## Tworzenie komponentów bez obsługi trasowania

Nie wszystkie komponenty są napisane w taki sposób, aby móc skorzystać z możliwości, jakie oferuje pakiet Vue Router i udostępniane przezeń właściwości `$router` i `$route`. Jeśli zdecydujesz się skorzystać z komponentów opracowanych przez zewnętrznych dostawców, przekonasz się, że wielu z nich korzysta z mechanizmu propów opisanego w rozdziale 16. Vue Router obsługuje dostarczanie wartości propów do komponentów jako fragment konfiguracji tras, dzięki czemu możemy zintegrować komponenty z aplikacją, która korzysta z trasowania URL, bez potrzeby wprowadzania specjalnych modyfikacji.

W ramach przykładu do katalogu `src/components` dodaję plik `MessageDisplay.vue` o treści z listingu 24.22.



Rysunek 24.10. Ukrycie widoku routera w trakcie ładowania komponentu

Listing 24.22. Zawartość pliku src/components/MessageDisplay.vue

```
<template>
 <h3 class="bg-success text-white text-center p-2">
 Komunikat: {{ message }}
 </h3>
</template>
<script>
 export default {
 props: ["message"]
 }
</script>
```

Ten komponent wyświetla komunikat za pomocą propa, co jest w pełni wystarczające do realizacji tego przykładu. W listingu 24.23 do konfiguracji aplikacji dodaje dwie trasy, których celem jest nowy komponent. Konfiguruje je z wykorzystaniem różnych wartości propów.

Listing 24.23. Dodawanie tras do pliku src/router/basicRoutes.js

```
import ProductDisplay from "../components/ProductDisplay";
import ProductEditor from "../components/ProductEditor";
import Preferences from "../components/Preferences";
import Products from "../components/Products";
import MessageDisplay from "../components/MessageDisplay";
const FilteredData = () => import("../components/FilteredData");

import dataStore from "../store";
export default [
 {
 path: "/preferences",
 component: Preferences
 },
 {
 path: "/products",
 component: Products,
 children: [
 {
 name: "table",
```

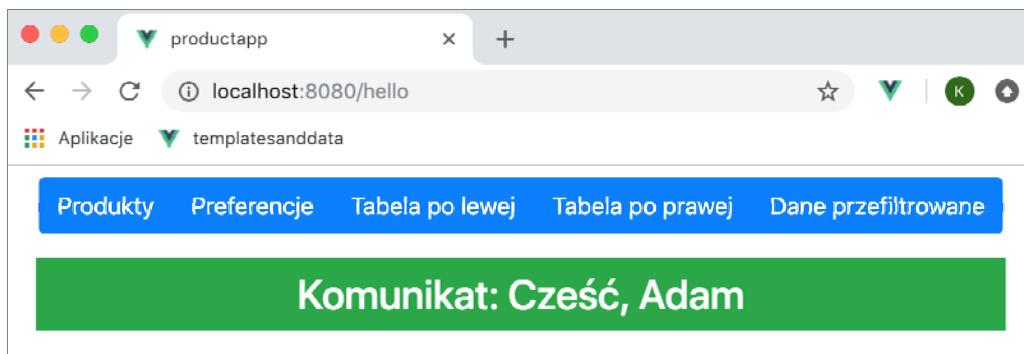
```

 path: "list",
 component: ProductDisplay
 },
 {
 name: "editor",
 path: ":op(create|edit)/:id(\d+)?",
 component: ProductEditor
 },
 {
 path: "",
 redirect: "list"
 }
]
},
{
 path: "/edit/:id",
 redirect: to => `/products/edit/${to.params.id}`
},
{
 path: "/filter/:category",
 component: FilteredData,
 beforeEnter: (to, from, next) => {
 dataStore.commit("setComponentLoading", true);
 next();
 }
},
{
 path: "/hello",
 component: MessageDisplay,
 props: {
 message: "Cześć, Adam"
 }
},
{
 path: "/hello/:text",
 component: MessageDisplay,
 props: (route) => ({
 message: `Cześć, ${route.params.text}`
 })
},
{
 path: "/message/:message",
 component: MessageDisplay,
 props: true
},
]

```

Propy są przekazywane do komponentu za pomocą właściwości `props` w momencie tworzenia trasy. Trasy dodane w listingu 24.23 demonstrują trzy sposoby przekazywania propów do komponentu za pomocą właściwości `props`. W pierwszej trasie wartość propa jest całkowicie niezależna od trasy i zawsze będzie miała taką samą wartość. Aby się o tym przekonać, przejdź na stronę <http://localhost:8080/hello>, a zobaczyś efekt jak na rysunku 24.11.

Aby ustawić wartość propa, pozostałe dwie trasy korzystają z wartości dynamicznego segmentu ścieżki. Wartością właściwości `props` może być funkcja, która otrzymuje bieżącą trasę w formie parametru i zwraca obiekt zawierający wartości propów:



Rysunek 24.11. Przekazywanie propa o stałej wartości

```
...
{ path: "/hello/:text", component: MessageDisplay,
 props: (route) => ({ message: `Cześć, ${route.params.text}`}),
...

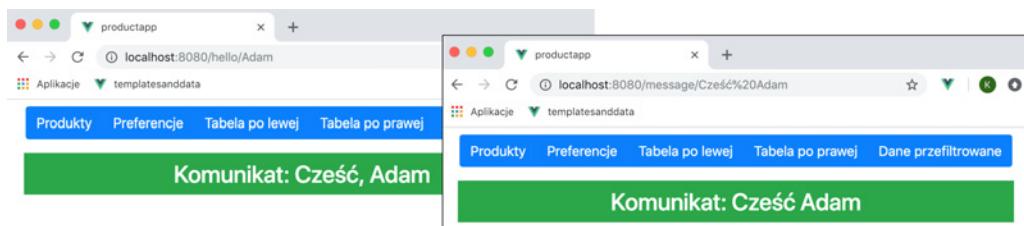
```

W tym przykładzie korzystamy z wartości segmentu `text` i definiujemy wartość propa `message`. Jest to użyteczny mechanizm, pozwalający na przetworzenie wartości z adresu URL. Jeśli nie chcesz przetwarzania segmentu dynamicznego, możesz zastosować ostatni wariant, który ustawia wartość właściwości `props` na `true`:

```
...
{ path: "/message/:message", component: MessageDisplay, props: true},
...

```

W ten sposób korzystamy z właściwości `params` aktualnej trasy jako wartości propów. Unikamy konieczności jawnego zdefiniowania powiązań dla każdego z dynamicznych segmentów i propów, choć należy pamiętać, że nazwy segmentów muszą być identyczne z nazwami propów oczekiwany przez komponent. Aby zobaczyć efekt, przejdź na stronę `http://localhost:8080/hello/Adam` i `http://localhost:8080/message/Cześć%20Adam`, co powinno zaowocować efektem jak na rysunku 24.12.



Rysunek 24.12. Powiązanie wartości segmentów dynamicznych z propami komponentów

## Podsumowanie

W tym rozdziale omówiłem sposób użycia wielu plików JavaScript do zgrupowania powiązanych tras w celu ułatwienia procesu zarządzania konfiguracją tras. Pokazałem także, jak chronić trasy przed niedozwoloną aktywacją i jak leniwie wczytywać komponenty, gdy są one wymagane przez określoną trasę. Na koniec rozdziału zademonstrowałem konfigurację propów komponentu, dzięki czemu możliwe jest zastosowanie w aplikacji komponentów bez wbudowanej obsługi systemu trasowania. W kolejnym rozdziale pokażę Ci, jak korzystać z funkcji przejść.



## ROZDZIAŁ 25.



# Przejścia

Mechanizm przejść w Vue.js pozwala na podjęcie działania w momencie, gdy element HTML zostaje dodany, usunięty lub zmienia swoje położenie. Połączenie tej funkcji z możliwościami nowoczesnych przeglądarek daje możliwość zwrócenia uwagi użytkownika na tę część aplikacji, która została zmodyfikowana w wyniku jego działań. W tym rozdziale omówię różne sposoby zastosowania przejść, pokażę, jak skorzystać z zewnętrznych arkuszy CSS i pakietów do animacji JavaScript, a także zademonstruję, jak skupić uwagę użytkownika na innego rodzaju zmianach, np. przy modyfikacji wartości danych. Tabela 25.1 umiejscowia ten rozdział w szerszym kontekście.

**Tabela 25.1.** Umiejscowienie przejść w szerszym kontekście

Pytanie	Odpowiedź
Czym są przejścia?	Przejścia to instrukcje dodawania i usuwania powiązań pomiędzy klasami a elementami, wykonywane z reguły w momencie dodawania elementu do drzewa DOM lub usuwania go z niego. Klasy są używane do stopniowego stosowania zmian w stylu elementu w celu uzyskania efektów animowanych.
Dlaczego są użyteczne?	Przejścia stanowią użyteczny sposób zwrócenia uwagi użytkownika na ważne zmiany i uprzyjemniają je.
Jak się z nich korzysta?	Przejścia stosuje się za pomocą elementów <code>transition</code> i <code>transition-group</code> .
Czy są jakieś pułapki lub ograniczenia?	Łatwo dać się ponieść fantazji i utworzyć aplikację, która zawiera zbyt dużo efektów, frustrujących użytkownika i zaburzających przepływ aplikacji.
Czy są jakieś rozwiązania alternatywne?	Przejścia są mechanizmem opcjonalnym — nie musisz z nich korzystać.

Tabela 25.2 podsumowuje rozdział.

## Przygotowania do tego rozdziału

Aby utworzyć projekt niezbędny do wykonania przykładów z tego rozdziału, wykonaj polecenie z listingu 25.1 w wybranym katalogu. Utworzysz w ten sposób nowy projekt Vue.js.

**Tabela 25.2.** Podsumowanie rozdziału

Problem	Rozwiązanie	Listing
Zastosuj przejście.	Skorzystaj z elementu transition i zdefiniuj style, które odpowiadają klasom przejść.	25.13, 25.15 – 25.16, 25.20 – 25.22
Skorzystaj z biblioteki do obsługi animacji.	Skorzystaj z atrybutów elementu transition, aby określić klasy, które zastosują animację.	25.14, 25.17
Określ, w jaki sposób zostanie zaprezentowane przejście między elementami.	Skorzystaj z atrybutu mode.	25.15
Zastosuj przejście do grupy powtarzalnych elementów.	Skorzystaj z elementu transition-group.	25.18
Otrzymuj powiadomienia przejść.	Obsłuż zdarzenia przejść.	25.19

**Listing 25.1.** Tworzenie przykładowego projektu

```
vue create transitions --default
```

To polecenie utworzy projekt o nazwie *transitions*. Po zakończeniu procesu konfiguracji wykonaj w katalogu *transitions* polecenia z listingu 25.2, aby dodać pakiety Bootstrap i Vue Router do projektu.

**Listing 25.2.** Dodawanie pakietów

```
npm install bootstrap@4.0.0
npm install vue-router@3.0.1
```

Aby dodać efekty zastosowane w przykładach z tego rozdziału, korzystam z pakietów *animate.css* i *popmotion*. Wykonaj w katalogu *transitions* polecenia z listingu 25.3, aby pobrać i zainstalować pakiety.

**Listing 25.3.** Dodawanie pakietów związanych z obsługą animacji

```
npm install animate.css@3.6.1
npm install popmotion@8.1.24
```

Dodaj instrukcje przedstawione w listingu 25.4 do pliku *main.js* w katalogu *src*, aby dołączyć Bootstrapa i pakiety animacji do aplikacji.

**Listing 25.4.** Dołączanie pakietów w pliku *src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
import "animate.css/animate.min.css";
import "popmotion/dist/popmotion.global.min.js";
Vue.config.productionTip = false
new Vue({
 render: h => h(App)
}).$mount('#app')
```

## Tworzenie komponentów

W tym rozdziale muszę skorzystać z kilku podstawowych komponentów. Zacznę od dodania do katalogu *src/components* pliku *SimpleDisplay.vue* (listing 25.5).

**Listing 25.5.** Zawartość pliku *src/components/SimpleDisplay.vue*

```
<template>
 <div class="mx-5 border border-dark p-2">
 <h3 class="bg-warning text-white text-center p-2">Wyświetlacz</h3>
 <div v-if="show" class="h4 bg-info text-center p-2">Cześć, Adam</div>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="toggle">
 Przełącz widoczność
 </button>
 </div>
 </div>
</template>
<script>
export default {
 data: function() {
 return {
 show: true
 }
 },
 methods: {
 toggle() {
 this.show = !this.show;
 }
 }
}
</script>
```

Ten komponent wyświetla zwykły komunikat. Jego widoczność jest ustalana za pomocą dyrektywy *v-if*. Następnie do katalogu *src/components* dodaję plik *Numbers.vue* o treści z listingu 25.6.

**Listing 25.6.** Zawartość pliku *src/components/Numbers.vue*

```
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div class="col h3">= {{ total }} </div>
 </div>
 </div>
 </template>
<script>
export default {
 data: function() {
 return {
 first: 0,
 second: 0
 }
 },
 computed: {
 total: function() {
 return this.first + this.second;
 }
 }
}
</script>
```

```

 first: 10,
 second: 20
 }
},
computed: {
 total() {
 return this.first + this.second;
 }
}
}
</script>

```

Ten komponent wyświetla dwa elementy `input`, które korzystają z dyrektywy `v-model` do aktualizowania właściwości danych. Właściwości te są z kolei sumowane przez właściwość obliczaną, wyświetlana również w szablonie.

Teraz do katalogu `src/components` dodaję plik `ListMaker.vue` o treści z listingu 25.7.

**Listing 25.7.** Zawartość pliku `src/components/ListMaker.vue`

```

<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-info text-white text-center p-2">Moja lista</h3>
 <table class="table table-sm">
 <tr><th>#</th><th>Element</th><th width="20%" colspan="2"></th></tr>
 <tr v-for="(item, i) in items" v-bind:key=item>
 <td>{{i}}</td>
 <td>{{item}}</td>
 <td>
 <button class="btn btn-sm btn-info" v-on:click="moveItem(i)">
 Przenieś
 </button>
 <button class="btn btn-sm btn-danger" v-on:click="removeItem(i)">
 Usuń
 </button>
 </td>
 </tr>
 <controls v-on:add="addItem" />
 </table>
 </div>
</template>
<script>
import Controls from "./ListMakerControls";
export default {
 components: {
 Controls
 },
 data: function() {
 return {
 items: ["Jabłka", "Pomarańcze", "Winogrona"]
 }
 },
 methods: {
 addItem(item) {
 this.items.push(item);
 },
 removeItem(index) {
 this.items.splice(index, 1);
 },
 }
}
</script>

```

```

 moveItem(index) {
 this.items.push(...this.items.splice(index, 1));
 }
 }
</script>

```

Ten komponent wyświetla tablicę elementów. Nowe elementy mogą być dodane do tablicy, a istniejące — przesunięte na jej koniec lub całkowicie z niej usunięte. Zmiany w szablonie komponentu wprowadzę nieco później. Aby uniknąć wyświetlania listy elementów HTML, które nie są bezpośrednio związane z przykładami, tworzę dodatkowy komponent w pliku *src/components/ListMakerControls.vue* o treści z listingu 25.8.

**Listing 25.8.** Zawartość pliku *src/components/ListMakerControls.vue*

```

<template>
 <tfoot>
 <tr v-if="showAdd">
 <td></td>
 <td><input class="form-control" v-model="currentItem" /></td>
 <td>
 <button id="add" class="btn btn-sm btn-info" v-on:click="handleAdd">
 Dodaj
 </button>
 <button id="cancel" class="btn btn-sm btn-secondary"
 v-on:click="showAdd = false">
 Anuluj
 </button>
 </td>
 </tr>
 <tr v-else>
 <td colspan="4" class="text-center p-2">
 <button class="btn btn-info" v-on:click="showAdd = true">
 Pokaż dodawanie
 </button>
 </td>
 </tr>
 </tfoot>
</template>
<script>
export default {
 data: function() {
 return {
 showAdd: false,
 currentItem: ""
 }
 },
 methods: {
 handleAdd() {
 this.$emit("add", this.currentItem);
 this.showAdd = false;
 }
 }
}
</script>

```

Ten komponent pozwala na dodawanie nowych elementów do listy. Jednocześnie jest on zależny od komponentu utworzonego w listingu 25.7.

## Konfiguracja trasowania URL

Aby skonfigurować system trasowania URL, utworzyłem katalog `src/router` i dodałem do niego plik `index.js` o treści z listingu 25.9.

*Listing 25.9. Zawartość pliku src/router/index.js*

```
import Vue from "vue"
import Router from "vue-router"
import SimpleDisplay from "../components/SimpleDisplay";
import ListMaker from "../components/ListMaker";
import Numbers from "../components/Numbers";
Vue.use(Router)
export default new Router({
 mode: "history",
 routes: [
 { path: "/display", component: SimpleDisplay },
 { path: "/list", component: ListMaker },
 { path: "/numbers", component: Numbers },
 { path: "*", redirect: "/display" }
]
})
```

Ta konfiguracja włącza tryb API historii i definiuje trasy `/display`, `/numbers` i `/list`, których celem są komponenty opracowane w poprzednim podrozdziale. Występuje także trasa typu `catchall`, która przekierowuje przeglądarkę pod adres `/display`. W listingu 25.10 importuję router w pliku `main.js`, a także dodaję właściwość, która włącza funkcje trasowania.

*Listing 25.10. Włączanie trasowania w pliku src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
import "animate.css/animate.min.css";
import "popmotion/dist/popmotion.global.min.js";
import router from "./router";
Vue.config.productionTip = false
new Vue({
 router,
 render: h => h(App)
}).$mount('#app')
```

## Tworzenie elementów nawigacji

Ostatnim krokiem przygotowawczym jest dodanie do szablonu komponentu głównego elementów nawigacji, których celami będą adresy URL zdefiniowane w konfiguracji trasowania (listing 25.11).

*Listing 25.11. Dodawanie elementów nawigacji w pliku src/App.vue*

```
<template>
 <div class="m-2">
 <div class="text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/display"
 exact-active-class="btn-warning" class="btn btn-secondary">
 Prosty wyświetlacz
 </router-link>
```

```

<router-link tag="button" to="/list"
 exact-active-class="btn-info" class="btn btn-secondary">
 Generator list
</router-link>
<router-link tag="button" to="/numbers"
 exact-active-class="btn-success" class="btn btn-secondary">
 Liczby
</router-link>
</div>
</div>
<router-view />
</div>
</template>
<script>
export default
{
 name: 'App'
}
</script>

```

Wykonaj polecenie z listingu 25.12 w katalogu *transitions*, aby uruchomić narzędzia deweloperskie.

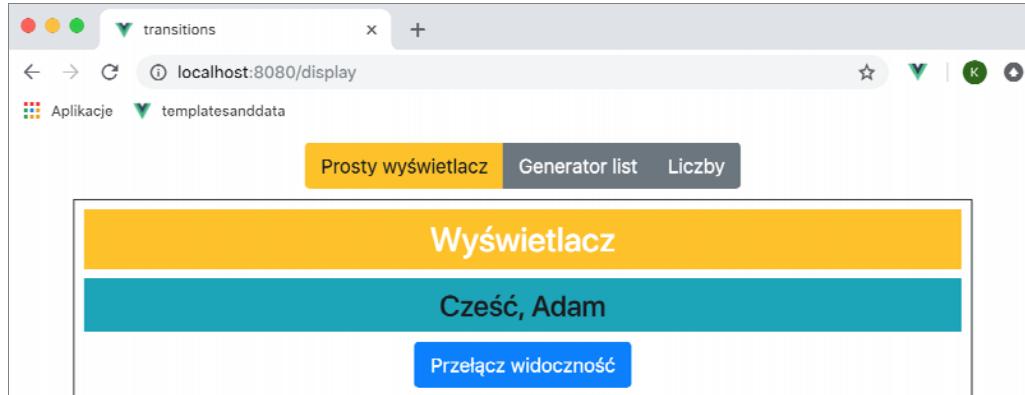
#### **Listing 25.12. Uruchamianie narzędzi deweloperskich**

---

```
npm run serve
```

---

Po zakończeniu procesu inicjalizacji wejdź na stronę <http://localhost:8080>, a zobaczysz treść podobną do tej z rysunku 25.1.



**Rysunek 25.1.** Uruchomienie przykładowej aplikacji

### Jak stosować przejścia?

Programiści często dają się ponieść fantazji w zakresie stosowania przejść, w wyniku czego powstają aplikacje irytujące użytkowników. Tego rodzaju funkcje powinny być stosowane oszczędnie, a także w prosty sposób. Przede wszystkim jednak powinny one działać szybko. Stosuj przejścia, aby ułatwić zrozumienie działania Twojej aplikacji, a nie po to, aby zademonstrować swoje zdolności artystyczne. Użytkownicy, zwłaszcza korporacyjni, muszą wykonywać te same zadania raz po raz, tak więc długo trwające animacje będą im tylko przeszkadzać.

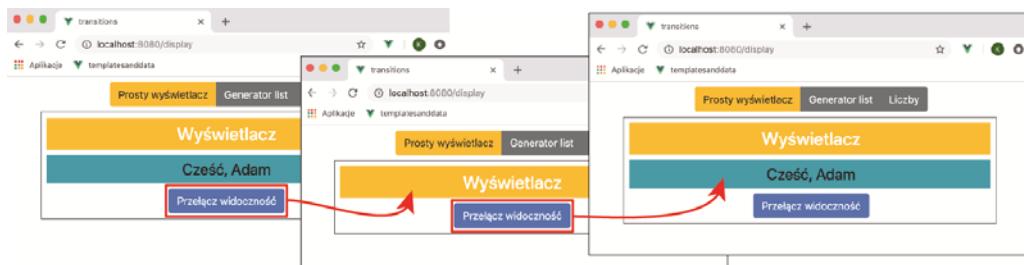
Sam cierpię na tę przypadłość — jeśli ktoś mnie nie przypilnuje, moje aplikacje wyglądają jak maszyny do jednorękiego bandyty w Las Vegas! Staram się jednak pamiętać o dwóch regułach, które pozwalają mi się opanować. Po pierwsze, wykonuję wszystkie istotne procesy w aplikacji 20 razy z rzędu. W przykładowej aplikacji może oznaczać to dodanie 20 elementów do listy, ich przemieszczanie i usuwanie. Usuwam lub skracam każdy efekt, gdy muszę czekać na jego zakończenie, zanim przejdę do kolejnego kroku w procesie.

Po drugie, nie włączam efektów w czasie tworzenia aplikacji. Wydaje się to niezwykle kuszące i można to dobrze uzasadnić — w końcu wykonuję tylko krótkie testy napisanego przez siebie kodu. Dzięki zachowaniu animacji w czasie procesu tworzenia aplikacji zauważę wszelkie nadmiarowe efekty, które dostrzeże również użytkownik. W związku z tym zostawiam wszelkie przejścia i dopasowuję je (zazwyczaj skracając czas ich trwania), aż przestaną być irytujące.

Nie musisz przestrzegać moich reguł, ale niezwykle ważne jest, aby pamiętać, żeby wszelkie efekty były dla użytkownika pomocą, a nie barierą w produktywnej pracy czy też irytującą uciążliwością.

## Rozpoczynamy pracę z przejściami

Domyślnie zmiany wprowadzone w elementach HTML szablonu komponentu są wdrażane natychmiast. Widać to np. po kliknięciu przycisku *Przelącz widoczność*, wyświetlanego w ramach komponentu *SimpleDisplay*. Po każdym kliknięciu przycisku dochodzi do modyfikacji właściwości danych *show*, co wpływa na dyrektywę *v-if*, która pokazuje i ukrywa powiązany ze sobą element (rysunek 25.2).



Rysunek 25.2. Domyślne zachowanie w przypadku zmiany stanu

- **Uwaga** Statyczne zrzuty ekranu nie oddają dobrze zmian w aplikacji. Z tego względu przykłady przedstawione w tym rozdziale najlepiej zrealizować samemu — w ten sposób najbardziej docenisz działanie mechanizmu przejść w Vue.js.

Funkcja przejść w Vue.js może być używana do zarządzania zmianą pomiędzy dwoma stanami za pomocą komponentu *transition*. Typowym przykładem użycia tego komponentu jest zastosowanie przejścia CSS, co wykonuję w listingu 25.13.

*Listing 25.13. Zastosowanie przejścia w pliku src/components/SimpleDisplay.vue*

```
<template>
 <div class="mx-5 border border-dark p-2">
 <h3 class="bg-warning text-white text-center p-2">Wyświetlacz</h3>
 <transition>
 <div v-if="show" class="h4 bg-info text-center p-2">Cześć, Adam</div>
 </transition>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="toggle">
 Przelącz widoczność
 </button>
 </div>
 </div>
</template>
<script>
 export default {
 data() {
 return {
 show: true
 }
 },
 methods: {
 toggle() {
 this.show = !this.show
 }
 }
 }
</script>
```

```

 </button>
 </div>
</div>
</template>
<script>
export default {
 data: function() {
 return {
 show: true
 }
 },
 methods: {
 toggle() {
 this.show = !this.show;
 }
 }
}
</script>
<style>
.v-leave-active {
 opacity: 0;
 font-size: 0em;
 transition: all 250ms;
}
.v-enter {
 opacity: 0;
 font-size: 0em;
}
.v-enter-to {
 opacity: 1;
 font-size: x-large;
 transition: all 250ms;
}
</style>

```

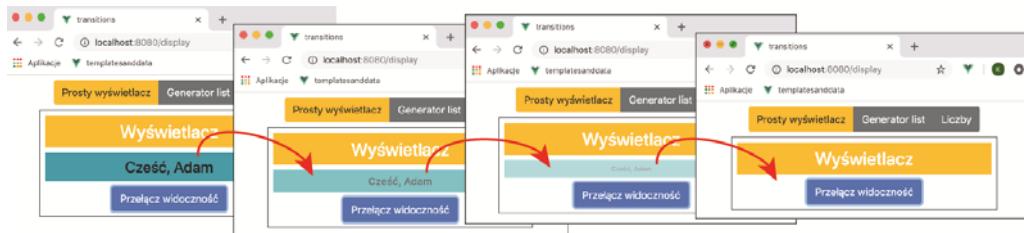
## Przejścia Vue.js a przejścia i animacje CSS

W tym rozdziale mamy do czynienia ze zbieżnością pojęć. Funkcja przejść Vue.js jest używana do reagowania na zmiany w stanie aplikacji. Z reguły wiąże się to z dodawaniem i usuwaniem elementów modelu DOM, ale może też dotyczyć zmiany wartości danych.

Typową reakcją na zmianę w elemencie HTML jest zastosowanie **mechanizmu przejścia CSS** (ang. *CSS transition*). Przejście CSS to płynna zmiana pomiędzy dwoma zbiorami wartości CSS, co daje efekt animacji. Przykład przejścia CSS jest przedstawiony w listingu 25.13. Animacja CSS jest podobna do przejścia CSS, ale w większym stopniu wpływa na to, w jaki sposób wykonywana jest animacja. Możesz też spotkać się z pojęciem **transformacji CSS** (ang. *CSS transform*), które pozwalają na **przesunięcie** (ang. *move*), **obrót** (ang. *rotate*), **skalowanie** (ang. *scale*) i **pochylenie** (ang. *skew*) elementów HTML. Transformacje często łączy się z przejściami i animacjami CSS w celu uzyskania bardziej zaawansowanych efektów.

Jeśli te pojęcia przyprawiają Cię o ból głowy, pamiętaj, że to przejścia Vue.js są najważniejsze w tworzeniu tego rodzaju aplikacji. Poza wykonaniem kilku podstawowych przykładów, które pozwalą mi zademonstrować działanie mechanizmów Vue.js, nie będziemy korzystać bezpośrednio z mechanizmów technologii CSS — skorzystamy z zewnętrznych bibliotek, które zapewniają tego rodzaju efekty. Zdecydowanie zachęcam do stosowania tego podejścia we własnych projektach, ponieważ wówczas wyniki stają się bardziej przewidywalne i jednocześnie mniej frustrujące dla programisty, niż ma to miejsce w przypadku tworzenia własnych, złożonych efektów za pomocą transformacji, przejść i animacji CSS.

Ten przykład łatwiej zrozumieć po zobaczeniu go w akcji. Po zapisaniu zmian pokazanych w listingu odśwież przeglądarkę i kliknij przycisk **Przełącz widoczność**. Nie nastąpi natychmiastowa zmiana widoczności elementu, lecz będzie się to dziać stopniowo (jak na rysunku 25.3). Na rysunku nie widać tego tak dobrze, ale element HTML staje się coraz mniejszy i stopniowo znika z widoku.



Rysunek 25.3. Efekt przejścia

- **Wskazówka** Aby sprawdzić efekt działania aplikacji, konieczne może być odświeżenie przeglądarki. Jest to konsekwencja sposobu budowania paczki przez webpacka.

Aby zastosować przejście, konieczne jest opakowanie elementu, wobec którego chcesz je zastosować, za pomocą elementu transition:

```
...
<transition>
 <div v-if="show" class="h4 bg-info text-center p-2">Hello, Adam</div>
</transition>
...
```

Vue.js nie animuje elementu samodzielnie — zamiast tego dodaje do elementu szereg klas i pozostawia przeglądarkę kwestię zastosowania wszelkich efektów związanych z tymi klasami. Nie jest to tak złożone, jak mogłoby się wydawać — rozwijam ten temat w kolejnych fragmentach rozdziału.

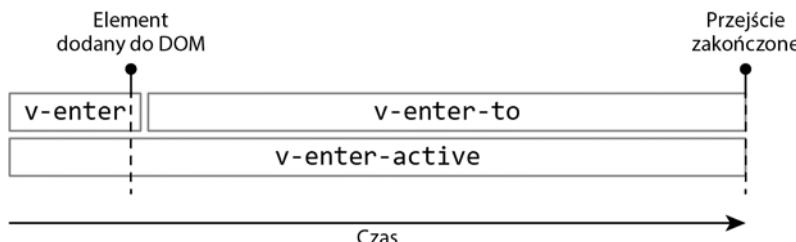
## Omówienie klas przejść i przejść CSS

Efektem zastosowania elementu transition jest przypisanie szeregu klas do elementu podczas przejścia, czyli np. w czasie dodawania go do drzewa DOM lub usuwania go z niego. Tabela 25.3 opisuje te klasy.

Tabela 25.3. Klasa przejść

Nazwa	Opis
v-enter	Element otrzymuje tę klasę przed dodaniem go do modelu DOM, po czym traci ją zaraz po tym.
v-enter-active	Element otrzymuje tę klasę przed dodaniem go do modelu DOM, po czym traci ją po zakończeniu przejścia.
v-enter-to	Element otrzymuje tę klasę tuż po dodaniu go do modelu DOM, po czym traci ją po zakończeniu przejścia.
v-leave	Element otrzymuje tę klasę na początku przejścia, a traci ją po jednej klatce.
v-leave-active	Element otrzymuje tę klasę po rozpoczęciu przejścia i traci ją po jego zakończeniu.
v-leave-to	Element otrzymuje tę klasę jedną klatkę po rozpoczęciu przejścia, a traci ją po jego zakończeniu.

Element, który podlega przejściu, jest przypisywany do klas przedstawionych w tabeli i wypisywany z nich, gdy przejście, odpowiednio, rozpoczyna się i kończy — zawsze w tej samej kolejności. Rysunek 25.4 przedstawia sekwencję przynależności do klas w momencie dodania elementu do modelu DOM, pokazując związek między procesem przejścia i klasami.



Rysunek 25.4. Przejście początkowe

Klasa `v-enter` służy do definiowania początkowego stanu elementu HTML przed jego dodaniem do modelu DOM. W listingu 25.13 definiuję styl CSS z selektorem, który dopasowuje elementy w klasie `v-enter`:

```
...
.v-enter {
 opacity: 0;
 font-size: 0em;
}
...
```

Gdy element należy do klasy `v-enter`, jego nieprzezroczystość (ang. *opacity*) wynosi 0, co oznacza, że element ten jest całkowicie przezroczysty, a tekst będzie miał zerową wysokość.

W listingu 25.13 skorzystałem ze stylu CSS zawierającego selektor, który dodaje atrybuty do klasy `v-enter-to`:

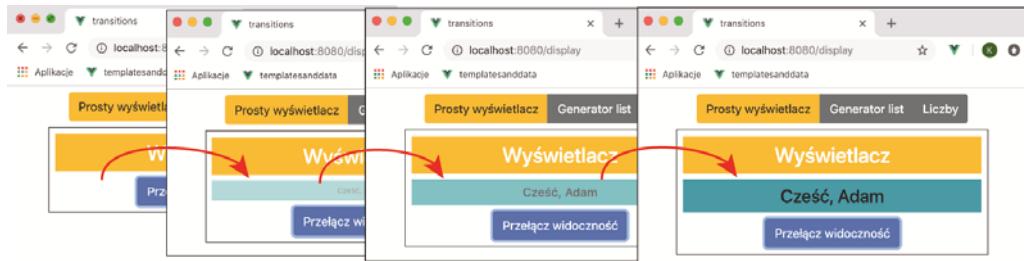
```
...
.v-enter-to {
 opacity: 1;
 font-size: x-large;
 transition: all 250ms;
}
...
```

Wartość właściwości `opacity` sprawia, że element jest w pełni nieprzezroczysty. Wartość właściwości `font-size` określa z kolei większy rozmiar tekstu. Właściwości zdefiniowane w stylu `v-enter-to` określają końcowy stan przejścia, co dla wielu programistów może wydawać się dziwne. Najważniejszą właściwością jest `transition` — nakazuje ona przeglądarce stopniową zmianę wartości wszystkich właściwości CSS zastosowanych wobec tego elementu od bieżących wartości do wartości zdefiniowanych w tym stylu, na przestrzeni 250 milisekund. Przeglądarka stopniowo zmienia wartość właściwości CSS wyrażonych w formie liczbowej, takich jak rozmiar czcionki, odstępy (ang. *padding*), marginesy, a nawet kolory.

## Omówienie sekwencji przejścia

Połączenie możliwości klas i stylów CSS tworzy przejście technologii Vue.js, które ujawnia element. W momencie kliknięcia przycisku *Przelóż widoczność* dyrektywa `v-if` określa, czy element `div` powinien znaleźć się w drzewie DOM. Element `div` jest przypisany do kilku klas Bootstrapa, które ustawnią rozmiar tekstu, tło, odstępy i inne kwestie związane z wyświetlaniem. Aby przygotować się do tego przejścia, Vue.js przypisuje element do klasy `v-enter`, która ustawia właściwości `font-size` i `opacity`. Właściwości te ustawnią początkowy stan elementu, sprawiając, że element jest przezroczysty i ma zerową wysokość.

Zaraz po dodaniu elementu do drzewa DOM element jest usuwany z klasy v-enter, a zostaje przypisany do klasy v-enter-to. Zmiana w stylu spowoduje zwiększenie nieprzezroczystości i rozmiaru tekstu na przestrzeni 250 milisekund. Efektem jest stopniowe, acz szybkie pojawianie się i zwiększanie elementu. Po upływie 250 milisekund element jest usuwany z klasy v-enter-to i jego style są obliczane wyłącznie na podstawie przynależności do klas Bootstrapa (rysunek 25.5).



Rysunek 25.5. Efekt przejścia początkowego

## Stosowanie biblioteki do obsługi animacji

Z przejść CSS, animacji i translacji można korzystać bezpośrednio, ale nie są one łatwe w użyciu. Potrzeba sporo doświadczenia i dużej liczby testów, aby uzyskać dobre wyniki w przypadku jakichś nietrywialnych efektów. Znacznie lepiej jest skorzystać z biblioteki do obsługi animacji, takiej jak pakiet *animate.css*, który dodałem na początku rozdziału. Istnieje wiele bibliotek z animacjami wysokiej jakości, które są gotowe do użycia i łatwe do zastosowania.

W listingu 25.14 zamieniam opracowane przeze mnie efekty, zastosowane wobec elementu div w komponencie SimpleDisplay, na te dostarczone przez bibliotekę *animate.css*.

*Listing 25.14. Zastosowanie animacji z biblioteki w pliku src/components/SimpleDisplay.vue*

```
<template>
 <div class="mx-5 border border-dark p-2">
 <h3 class="bg-warning text-white text-center p-2">Wyświetlacz</h3>
 <transition enter-to-class="fadeIn" leave-to-class="fadeOut">
 <div v-if="show" class="animated h4 bg-info text-center p-2">
 Cześć, Adam
 </div>
 </transition>
 <div class="text-center">
 <button class="btn btn-primary" v-on:click="toggle">
 Przełącz widoczność
 </button>
 </div>
 </div>
</template>
<script>
export default {
 data: function() {
 return {
 show: true
 }
 },
 methods: {
 toggle() {
 this.show = !this.show;
 }
 }
}
</script>
```

```

 }
 }
</script>

```

- **Wskazówka** Nie musisz korzystać z biblioteki *animate.css*, ale zdecydowanie warto od niej zacząć, zwłaszcza jeśli nie masz doświadczenia z przejściami. Pełny zakres dostępnych efektów jest opisany na stronie <https://github.com/daneden/animate.css>.

Pakiet *animate.css* oczekuje, że elementy, które chcemy poddać animacji, otrzymają klasę `animated`. W związku z tym w listingu 25.14 zastosowałem ją bezpośrednio w elemencie `div`. Aby zastosować pojedyncze efekty, korzystam z atrybutów `enter-to-class` i `leave-to-class`, które pozwalają na zmianę nazw klas dodawanych w czasie przejścia. Tabela 25.4 zawiera wszystkie atrybuty, które umożliwiają zastosowanie klas.

**Tabela 25.4. Atrybuty wyboru klas przejścia**

Nazwa	Opis
<code>enter-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-enter</code> .
<code>enter-active-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-enter-active</code> .
<code>enter-to-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-enter-to</code> .
<code>leave-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-leave</code> .
<code>leave-active-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-leave-active</code> .
<code>leave-to-class</code>	Ten atrybut służy do określenia nazwy klasy, która zostanie użyta zamiast klasy <code>v-leave-to</code> .

W listingu skorzystałem z atrybutów `enter-to-class` i `leave-to-class`, aby określić klasy animacji dostarczone przez pakiet *animate.css*. Zgodnie z nazewnictwem klasa `fadeIn` stosuje efekt pojawienia się (ang. *fade in*) elementu, a klasa `fadeOut` stosuje efekt, który powoduje jego zniknięcie (ang. *fade out*).

## Przełączanie pomiędzy wieloma elementami

Efekt przejścia można zastosować do wielu elementów wyświetlanych za pomocą różnych kombinacji dyrektyw `v-if`, `v-else-if` i `v-else`. Wymagane jest użycie pojedynczego elementu `transition`, a Vue.js przypisze automatycznie elementy do właściwych klas w miarę ich dodawania do modelu DOM lub usuwania z niego.

W listingu 25.15 dodaję element, którego widoczność jest zarządzana za pomocą dyrektywy `v-else` i który zawiera ten sam element `transition` co element, do którego została zastosowana dyrektywa `v-if`.

**Listing 25.15. Dodawanie elementu do pliku `src/components/SimpleDisplay.vue`**

```

<template>
 <div class="mx-5 border border-dark p-2">
 <h3 class="bg-warning text-white text-center p-2">Wyświetacz</h3>
 <transition enter-active-class="fadeIn"
 leave-active-class="fadeOut" mode="out-in">

```

```

<div v-if="show" class="animated h4 bg-info text-center p-2"
 key="hello">
 Witaj, Adam
</div>
<div v-else class="animated h4 bg-success text-center p-2"
 key="goodbye">
 Żegnaj, Adam
</div>
</transition>
<div class="text-center">
 <button class="btn btn-primary" v-on:click="toggle">
 Przełącz widoczność
 </button>
</div>
</div>
</template>
<script>
export default {
 data: function() {
 return {
 show: true
 }
 },
 methods: {
 toggle() {
 this.show = !this.show;
 }
 }
}
</script>

```

Gdy chcesz zastosować przejścia do wielu elementów tego samego typu (np. do dwóch elementów div), musisz użyć atrybutu key, aby Vue.js mogło rozpoznać te elementy:

```

...
<div v-if="show" class="animated h4 bg-info text-center p-2" key="hello">
 Witaj, Adam
</div>
...

```

---

■ **Wskazówka** Zwróć uwagę, że zastosowałem efekt, używając atrybutów enter-active-class i leave-active-class. Jeśli korzysta się z biblioteki do obsługi animacji w celu utworzenia przejścia między elementami, niezwykle ważne jest zastosowanie animacji w czasie całego przejścia. W przeciwnym razie mogą pojawić się usterki polegające na błyskawicznym pokazaniu się odchodzącego elementu na ułamek sekundy przed ostatecznym usunięciem go z widoku.

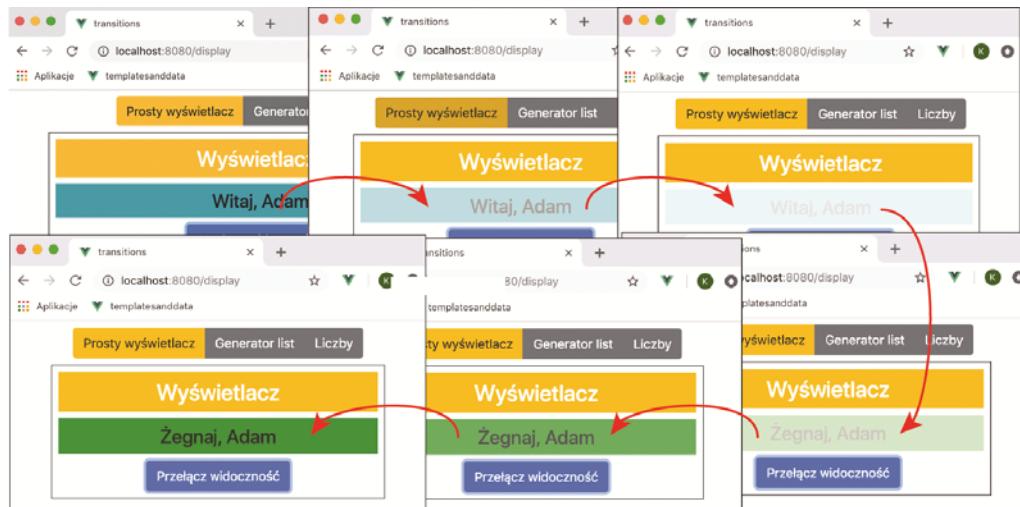
---

Vue.js domyślnie stosuje przejście do obu elementów w tym samym czasie, co oznacza, że jeden element będzie zanikał, a drugi będzie się pojawiał. Nie da to pożądaneego efektu w tym przykładzie, ponieważ jeden element powinien być zastępstwem dla drugiego. Zmieniam to zachowanie za pomocą atrybutu mode elementu transition, do którego to atrybutu można przypisać wartości z tabeli 25.5.

**Tabela 25.5. Wartości atrybutu mode**

Nazwa	Opis
in-out	Element przychodzący jest animowany najpierw, element wychodzący — później.
out-in	Element wychodzący jest animowany najpierw, element przychodzący — później.

W listingu 25.15 korzystam z trybu `out-in`, co oznacza, że Vue.js poczeka, aż element wychodzący zakończy swoje przejście przed rozpoczęciem przejścia dla elementu przychodzącego (rysunek 25.6).



Rysunek 25.6. Przejście zastosowane wobec wielu elementów

### Dostosowywanie prędkości efektów pochodzących z biblioteki animacji

Zastosowanie biblioteki do obsługi animacji jest dobrą metodą dodawania przejść, ale nie zawsze będą one dopasowane do Twoich potrzeb. Typowym problemem, który często napotykam, jest czas trwania przejść, co staje się kłopotliwe, gdy przełączasz się pomiędzy wieloma elementami i musisz poczekać na wykonanie wielu efektów.

Niektóre biblioteki do obsługi animacji pozwalają na określenie prędkości efektu, a inne — w tym `animate.css` — umożliwiają zmianę czasu przez utworzenie klasy, która ustawia właściwość `animation-duration`:

```
...
<style>
 .quick { animation-duration: 250ms }
</style>
...
```

Następnie możesz przypisać element do klasy podczas przejścia:

```
...
<transition enter-active-class="fadeIn quick"
 leave-active-class="fadeOut quick" mode="out-in">
...

```

Wszystkie przejścia będą wykonywane w określonym czasie, czyli 250 milisekund w tym przykładzie.

## Stosowanie przejścia do elementów z trasowaniem URL

To samo podejście można zastosować do zmiany elementu wyświetlonego przez element `router-view` (listing 25.16).

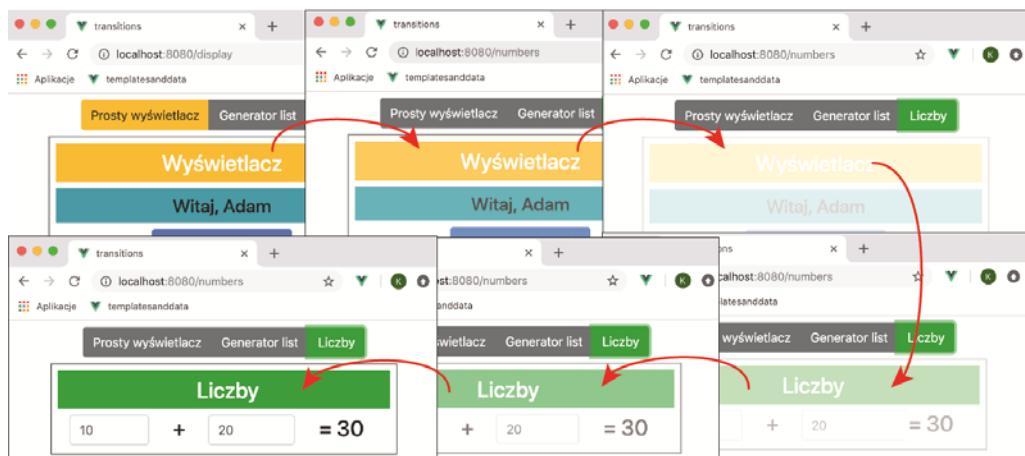
**Listing 25.16.** Zastosowanie przejścia w pliku src/App.vue

```

<template>
 <div class="m-2">
 <div class="text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/display"
 exact-active-class="btn-warning" class="btn btn-secondary">
 Prosty wyświetlacz
 </router-link>
 <router-link tag="button" to="/list"
 exact-active-class="btn-info" class="btn btn-secondary">
 Generator list
 </router-link>
 <router-link tag="button" to="/numbers"
 exact-active-class="btn-success" class="btn btn-secondary">
 Liczby
 </router-link>
 </div>
 </div>
 <transition enter-active-class="animated fadeIn"
 leave-active-class=" animated fadeOut" mode="out-in">
 <router-view />
 </transition>
 </div>
</template>
<script>
 export default
 {
 name: 'App'
 }
</script>

```

Nie musisz określić atrybutu key w przypadku elementu router-view, ponieważ system trasowania URL jest w stanie rozróżnić komponenty. Efekt przejęcia z listingu 25.16 możesz sprawdzić za pomocą przycisków nawigacji (rysunek 25.7).

**Rysunek 25.7.** Zastosowanie przejścia w elementach z trasowaniem URL

## Stosowanie przejścia podczas pojawiania się elementu

Vue.js domyślnie nie stosuje przejść względem początkowego wyglądu elementu. Aby zmienić to zachowanie, dodaj atrybut appear w elemencie transition. Standardowo Vue.js korzysta z klas enter, ale możesz użyć także klas przeznaczonych do określania początkowego wyglądu lub określenia klas za pomocą atrybutów (tabela 25.6).

**Tabela 25.6.** Klasy VueTransition dostosowane do pojawienia się elementu

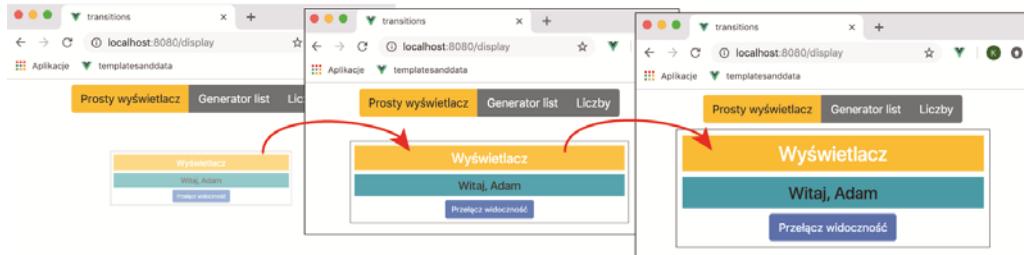
Nazwa	Opis
v-appear	Element jest przypisywany do tej klasy przed pierwszym pojawieniem się i jest usuwany zaraz po dodaniu do drzewa DOM. Własną klasę można określić za pomocą atrybutu appear-class.
v-appear-active	Element jest przypisywany do tej klasy przed pierwszym pojawieniem się i jest usuwany po zakończeniu przejścia. Własną klasę można określić za pomocą atrybutu appear-active-class.
v-appear-to	Element jest przypisywany do tej klasy zaraz po dodaniu do drzewa DOM, a usuwany po zakończeniu przejścia. Własną klasę można określić za pomocą atrybutu appear-to-class.

W listingu 25.17 dodaję przejście, które zostanie wdrożone tylko w momencie pierwszego pojawienia się elementu.

**Listing 25.17.** Dodawanie przejścia do pliku src/App.vue

```
<template>
 <div class="m-2">
 <div class="text-center m-2">
 <div class="btn-group">
 <router-link tag="button" to="/display"
 exact-active-class="btn-warning" class="btn btn-secondary">
 Prosty wyświetlacz
 </router-link>
 <router-link tag="button" to="/list"
 exact-active-class="btn-info" class="btn btn-secondary">
 Generator list
 </router-link>
 <router-link tag="button" to="/numbers"
 exact-active-class="btn-success" class="btn btn-secondary">
 Liczby
 </router-link>
 </div>
 </div>
 <transition enter-active-class="animated fadeIn"
 leave-active-class=" animated fadeOut" mode="out-in"
 appear appear-active-class="animated zoomIn">
 <router-view />
 </transition>
 </div>
</template>
<script>
 export default
 {
 name: 'App'
 }
</script>
```

Dodałem atrybut appear, który nie wymaga podania wartości, a także skorzystałem z atrybutu appear-active-class, aby przypisać element do klas animated i zoomIn w czasie przejścia. W rezultacie komponent wy wyświetlony przez element router-view zostanie przybliżony w momencie pierwszego uruchomienia aplikacji (rysunek 25.8).



Rysunek 25.8. Zastosowanie przejścia dla pierwszego pojawienia się elementu

- **Wskazówka** Aby zobaczyć to przejście, konieczne jest odświeżenie przeglądarki.

## Stosowanie przejść dla zmian w kolekcji

Vue.js obsługuje przejścia dla elementów generowanych za pomocą dyrektywy v-for, pozwalając na określenie efektów w momencie dodania, usunięcia lub przesunięcia. W listingu 25.18 przejścia stosuję wobec komponentu ListMaker.

*Listing 25.18. Zastosowanie przejścia w pliku src/components/ListMaker.vue*

```
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-info text-white text-center p-2">Moja lista</h3>
 <table class="table table-sm">
 <tr><th>#</th><th>Item</th><th width="20%" colspan="2"></th></tr>
 <transition-group enter-active-class="animated fadeIn"
 leave-active-class="animated fadeOut"
 move-class="time"
 tag="tbody">
 <tr v-for="(item, i) in items" v-bind:key=item>
 <td>{{i}}</td>
 <td>{{item}}</td>
 <td>
 <button class="btn btn-sm btn-info" v-on:click="moveItem(i)">
 Przesuń
 </button>
 <button class="btn btn-sm btn-danger"
 v-on:click="removeItem(i)">
 Usuń
 </button>
 </td>
 </tr>
 </transition-group>
 <controls v-on:add="addItem" />
 </table>
 </div>
```

```

</template>
<script>

import Controls from "./ListMakerControls";
export default {
 components: {
 Controls
 },
 data: function() {
 return {
 items: ["Jabłka", "Pomarańcze", "Winogrona"]
 }
 },
 methods: {
 addItem(item) {
 this.items.push(item);
 },
 removeItem(index) {
 this.items.splice(index, 1);
 },
 moveItem(index) {
 this.items.push(...this.items.splice(index, 1));
 }
 }
}
</script>
<style>
 .time {
 transition: all 250ms;
 }
</style>

```

Element `transition-group` wymaga zachowania ostrożności w użyciu, ponieważ — w przeciwnieństwie do elementu `transition` — dodaje element do drzewa DOM. Aby zapewnić poprawność generowanego kodu HTML, atrybut `tag` służy do określenia rodzaju elementu HTML, który zostanie wygenerowany. Skoro przykładowy komponent korzysta z dyrektywy `v-for` w celu wygenerowania wierszy tabeli, używam elementu `tbody`, który reprezentuje treść tabeli.

```

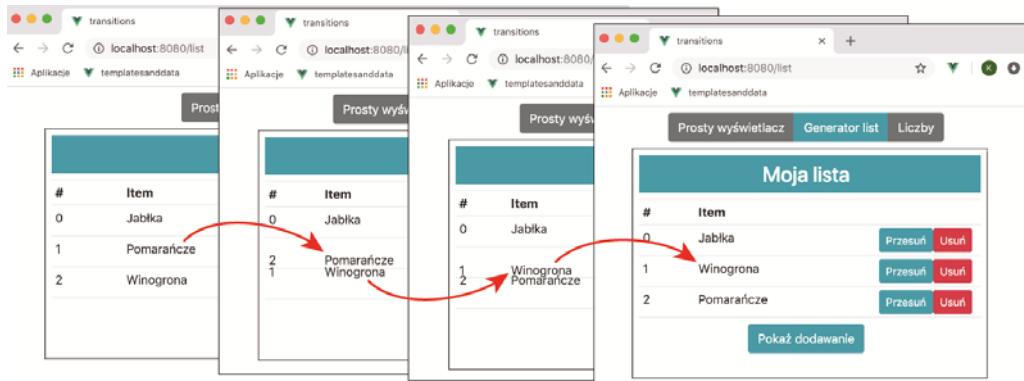
...
<transition-group enter-active-class="animated fadeIn"
 leave-active-class="animated fadeOut" move-class="time" tag="tbody">
...

```

Przejścia wejścia i wyjścia są stosowane w taki sam sposób jak w poprzednich przykładach. Skorzystałem także z atrybutów `enter-active-class` i `leave-active-class`, aby przypisać elementom wygenerowanym przez dyrektywę `v-for` klasy `animated`, `fadeIn` i `fadeOut` w miarę ich dodawania i usuwania. Ostatni atrybut to `move-class`, który przypisze klasę do elementu, gdy ten będzie przesuwany w kolekcji. Vue.js automatycznie doprowadzi do przejścia z obecnej do nowej pozycji. Jeśli jest to jedyny wymagany przez Ciebie efekt, atrybut `move-class` pozwoli na skojarzenie elementu ze stylem określającym czas, jaki zajmie przejście. W listingu wprowadzam klasę o nazwie `time` i definiuję powiązany styl CSS, który korzysta z właściwości `transition`, aby określić, że wszystkie właściwości elementu powinny zostać zmienione na przestrzeni 250 milisekund.

■ **Uwaga** Elementy zarządzane przez atrybut `transition-group` muszą mieć klucz (por. rozdział 13.).

Atrybuty, które zastosowałem w elemencie `transition-group`, sprawiły, że nowe elementy pojawiają się stopniowo na swoim miejscu, usuwane elementy znikają, a elementy przemieszczone w wyniku kliknięcia przycisku `Przesuń` przesuwają się (ostatni efekt jest pokazany na rysunku 25.9).



Rysunek 25.9. Animacja przesunięcia elementów kolekcji

## Stosowanie zdarzeń przejść

Elementy `transition` i `transition-group` generują zdarzenia, które można obsłużyć w celu zapewnienia dokładnej kontroli nad przejściami, włączając w to dostosowanie ich zachowania na podstawie stanu aplikacji. Tabela 25.7 opisuje te zdarzenia.

Tabela 25.7. Zdarzenia przejść

Nazwa	Opis
<code>before-enter</code>	Ta metoda jest wykonywana przed rozpoczęciem przejścia początkowego. Otrzymuje ona element HTML, który podlega przejściu.
<code>enter</code>	Ta metoda jest wykonywana przed rozpoczęciem przejścia początkowego. Otrzymuje ona element HTML, który podlega przejściu, a także wywołanie zwrotne. To wywołanie musi być wykonane, aby poinformować Vue.js, że przejście się zakończyło.
<code>after-enter</code>	Ta metoda jest wykonywana przed zakończeniem przejścia początkowego. Otrzymuje ona element HTML, który podlega przejściu.
<code>before-leave</code>	Ta metoda jest wykonywana przed zakończeniem przejścia końcowego. Otrzymuje ona element HTML, który podlega przejściu.
<code>leave</code>	Ta metoda jest wykonywana przed zakończeniem przejścia końcowego. Otrzymuje ona element HTML, który podlega przejściu, a także wywołanie zwrotne. To wywołanie musi być wykonane, aby poinformować Vue.js, że przejście się zakończyło.
<code>after-leave</code>	Ta metoda jest wykonywana przed zakończeniem przejścia końcowego. Otrzymuje ona element HTML, który podlega przejściu.

W listingu 25.19 dodałem funkcje obsługi zdarzeń przejść, korzystając z dyrektywy `v-on`. Funkcje te posłużyły mi do zastosowania efektu z poziomu kodu.

**Listing 25.19.** Obsługa zdarzeń przejścia w pliku *src/components/ListMakerControls.vue*

```

<template>
 <tfoot>
 <transition v-on:beforeEnter="beforeEnter"
 v-on:after-enter="afterEnter" mode="out-in">
 <tr v-if="showAdd" key="addcancel">
 <td></td>
 <td><input class="form-control" v-model="currentItem" /></td>
 <td>
 <button id="add" class="btn btn-sm btn-info"
 v-on:click="handleAdd">
 Dodaj
 </button>
 <button id="cancel" class="btn btn-sm btn-secondary"
 v-on:click="showAdd = false">
 Anuluj
 </button>
 </td>
 </tr>
 <tr v-else key="show">
 <td colspan="4" class="text-center p-2">
 <button class="btn btn-info" v-on:click="showAdd = true">
 Pokaż dodawanie
 </button>
 </td>
 </tr>
 </transition>
 </tfoot>
</template>
<script>

export default {
 data: function() {
 return {
 showAdd: false,
 currentItem: ""
 }
 },
 methods: {
 handleAdd() {
 this.$emit("add", this.currentItem);
 this.showAdd = false;
 },
 beforeEnter(el) {
 if (this.showAdd) {
 el.classList.add("animated", "fadeIn");
 }
 },
 afterEnter(el) {
 el.classList.remove("animated", "fadeIn");
 }
 }
}
</script>

```

W metodzie obsługi zdarzenia before-enter sprawdzam wartość właściwości danych showAdd, aby zobaczyć, czy zmiana powinna być animowana. W rezultacie efekt przejścia zostanie zastosowany po kliknięciu przycisku *Pokaż dodawanie*, choć przyciski *Dodaj* i *Anuluj* jeszcze nie działają.

## Stosowanie zdarzeń początkowych i końcowych

Zdarzenia enter i leave są użyteczne, gdy chcesz wykonać własne przejścia od początku do końca. Dotyczy to sytuacji, gdy chcesz wygenerować sekwencję wartości z poziomu kodu lub skorzystać z biblioteki JavaScript, która zrobi to za Ciebie. W listingu 25.20 obsługuję metodę enter poprzez przypisanie elementu HTML do klas biblioteki *animate.css*, a następnie nasłuchiwanie zdarzenia DOM, które poinformuje o zakończeniu animacji.

**Listing 25.20.** Zastosowanie zdarzenia początkowego w pliku *src/components/ListMakerControls.vue*

```
<template>
 <tfoot>
 <transition v-on:enter="enter" mode="out-in">
 <tr v-if="showAdd" key="addcancel">
 <td></td>
 <td><input class="form-control" v-model="currentItem" /></td>
 <td>
 <button id="add" class="btn btn-sm btn-info"
 v-on:click="handleAdd">
 Dodaj
 </button>
 <button id="cancel" class="btn btn-sm btn-secondary"
 v-on:click="showAdd = false">
 Anuluj
 </button>
 </td>
 </tr>
 <tr v-else key="show">
 <td colspan="4" class="text-center p-2">
 <button class="btn btn-info" v-on:click="showAdd = true">
 Pokaż dodawanie
 </button>
 </td>
 </tr>
 </transition>
 </tfoot>
</template>
<script>

import {
 styler,
 tween
} from "popmotion";
export default {
 data: function() {
 return {
 showAdd: false,
 currentItem: ""
 }
 },
 methods: {
 handleAdd() {
 this.$emit("add", this.currentItem);
 this.showAdd = false;
 }
 }
}


```

```

 },
 enter(el, done) {
 if (this.showAdd) {
 let t = tween({
 from: {
 opacity: 0
 },
 to: {
 opacity: 1
 },
 duration: 250
 });
 t.start({
 update: styler(el).set,
 complete: done
 })
 }
 }
 }
</script>

```

W tym przykładzie posłużyono się pakietem *popmotion*, który do animacji wykorzystuje język JavaScript, a nie sam CSS. Szczegóły działania pakietu *popmotion* nie są w tym momencie istotne — więcej informacji znajdziesz na stronie <http://popmotion.io/> — dlatego listing służy jedynie do zademonstrowania możliwości zdarzeń enter i leave w celu wykonania przejść.

- 
- **Uwaga** Zwróć uwagę, że metoda enter z listingu 25.20 wprowadza parametr done. Jest to funkcja wywołania zwracanego, która służy do poinformowania Vue.js o zakończeniu przejścia. Vue.js nie wykona zdarzenia after-event, dopóki funkcja done nie zostanie zakończona. Trzeba więc upewnić się, że wywołuje się tę metodę bezpośrednio lub przekazuje ją do biblioteki JavaScript, co zrobitem w tym listingu.
- 

## Przyciąganie uwagi do innych zmian

Jeśli chcesz przyciągnąć uwagę użytkownika do zmian, jakie zachodzą w danych aplikacji, możesz utworzyć obserwatora, który będzie reagował na nowe wartości. W listingu 25.21 wykorzystuję pakiet *popmotion* do tego, aby utworzyć efekt przejścia, gdy użytkownik wprowadza nowe wartości w elementach input, wyświetlane za pomocą komponentu Numbers.

*Listing 25.21. Reagowanie na zmiany w pliku src/components/Numbers.vue*

```

<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div class="col h3">= {{ displayTotal }} </div>
 </div>
 </div>
 </div>

```

```

 </div>
 </div>
</template>
<script>
import {
 tween
} from "popmotion";
export default {
 data: function() {
 return {
 first: 10,
 second: 20,
 displayTotal: 30
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 },
 watch: {
 total(newVal, oldVal) {
 let t = tween({
 from: Number(oldVal),
 to: Number(newVal),
 duration: 250
 });
 t.start((val) => this.displayTotal = val.toFixed(0));
 }
 }
}
</script>

```

Gdy wartość właściwości obliczanej `total` ulegnie zmianie, obserwator zareaguje, korzystając z pakietu `popmotion` w celu wygenerowania serii liczb ulokowanych między starymi i nowymi wartościami, z których każda zostanie użyta do aktualizacji właściwości `displayTotal` wyświetlonej w szablonie komponentu.

Ten rodzaj zmiany można połączyć z animacją CSS za pomocą właściwości `$el`, aby uzyskać element DOM komponentu. W ten sposób znajdziesz element, który chcesz zanimować, a następnie dodajesz do niego odpowiednie klasy, jak w listingu 25.22.

*Listing 25.22. Dodawanie animacji w pliku src/components/Numbers.vue*

```

<template>
<div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div id="total" class="col h3">= {{ displayTotal }} </div>
 </div>
 </div>
</div>

```

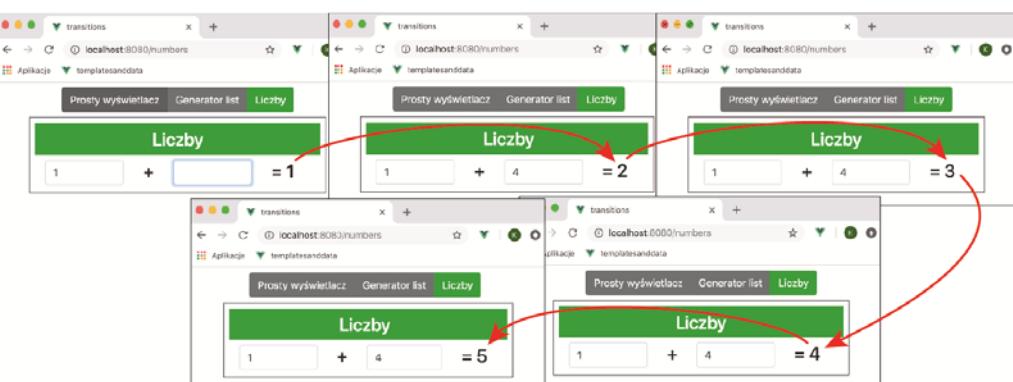
```

</template>
<script>

import {
 tween
} from "popmotion";
export default {
 data: function() {
 return {
 first: 10,
 second: 20,
 displayTotal: 30
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 },
 watch: {
 total(newVal, oldVal) {
 let classes = ["animated", "fadeIn"]
 let totalElem = this.$el.querySelector("#total");
 totalElem.classList.add(...classes);
 let t = tween({
 from: Number(oldVal),
 to: Number(newVal),
 duration: 250
 });
 t.start({
 update: (val) => this.displayTotal = val.toFixed(0),
 complete: () => totalElem.classList.remove(...classes)
 });
 }
 }
}
</script>

```

Aby upewnić się, że animacja może być zastosowana ponownie, usuwam klasy animacji z elementu po zakończeniu zmiany. Dzięki temu nowe wyniki wyświetlane są z pomocą płynnego przejścia, jak na rysunku 25.10, choć efekt ten jest trudny do uchwycenia na zrzucie.



Rysunek 25.10. Zastosowanie obserwatora w celu reagowania na zmiany wartości

## Podsumowanie

W tym rozdziale omówilem różne metody stosowania przejść w Vue.js. Pokazałem, jak korzystać z elementów `transition` i `transition-group`, jak przypisać klasy do elementów, jak używać pakietów zewnętrznych, a także jak reagować na zdarzenia przejść. Omówiłem też metody przyciągania uwagi na inne rodzaje zmian, np. zmiany wartości danych. W kolejnym rozdziale opiszę inne sposoby rozszerzenia możliwości Vue.js.

## ROZDZIAŁ 26.



# Rozszerzanie możliwości Vue.js

Vue.js zapewnia wszystkie mechanizmy, które są niezbędne w większości projektów aplikacji webowych. Czasami może jednak pojawić się konieczność rozszerzenia Vue.js. W takiej sytuacji można skorzystać z jednej z kilku dostępnych technik, które omawiam w tym rozdziale. Pokażę w nim, jak rozszerzyć możliwości wbudowanych dyrektyw za pomocą własnego kodu, jak zdefiniować wspólne funkcje do użycia w wielu komponentach za pomocą domieszek (ang. *mixins*), a także jak zgrupować szerszy zbiór powiązanych funkcji za pomocą wtyczki. Tabela 26.1 umiejscowia ten rozdział w szerszym kontekście.

**Tabela 26.1.** Umiejscowienie funkcji Vue.js w szerszym kontekście

Pytanie	Odpowiedź
Czym są funkcje Vue.js?	Funkcje opisane w tym rozdziale pozwalają na rozszerzenie możliwości oferowanych standardowo przez Vue.js.
Dlaczego są użyteczne?	Funkcje są użyteczne, jeśli dysponujesz pewną ilością kodu, który jest współdzielony w całej aplikacji i dla którego nie możesz skorzystać z wstrzykiwania zależności. Funkcje te są użyteczne również, jeśli chcesz stworzyć pewien zakres funkcjonalny, który jest wymagany w wielu projektach.
Jak się z nich korzysta?	Dyrektywy są tworzone za pomocą szeregu funkcji wywoływanych w momencie zmiany stanu powiązanego z danym elementem. Domieszki definiuje się jako grupę funkcji łączonych z komponentem w momencie utworzenia nowej instancji. Wtyczki to moduły języka JavaScript, które zawierają szeroki zakres funkcji Vue.js używanych w całej aplikacji.
Czy są jakieś pułapki lub ograniczenia?	Wszystkie omawiane funkcje są dość zaawansowane i powinny być stosowane ostrożnie. Nie są wymagane w większości przypadków. Zanim skorzystasz z tych mechanizmów, zastanów się, czy nie prościej będzie użyć funkcji poznanych we wcześniejszych rozdziałach.
Czy są jakieś rozwiązania alternatywne?	Wszystkie opisane mechanizmy są opcjonalne i nie są wymagane w większości projektów — standardowe możliwości funkcjonalne Vue.js okazują się z reguły wystarczające.

Tabela 26.2 podsumowuje rozdział.

**Tabela 26.2.** Podsumowanie rozdziału

Problem	Rozwiążanie	Listing
Zdefiniuj własną dyrektywę.	Zaimplementuj jedną lub więcej funkcji haków i zarejestruj dyrektywę za pomocą właściwości <code>directives</code> komponentu.	26.7 – 26.8, 26.16 – 26.18
Pobierz informacje o tym, w jaki sposób własna dyrektywa została zastosowana.	Odczytaj właściwości powiązanego obiektu.	26.9 – 26.14
Przekaż dane pomiędzy funkcjami haków.	Skorzystaj z właściwości danych elementu HTML, do którego komponent został zastosowany.	26.15
Zdefiniuj podstawowe funkcje dla komponentów.	Zdefiniuj domieszkę.	26.19 – 26.22
Utwórz zbiór powiązanych mechanizmów.	Utwórz wtyczkę.	26.23 – 26.31

## Przygotowania do tego rozdziału

Aby utworzyć projekt niezbędny do wykonania przykładów z tego rozdziału, wykonaj polecenie z listingu 26.1 w dogodnej lokalizacji.

### *Listing 26.1. Tworzenie przykładowego projektu*

```
vue create extendingvue --default
```

Po zakończeniu inicjalizacji projektu uruchom polecenie z listingu 26.2 w katalogu `extendingvue`, aby dodać pakiet Bootstrap do projektu.

### *Listing 26.2. Dodawanie pakietu Bootstrap*

```
npm install bootstrap@4.0.0
```

Dodaj instrukcje z listingu 26.3 do pliku `src/main.js`, aby dołączyć pakiet Bootstrap do aplikacji.

### *Listing 26.3. Dołączanie pakietu Bootstrap do pliku src/main.js*

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false
new Vue({
 render: h => h(App)
}).$mount('#app')
```

Do katalogu `src/components` dodaję plik `Numbers.vue` o treści z listingu 26.4.

### *Listing 26.4. Zawartość pliku src/components/Numbers.vue*

```
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
```

```

<div class="col">
 <input class="form-control" v-model.number="first" />
</div>
<div class="col-1 h3">+</div>
<div class="col">
 <input class="form-control" v-model.number="second" />
</div>
<div class="col h3">= {{ total }} </div>
</div>
</div>
</template>
<script>
export default {
 data: function() {
 return {
 first: 10,
 second: 20
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 }
}
</script>

```

Jest to ten sam komponent, z którego skorzystałem na początku rozdziału 25., ale bez przejścia, które dodałem w kolejnych przykładach. Aby zintegrować komponent z aplikacją, zamień zawartość pliku *App.vue* na tę z listingu 26.5.

#### ***Listing 26.5. Zawartość pliku src/App.vue***

```

<template>
<div class="m-2">
 <numbers />
</div>
</template>
<script>
import Numbers from "./components/Numbers"
export default {
 name: 'App',
 components: { Numbers }
}
</script>

```

Wykonaj polecenie z listingu 26.6 w katalogu *transitions*, aby uruchomić narzędzia deweloperskie.

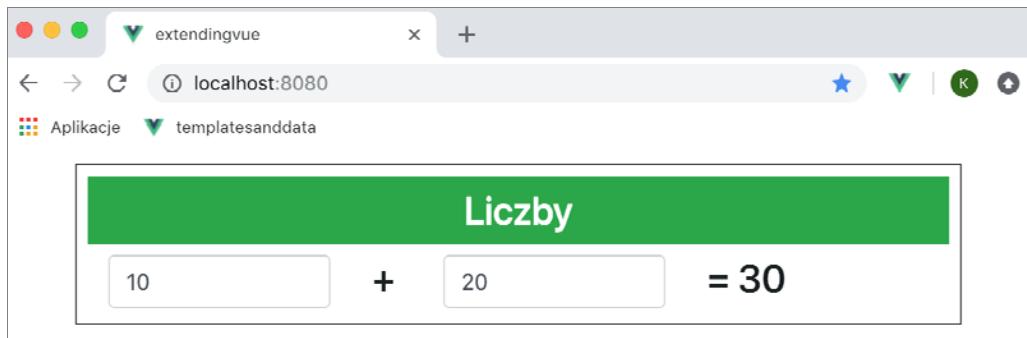
#### ***Listing 26.6. Uruchamianie narzędzi deweloperskich***

---

```
npm run serve
```

---

Po zakończeniu tworzenia paczki z aplikacją otwórz przeglądarkę i przejdź pod adres <http://localhost:8080>, a zobaczysz efekt jak na rysunku 26.1.



Rysunek 26.1. Uruchomienie przykładowej aplikacji

## Tworzenie własnych dyrektyw

Wbudowane dyrektywy, dostarczone przez Vue.js, pozwalają na realizację kluczowych zadań wymaganych w większości aplikacji. Nic nie stoi na przeszkodzie, aby utworzyć własne dyrektywy, jeśli chcesz pracować bezpośrednio na elementach HTML generowanych w aplikacji i ta potrzeba dotyczy całej aplikacji. Do nowo utworzonego katalogu `src/directives` dodaję plik `colorize.js` o treści z listingu 26.7.

*Listing 26.7. Zawartość pliku `src/directives/colorize.js`*

```
export default {
 update(el, binding) {
 if (binding.value > 100) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 }
}
```

---

■ **Wskazówka** Zwróć uwagę, że utworzyłem plik JavaScript. Tylko komponenty są tworzone w plikach `.vue`, co pozwala na łączenie kodu HTML, CSS i JavaScript. Dyrektywy tworzą się wyłącznie w JavaScriptie.

---

Zasadę działania własnych dyrektyw omówię niebawem. Teraz, zanim przedstawię dokładnie, jak odbywa się cały ten proces, warto zapoznać się z efektem działania kodu. W listingu 26.8 rejestruję dyrektywę i stosuję ją do elementu HTML.

*Listing 26.8. Rejestrowanie i stosowanie dyrektywy w pliku `src/components/Numbers.vue`*

```
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 </div>
 </div>
 </div>
```

```

 </div>
 <div v-colorize="total" class="col h3">= {{ total }} </div>
 </div>
</div>
</div>
</template>
<script>
import Colorize from "../directives/colorize";
export default {
 data: function() {
 return {
 first: 10,
 second: 20
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 },
 directives: {
 Colorize
 }
}
</script>

```

## Dlaczego najprawdopodobniej nie potrzebujesz własnej dyrektywy?

Podstawowym składnikiem aplikacji Vue.js jest komponent. To właśnie za pomocą komponentów powinno się dodawać nowe funkcje do projektu. Dyrektywy są bardziej skomplikowane w obsłudze, mają mniejsze możliwości, a także mogą wymagać bezpośredniej pracy z elementami HTML za pomocą API języka JavaScript, co może być niezwykle męczącym procesem.

Dyrektwy są użyteczne, gdy chcesz modyfikować elementy HTML niskopoziomowo, jednak często jest to możliwe również za pomocą dyrektyw wbudowanych — większość zmian w kodzie HTML jest możliwa przy użyciu dyrektywy takiej jak `v-bind`, którą opisałem w rozdziale 12. Jeśli chcesz utworzyć własną dyrektywę, poświęć chwilę na próbę osiągnięcia tego samego efektu za pomocą innych funkcji Vue.js.

---

Własne dyrektywy są rejestrowane za pomocą właściwości `directives`, która otrzymuje obiekt. W tym przykładzie korzystam ze słowa kluczowego `import`, aby nadać dyrektywie nazwę `Colorize`. Następnie stosuję ją w szablonie komponentu, poprzedzając nazwę prefiksem `v-`:

```

...
<div v-colorize="total" class="col h3">= {{ total }} </div>
...

```

Właściwość `total` przekazałem jako wartość atrybutu. Powróćmy do niej przy okazji objaśniania zasad działania dyrektywy. Aby sprawdzić dyrektywę, odśwież przeglądarkę i wprowadź wartości, które dają sumę większą niż 100. W wyniku tej operacji kolor tła i kolor tekstu ulegną zmianie (rysunek 26.2).



Rysunek 26.2. Efekt zastosowania własnej dyrektywy

## Omówienie zasady działania dyrektyw

Dyrektyny definiują metody znane pod nazwą **funkcji haków** (ang. *hook functions*), które są wykonywane w kluczowych momentach cyklu życia komponentu, do którego szablonu dyrektywa została zastosowana. Tabela 26.3 opisuje funkcje haków dyrektyw.

Tabela 26.3. Funkcje haków dyrektyw

Nazwa	Opis
bind	Ta metoda jest wywoływana przy inicjalizacji dyrektywy, dając możliwość wykonania początkowych zadań.
inserted	Ta metoda jest wywoływana, gdy element, do którego dyrektywa została zastosowana, zostaje wstawiony do swojego rodzica.
update	Ta metoda jest wywoływana, gdy komponent, którego szablon zawiera element powiązany z dyrektywą, został zaktualizowany. Ta metoda może być wywoływana przed aktualizacją dzieci komponentu.
componentUpdated	Ta metoda jest wywoywana, gdy komponent, którego szablon zawiera element powiązany z dyrektywą, został zaktualizowany i po aktualizacji jego dzieci.
unbind	Ta metoda jest wywoływana, aby dać możliwość wyczyszczenia zasobów przed odwiązaniem dyrektywy od elementu.

W listingu 26.7 własna dyrektywa implementuje hak update, który pozwala na aktualizację elementu HTML w momencie aktualizacji komponentu zawierającego powiązany z dyrektywą element HTML. Po wprowadzeniu nowej wartości do jednego z elementów input zmiana wyzwala aktualizację komponentu, co prowadzi do wywołania funkcji haka update dyrektywy. W ten sposób dyrektywa otrzymuje możliwość zmodyfikowania elementu HTML, do którego została zastosowana.

Pierwszym argumentem funkcji haków jest obiekt `HTMLElement`, który implementuje standardowe API modelu DOM i może być użyty do zmodyfikowania treści HTML przedstawianej użytkownikowi. Skorzystałem z tych obiektów w funkcji haka, aby dodawać i usuwać klasy związane ze stylami Bootstrapa:

```
...
export default {
 update(el, binding) {
 if (binding.value > 100) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 }
}
...
```

Jest to standardowe API modelu DOM, którego nie opisuję w niniejszej książce, ale o którym dowiesz się więcej na stronie <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>.

## Globalne rejestrowanie dyrektyw

W listingu 26.8 zarejestrowałem własną dyrektywę i zastosowałem ją w jednym komponencie. Dyrektywy mogą także być rejestrowane globalnie, dzięki czemu nie trzeba rejestrować ich dla pojedynczych komponentów. Globalną rejestrację przeprowadza się w pliku *main.js*, korzystając z metody *Vue.directive*:

```
...
import Vue from 'vue'
import App from './App'
import "bootstrap/dist/css/bootstrap.min.css";
import Colorize from "./directives/colorize";
Vue.directive("colorize", Colorize);
Vue.config.productionTip = false
new Vue({
 el: '#app',
 components: { App },
 template: '<App/>'
})
...
```

Pierwszy argument to nazwa, za pomocą której dyrektywa będzie używana. Drugi argument to obiekt lub funkcja zainportowana z pliku JavaScript, zawierającego własną dyrektywę. Metoda *Vue.directive* musi być wywołana przed utworzeniem nowego obiektu Vue w aplikacji, jak widać w załączonym fragmencie kodu.

W rezultacie możesz korzystać z dyrektywy w całej aplikacji bez konieczności użycia właściwości *directives*.

---

Drugi argument przekazany do funkcji haka to obiekt, który reprezentuje wiązanie dyrektywy z elementem HTML i zawiera właściwości opisane w tabeli 26.4.

**Tabela 26.4.** Właściwości zdefiniowane w obiekcie wiązania

Nazwa	Opis
name	Ta właściwość zwraca nazwę, której można użyć w elemencie HTML, aby zastosować dyrektywę bez prefiksu <i>v-</i> . W przypadku listingu 26.8 ta właściwość przyjęłaby wartość <i>colorize</i> .
expression	Ta właściwość zwraca wyrażenie zastosowane w dyrektywie w formie łańcucha znaków. W omawianym przykładzie byłaby to wartość <i>total</i> . Nie musisz przetwarzać wyrażenia, aby uzyskać wynik — wystarczy skorzystać z właściwości <i>value</i> .
value	Ta właściwość zwraca aktualną wartość wyrażenia dla tej dyrektywy. W omawianym przykładzie byłaby to bieżąca wartość właściwości <i>total</i> komponentu.
oldValue	Ta właściwość zwraca poprzednią wartość wyrażenia, ale jest ona dostępna tylko w funkcjach haków <i>update</i> i <i>componentUpdated</i> .
arg	Ta właściwość zwraca argument zastosowany wobec dyrektywy, jeżeli taki istnieje.
modifiers	Ta właściwość zwraca modyfikatory zastosowane wobec dyrektywy, jeżeli takie istnieją.

- **Wskazówka** Funkcje haka są dostarczane z obiektami VNode, które Vue.js wykorzystuje wewnętrznie do śledzenia elementów HTML. Nie omawiam ich jednak w tym rozdziale, ponieważ nie są one dla nas przydatne. Więcej szczegółów znajdziesz na stronie <https://vuejs.org/v2/api/#VNode-Interface>.

Dyrektywa własna, którą wprowadzam w listingu 26.7, korzysta z wartości właściwości value, aby pobrać wartość bieżącą wyrażenia. Na jej podstawie podejmowana jest decyzja, czy element zostanie dodany do klas Bootstrapa, czy usunięty z nich.

```
...
export default {
 update(el, binding) {
 if (binding.value > 100) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 }
}
...

```

Zwróć uwagę, że dyrektywa nie ma bezpośredniego związku z komponentem — wartość jest przekazywana za pomocą wyrażenia, bez jakiegokolwiek informacji o jej znaczeniu.

- **Ostrzeżenie** Właściwości opisane w tabeli 26.4 są tylko do odczytu. Własne dyrektywy powinny wprowadzać zmiany jedynie za pośrednictwem elementu HTML.

## **Stosowanie wyrażeń własnych dyrektyw**

Pewną pokusą podczas tworzenia własnych dyrektyw jest umieszczenie większej ilości kodu logiki w ramach dyrektywy zamiast zastosowania podstawowych funkcji Vue.js. Utworzona dyrektywa w rezultacie nie może być stosowana zbyt powszechnie. Dyrektywa, którą zdefiniowałem w listingu 26.7, niewątpliwie ma ten sam problem, ponieważ umieściłem w kodzie „na sztywno” wartość, która wyzwała kolorowanie elementu. Lepszym podejściem jest skorzystanie z mechanizmu wyrażeń Vue.js, aby umożliwić komponentowi kontrolę zachowania dyrektywy (listing 26.9).

*Listing 26.9. Zastosowanie wyrażenia w pliku src/components/Numbers.vue*

```
...
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input v-colorize="first > 45" class="form-control"
 v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div v-colorize="total > 50" class="col h3">= {{ total }} </div>
 </div>
 </div>
 </div>
```

```
</div>
</div>
</template>
...

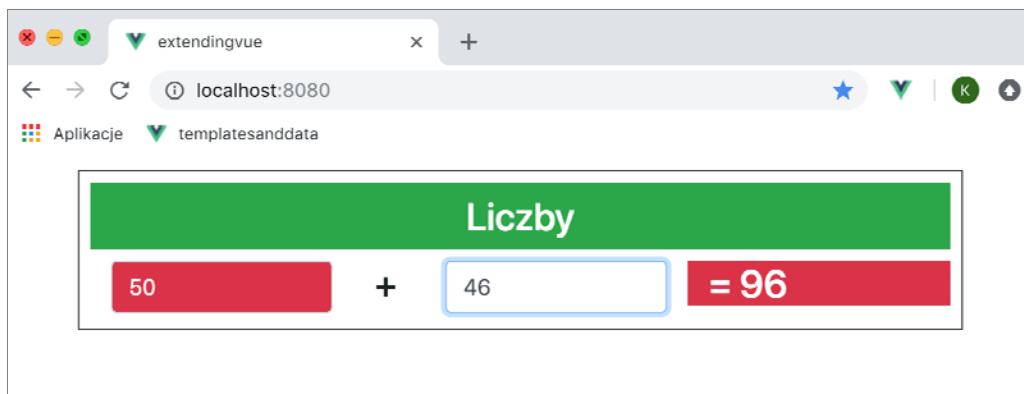
```

Zamiast z możliwości dostarczenia wartości total do dyrektywy skorzystałem z wyrażeń, które pozwalają na wyzwolenie klas. Oznacza to, że mogę zastosować tę samą dyrektywę do różnych elementów z różnymi wartościami wyzwolenia. W listingu 26.10 wprowadziłem analogiczną zmianę do dyrektywy.

**Listing 26.10.** Usuwanie wartości wyzwolenia w pliku src/directives/colorize.js

```
export default {
 update(el, binding) {
 if (binding.value) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 }
}
```

W efekcie dyrektywa zmieni kolor tła i czcionki pierwszego elementu input, jeśli wartość przekroczy 45, a dla elementu div — jeśli wartość total przekroczy 50 (rysunek 26.3).



**Rysunek 26.3.** Zastosowanie tej samej dyrektywy do wielu elementów

## Stosowanie argumentów własnej dyrektywy

Dyrektyna może otrzymać argumenty, które dostarczą dodatkowych informacji związanych z jej zachowaniem — np. na temat sposobu obsługi zdarzenia za pomocą dyrektywy v-on (więcej informacji znajdziesz w rozdziale 14.). Własne dyrektywy również mogą otrzymywać argumenty. W listingu 26.11 zastosowałem argument w celu określenia nazwy klasy, która będzie zastosowana do zmiany koloru tła elementu zawierającego daną dyrektywę.

**Listing 26.11.** Otrzymywanie argumentu w pliku src/directives/colorize.js

```
export default {
 update(el, binding) {
 const bgClass = binding.arg || "bg-danger";
 if (binding.value) {
 el.classList.add(bgClass, "text-white");
```

```

 } else {
 el.classList.remove(bgClass, "text-white");
 }
 }
}

```

Korzystam z właściwości arg, aby pobrać nazwę klasy, i odwołuję się do klasy bg-danger, jeśli argument nie zostanie dostarczony. W listingu 26.12 dodałem argument do jednej z dyrektyw, określając klasę bg-info.

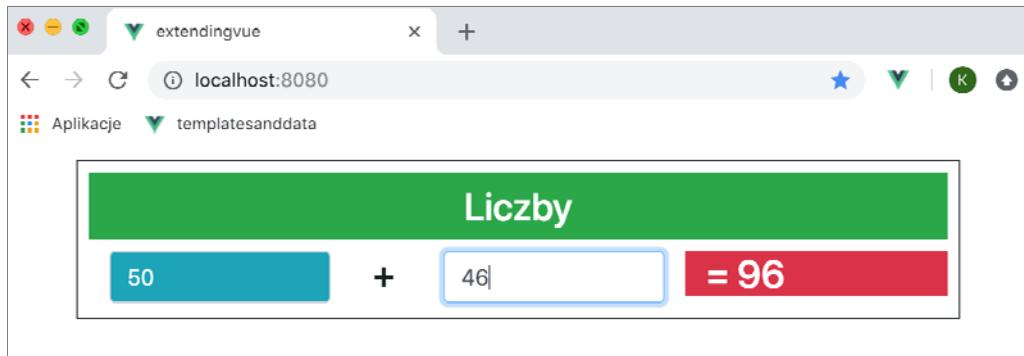
*Listing 26.12. Dodawanie argumentu do pliku src/components/Numbers.vue*

```

...
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input v-colorize:bg-info="first > 45" class="form-control"
 v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div v-colorize="total > 50" class="col h3" style="background-color: #f0f0f0;">= {{ total }}

```

Teraz, gdy wprowadzisz wartość większą niż 45 do pierwszego pola tekstowego, do Twojego elementu zostanie przypisana klasa Bootstrapa określająca inny kolor tła (rysunek 26.4).



*Rysunek 26.4. Zastosowanie argumentu we własnej dyrektywie*

## **Stosowanie modyfikatorów własnej dyrektywy**

Modyfikatory udostępniają dyrektywy dzięki dodatkowym instrukcjom, za pomocą których można rozwinąć możliwości argumentów. W listingu 26.13 aktualizuję własną dyrektywę, dzięki czemu sprawdza ona modyfikatory określające, czy kolory tła i tekstu powinny być zmienione.

**Listing 26.13.** Zastosowanie modyfikatorów w pliku src/directives/colorize.js

```
export default {
 update(el, binding) {
 const bgClass = binding.arg || "bg-danger";
 const noMods = Object.keys(binding.modifiers).length == 0;
 if (binding.value) {
 if (noMods || binding.modifiers.bg) {
 el.classList.add(bgClass);
 }
 if (noMods || binding.modifiers.text) {
 el.classList.add("text-white");
 }
 } else {
 el.classList.remove(bgClass, "text-white");
 }
 }
}
```

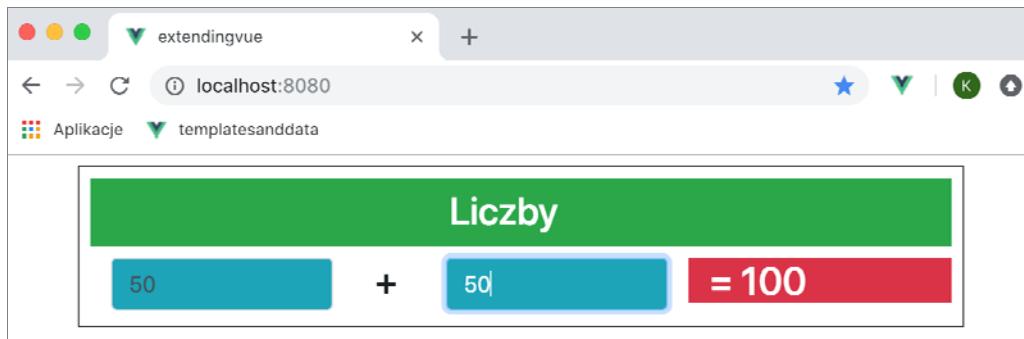
Modyfikatory są dostępne za pomocą obiektu zwróconego przy użyciu właściwości `modifiers` parametru `binding`. Jeśli nie zastosowano żadnego modyfikatora, obiekt nie będzie miał żadnych właściwości. Dla każdego z zastosowanych modyfikatorów zostanie utworzona właściwość, której nazwa jest nazwą modyfikatora, a wartością jest `true`. W przypadku mojej dyrektywy zmieniam kolor tła i tekstu, ale tylko jeśli nie ma żadnych modyfikatorów. Jeżeli zostanie zastosowany modyfikator `bg`, będzie oznaczać to, że ma zostać zmieniony kolor tła — analogicznie w przypadku modyfikatora `text` i koloru tekstu. W listingu 26.14 korzystam z różnych kombinacji modyfikatorów dyrektywy, a także stosuję dyrektywę wobec drugiego elementu `input`.

**Listing 26.14.** Zastosowanie modyfikatorów dyrektywy w pliku src/components/Numbers.vue

```
...
<template>
 <div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input v-colorize:bg-info.bg="first" > 45" class="form-control"
 v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input v-colorize:bg-info="second" > 30"
 class="form-control" v-model.number="second" />
 </div>
 <div v-colorize.bg.text="total > 50" class="col h3">
 = {{ total }}
 </div>
 </div>
 </div>
 </template>
 ...

```

Modyfikatory przykładowej dyrektywy są opcjonalne, dlatego mogę korzystać z atrybutu `v-colorize` bez żadnych modyfikatorów, tylko z modyfikatorem `bg`, a także z obydwoma modyfikatorami `bg` i `text`. W ten sposób mogę konfigurować osobno każdy z elementów HTML, który otrzymał dyrektywę (rysunek 26.5).



Rysunek 26.5. Zastosowanie modyfikatorów w celu skonfigurowania dyrektywy

## Komunikacja między funkcjami haków

Własne dyrektywy z założenia są proste i bezstanowe. Jeśli chcesz przekazywać dane między funkcjami haków, np. chcesz wykorzystać w haku update treść wygenerowaną w haku bind (lub skorzystać z wyniku wywołania haka update podczas kolejnej aktualizacji), musisz liczyć się z dodatkową pracą. Rozwiążaniem jest zastosowanie elementu HTML do przechowywania danych za pomocą atrybutów data. W listingu 26.15 modyfikuję własną dyrektywę, aby skorzystała z atrybutu data w celu śledzenia, czy element został dodany do klas Bootstrapa.

*Listing 26.15. Zastosowanie atrybutów danych w pliku src/directives/colorize.js*

```
export default {
 update(el, binding) {
 const bgClass = binding.arg || "bg-danger";
 const noMods = Object.keys(binding.modifiers).length == 0;
 if (binding.value) {
 if (noMods || binding.modifiers.bg) {
 el.classList.add(bgClass);
 el.dataset["bgClass"] = true;
 }
 if (noMods || binding.modifiers.text) {
 el.classList.add("text-white");
 el.dataset["textClass"] = true;
 }
 } else {
 if (el.dataset["bgClass"]) {
 el.classList.remove(bgClass);
 el.dataset["bgClass"] = false;
 }
 if (el.dataset["textClass"]) {
 el.classList.remove("text-white");
 el.dataset["textClass"] = false;
 }
 }
 }
}
```

Właściwość dataset daje dostęp do atrybutów data elementu HTML. Tworzę atrybuty data-bgClass i data-textClass w celu wskazania, czy element został dodany do klas Bootstrapa. W tym przykładzie nie ma żadnej widocznej zmiany, ale jeśli ustawisz wartość pierwszego elementu HTML na większą niż 45, a następnie przeanalizujesz element za pomocą narzędzi F12, zobaczyś, że dyrektywa skorzystała z elementu w celu przechowania danych stanu:

```
...
<div class="col h3 bg-danger text-white"
 data-bg-class="true" data-text-class="true">
 = 70
</div>
...
```

To zachowanie może wydawać się dziwne, ale w ten sposób element HTML udostępnia spójne źródło danych wykorzystywane przez dyrektywę, bez potrzeby dodawania funkcji takich jak obsługa danych lokalnych czy odrębnego cyklu życia dyrektywy.

## Dyrektyny jednofunkcyjne

Dyrektyny są podatne na duplikację kodu, jeśli muszą wykonywać te same zadania najpierw w trakcie konfiguracji, a później w trakcie zmiany (listing 26.16).

*Listing 26.16. Dodawanie haka do pliku src/directives/colorize.js*

```
export default {
 bind(el, binding) {
 if (binding.value) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 },
 update(el, binding) {
 if (binding.value) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
 }
}
```

Uprościłem dyrektywę, dzięki czemu nie korzysta ona z argumentów, modyfikatorów ani atrybutów danych, a także dodałem hak `bind`. Dzięki temu początkowa wartość wyrażenia zastosowanego do wdrożenia dyrektywy jest używana do skonfigurowania elementu HTML. Niestety duplikujemy instrukcje w każdymaku.

Taka sytuacja pojawia się niezwykle często, dlatego Vue.js wspiera optymalizację, która pozwala na wyrażenie dyrektyw, wymagających jedynie haków `bind` i `update`, w formie jednej funkcji (listing 26.17).

*Listing 26.17. Zastosowanie pojedynczej funkcji w pliku src/directives/colorize.js*

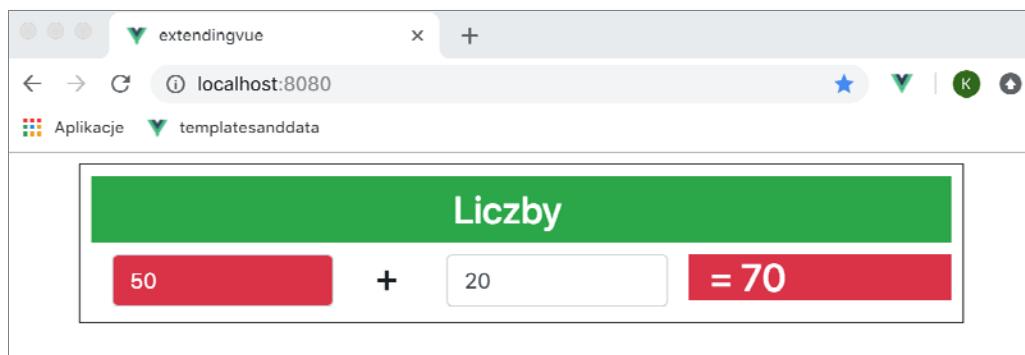
```
export default function (el, binding) {
 if (binding.value) {
 el.classList.add("bg-danger", "text-white");
 } else {
 el.classList.remove("bg-danger", "text-white");
 }
}
```

Wadą takiego podejścia jest brak możliwości określenia innych haków, ale z drugiej strony większość dyrektyw może być w ten sposób wyrażona w formie jednej funkcji bez duplikowania kodu. Aby pokazać, że Vue.js wdraża dyrektywę, tak jakby zawierała hak `bind`, zwiększałem wartość początkową jednej z właściwości danych komponentu `Numbers` (listing 26.18).

**Listing 26.18.** Zwiększenie właściwości danych w pliku *src/components/Numbers.vue*

```
...
<script>
import Colorize from "../directives/colorize";
export default {
 data: function() {
 return {
 first: 50,
 second: 20
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 },
 directives: {
 Colorize
 }
}
</script>
...
```

Nowa wartość przekracza próg zastosowany w dyrektywie, co daje wynik jak na rysunku 26.6.

**Rysunek 26.6.** Zastosowanie pojedynczej funkcji w celu dostarczenia haka bind

## Tworzenie domieszek komponentów

Domieszki stanowią użyteczny sposób zapewniania komponentom współdzielonych mechanizmów w celu ograniczenia duplikacji kodu. Zaletą domieszk jest ich prostota, a wadą — brak możliwości zastosowania do współdzielenia stanu, przez co trzeba skorzystać z funkcji takich jak wstrzykiwanie zależności czy magazyn danych.

Lubię definiować domieszki w odrębnym katalogu, aby odseparować je od reszty projektu. W ramach przykładu tworzę katalogu *src/mixins* i dodaję plik *numbersMixin.js* o treści z listingu 26.19.

**Listing 26.19.** Zawartość pliku *src/mixins/numbersMixin.js*

```
import Colorize from "../directives/colorize";
export default {
 data: function() {
 return {
 first: 50,
 second: 20
 }
 },
 computed: {
 total() {
 return this.first + this.second;
 }
 }
}
```

```

 first: 50,
 second: 20
 }
},
computed: {
 total() {
 return 0;
 }
},
directives: {
 Colorize
},
}
}

```

Domieszka może zawierać dowolny kod typowy dla komponentu, włączając w to właściwości danych i obliczane metody, filtry i dyrektywy. Jeśli tworzysz zbiór powiązanych komponentów, które współdzielą typowe funkcje, domieszka stanowi dobrą metodę zapobiegania kopiowaniu i wklejaniu tego samego kodu do elementu script plików *.vue*. Domieszka z listingu 26.19 zawiera właściwości danych, właściwość obliczaną, a także rejestrację dyrektywy — wszystko to pochodzi z komponentu *Numbers*, choć właściwość obliczana total zwraca 0. Skonstruowałem przykład w ten sposób, aby pokazać zasadę działania domieszek.

W listingu 26.20 zmieniam komponent *Numbers*, aby zawierał jedynie domieszkę i funkcje, które go wyróżniają.

**Listing 26.20.** Zastosowanie domieszki w pliku *src/components/Numbers.vue*

```

...
<script>
 import mixin from "../mixins/numbersMixin";
 export default {
 computed: {
 total() {
 return this.first + this.second;
 }
 },
 mixins: [mixin]
 }
</script>
...

```

Domieszki stosuje się za pomocą właściwości *mixins*, która otrzymuje tablicę obiektów domieszek. W momencie użycia domieszki Vue.js traktuje komponent tak, jakby zawierał on dane i funkcje udostępniane przez domieszkę. Jeżeli komponent zawiera składnik o takiej samej nazwie — tak jak właściwość obliczana *total* w listingu 26.20 — to składnik komponentu przesyłania ten udostępniany przez domieszkę.

## Stosowanie domieszki we wszystkich komponentach

Domieszki są rejestrowane globalnie, co pozwala udostępnić wybraną funkcję we wszystkich komponentach aplikacji. Nie jest to mechanizm, który powinno się stosować często, ponieważ konsekwencje są niezwykle poważne i nierzadko powodują nieoczekiwane efekty. Domieszki są rejestrowane globalnie w pliku *main.js* za pomocą metody *Vue.mixin*:

```

...
import Vue from 'vue'
import App from './App'
import "bootstrap/dist/css/bootstrap.min.css";
Vue.config.productionTip = false

```

```
import mixin from "./mixins/numbersMixin";
Vue.mixin(mixin);
new Vue({
 el: '#app',
 components: { App },
 template: '<App>'
})
...

```

Metoda `Vue.mixin` musi być wywołana przed utworzeniem nowego obiektu Vue, tak jak w przykładowym fragmencie. Po zarejestrowaniu domieszki w taki sposób nie musisz korzystać z właściwości `mixins` bezpośrednio w komponentach.

W ten sposób domieszka pozwala na uzyskanie ogólnych, typowych funkcji, które później mogą być przesyłane w komponentach. W listingu 26.21 pokazuję, jak pojedyncza domieszka może być zastosowana w celu stworzenia podstaw dla powiązanych komponentów (w pliku `src/components/Subtraction.vue`).

**Listing 26.21.** Zawartość pliku `src/components/Subtraction.vue`

```
<template>
<div class="mx-5 p-2 border border-dark">
 <h3 class="bg-info text-white text-center p-2">Odejmowanie</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">-</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div v-colorize.bg.text="total > 50" class="col h3">= {{ total }}</div>
 </div>
 </div>
</div>
</template>
<script>
import mixin from "../mixins/numbersMixin";
export default {
 computed: {
 total() {
 return this.first - this.second;
 }
 },
 mixins: [mixin]
}
</script>
```

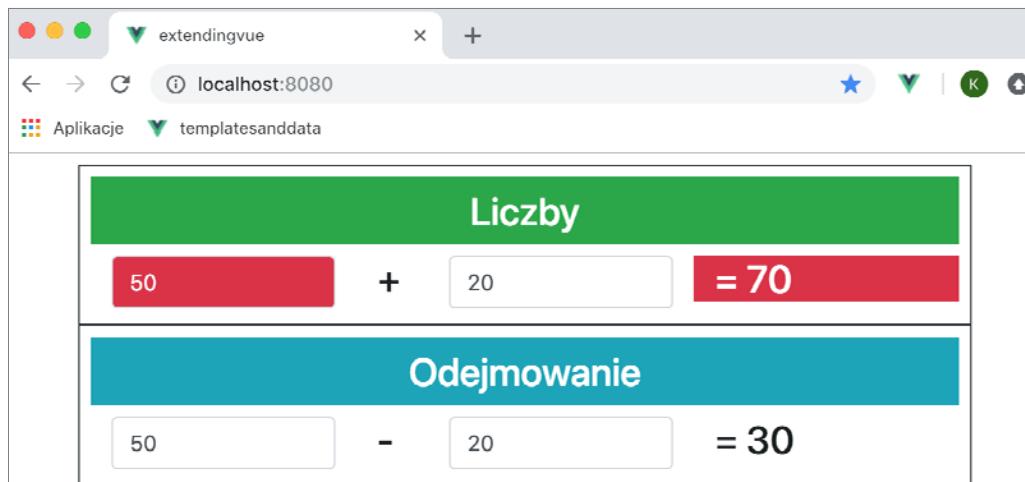
- **Wskazówka** Jeżeli daną metodę cyklu życia implementuje zarówno domieszka, jak i komponent (por. rozdział 17.), Vue.js najpierw wywoła metodę domieszki, a potem — komponentu.

Ten komponent zawiera taką samą strukturę jak komponent `Numbers`, ale jednocześnie przesyłania właściwość obliczaną `total`, dzięki czemu dochodzi do odejmowania jednej właściwości danych od drugiej. Wszystkie inne mechanizmy komponentu są dostarczane przez domieszkę. W listingu 26.22 modyfikuję komponent `App`, aby wyświetlić nowo dodany komponent.

**Listing 26.22.** Dodawanie komponentu w pliku src/App.vue

```
<template>
 <div class="m-2">
 <numbers />
 <subtraction />
 </div>
</template>
<script>
import Numbers from "./components/Numbers";
import Subtraction from "./components/Subtraction";
export default {
 name: 'App',
 components: { Numbers, Subtraction }
}
</script>
```

Wyświetlam nowy komponent obok istniejącego, co daje efekt jak na rysunku 26.7.

**Rysunek 26.7.** Tworzenie podobnych komponentów za pomocą domieszki

- **Uwaga** Każdy komponent, który korzysta z domieszki, otrzymuje własne właściwości danych, które nie są współdzielone z innymi komponentami. Jeśli chcesz operować na tych samych wartościach danych, zajrzyj do rozdziału 18. i zapoznaj się z wstrzykiwaniem zależności lub do rozdziału 20. w celu zapoznania się z magazynami danych.

## Tworzenie wtyczki Vue.js

Wtyczki pozwalają na współdzielenie szerokiego zbioru mechanizmów globalnie, w całej aplikacji, bez potrzeby konfiguracji każdej funkcji z osobna. Do tych funkcji należą dyrektywy i domieszki, ale istnieje możliwość globalnego definiowania metod i właściwości, co pozwala na funkcjonowanie pakietów takich jak Vuex (rozdział 20.) czy Vue Router (rozdział 22.).

Aby pokazać, jak działają wtyczki, utworzę zbiór globalnych funkcji, które obsługują najprostsze operacje arytmetyczne, w przykładowej aplikacji.

Rozpocznę od utworzenia katalogu `src/plugins/math` i dodam do niego plik `filters.js` o treści z listingu 26.23.

**Listing 26.23.** Zawartość *src/plugins/maths/filters.js*

```
export default {
 currency: function(value) {
 return new Intl.NumberFormat("pl-PL", {
 style: "currency",
 currency: "PLN"
 }).format(value);
 },
 noDecimal: function(value) {
 return Number(value).toFixed(0);
 }
}
```

Ten plik zawiera dwie funkcje filtru, które formatują wartości liczbowe, a następnie przypisuje je do właściwości. Niebawem skorzystam z nazw tych właściwości, aby zarejestrować filtry. Cel tych filtrów nie jest dla nas tak ważny jak możliwość dołączenia ich do wtyczki. Więcej informacji na temat działania filtrów znajdziesz w rozdziale 11.

Wtyczki mogą zawierać również dyrektywy, dlatego do katalogu *src/plugins/maths* dodaję plik *directives.js* o treści z listingu 26.24.

**Listing 26.24.** Zawartość pliku *src/plugins/maths/directives.js*

```
export default {
 borderize: function(el, binding) {
 if (binding.value) {
 el.classList.add("border", "border-dark");
 } else {
 el.classList.remove("border", "border-dark");
 }
 }
}
```

Ten plik zawiera dyrektywę jednofunkcyjną, która stosuje obramowanie do swojego elementu HTML, gdy wyrażenie przyjmuje wartość true. Nie jest to zbyt użyteczna dyrektywa, ale własne dyrektywy rzadko są użyteczne i nie są potrzebne w większości aplikacji.

Wtyczki mogą zawierać także globalne metody i właściwości, do których dostęp jest możliwy z całej aplikacji. Do katalogu *src/plugins/maths* dodaję plik *globals.js* o treści z listingu 26.25.

**Listing 26.25.** Zawartość pliku *src/plugins/maths/globals.js*

```
export default {
 sumValues(...vals) {
 return vals.reduce((val, total) => total += val, 0);
 },
 getSymbol(operation) {
 switch (operation.toLowerCase()) {
 case "add":
 return "+";
 case "subtract":
 return "-";
 case "multiply":
 return "*";
 default:
 return "/";
 }
 }
}
```

W listingu definiuję funkcję `sumValues`, która korzysta z argumentu `reszty`, aby otrzymać tablicę wartości poddaną sumowaniu w celu uzyskania wyniku, oraz metodę `getSymbol`, która przyjmuje nazwę operacji matematycznej i zwraca właściwy symbol.

Z wtyczki możesz skorzystać także, aby dodać właściwości i metody do wszystkich komponentów, na podobnej zasadzie jak Vuex, który dodaje właściwość `$store`, czy Vue Router, odpowiedzialny za właściwości `$route` i `$router`. Do katalogu `src/plugins/math`s dodaję plik `componentFeatures.js` o treści z listingu 26.26.

**Listing 26.26.** Zawartość pliku `src/plugins/math/componentFeatures.js`

```
export default {
 $calc: {
 add(first, second) {
 return first + second;
 },
 subtract(first, second) {
 return first - second;
 },
 multiply(first, second) {
 return first * second;
 },
 divide(first, second) {
 return first / second;
 }
 }
}
```

Zgodnie z konwencją funkcje dostarczone do komponentów mają nazwy zaczynające się od znaku dolara. W listingu definiuję obiekt `$calc` zawierający metody `add`, `subtract`, `multiply` i `divide`, które wykonują podstawowe operacje matematyczne.

Wtyczki mogą zawierać komponenty, co stanowi przydatną metodę zapewniania mechanizmów dostępnych w całej aplikacji. Do katalogu `src/plugins/math`s dodaję plik `Operation.vue` o treści z listingu 26.27.

**Listing 26.27.** Zawartość pliku `src/plugins/math/Operation.vue`

```
<template>
<div class="mx-5 p-2 border border-dark">
 <h3 class="bg-info text-white text-center p-2">{{ operation }}</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input class="form-control" v-model.number="first" />
 </div>
 <div class="col-1 h3">{{ symbol }}</div>
 <div class="col">
 <input class="form-control" v-model.number="second" />
 </div>
 <div class="col h3" v-borderize="total > 25">= {{ total }}</div>
 </div>
 </div>
</template>
<script>

import Vue from "vue";
export default {
 props: ["firstVal", "secondVal", "operation"],
 data: function() {
```

```

 return {
 first: Number(this.firstVal),
 second: Number(this.secondVal)
 }
 },
 computed: {
 symbol() {
 return Vue.getSymbol(this.operation);
 },
 total() {
 switch (this.operation.toLowerCase()) {
 case "add":
 return this.$calc.add(this.first, this.second);
 case "subtract":
 return this.$calc.subtract(this.first, this.second);
 case "multiply":
 return this.$calc.multiply(this.first, this.second);
 case "divide":
 return this.$calc.divide(this.first, this.second);
 }
 }
 }
}
</script>

```

Ten komponent przedstawia ustandaryzowany interfejs do wykonywania prostych operacji na dwóch liczbach, korzystając z innych funkcji wtyczki, które omówię niebawem.

## Tworzenie wtyczki

Funkcje zdefiniowane przed chwilą muszą zostać połączone, aby utworzyć wtyczkę. Do katalogu `src/plugins/math` dodaję plik `index.js` o zawartości z listingu 26.28, łączący różne mechanizmy w celu utworzenia wtyczki.

**Listing 26.28.** Zawartość pliku `src/plugins/math/index.js`

```

import filters from "./filters";
import directives from "./directives";
import globals from "./globals";
import componentFeatures from "./componentFeatures";
import Operation from "./Operation";
export default {
 install: function(Vue) {
 Vue.filter("currency", filters.currency);
 Vue.filter("noDecimal", filters.noDecimal);
 Vue.directive("borderize", directives.borderize);
 Vue.component("maths", Operation);
 Vue.sumValues = globals.sumValues;
 Vue.getSymbol = globals.getSymbol;
 Vue.prototype.$calc = componentFeatures.$calc;
 }
}

```

Wtyczki to obiekty, które definiują funkcję `install`. Funkcja ta otrzymuje obiekt `Vue` i opcjonalny obiekt konfiguracji. Metody dostarczone przez obiekt `Vue` są używane do zarejestrowania poszczególnych funkcji, które importuję z plików JavaScript utworzonych w poprzednich podrozdziałach (por. tabela 26.5).

**Tabela 26.5.** Metody Vue odpowiedzialne za rejestrowanie wtyczek

Nazwa	Opis
Vue.directive	Ta metoda służy do rejestrowania dyrektywy. Argumentami są nazwa, za pomocą której dyrektywa zostanie wdrożona, i obiekt dyrektywy.
Vue.filter	Ta metoda służy do rejestrowania filtru. Argumentami są nazwa, za pomocą której filtr zostanie wdrożony, i obiekt filtru.
Vue.component	Ta metoda służy do rejestrowania komponentu. Argumentami są nazwa, za pomocą której komponent zostanie wdrożony, i obiekt komponentu.
Vue.mixin	Ta metoda jest używana do rejestrowania domieszki. Argumentem jest obiekt domieszki.

W listingu 26.28 skorzystałem z metod `filter`, `directive` i `component`, aby zarejestrować uprzednio zdefiniowane funkcje. Aby zarejestrować metody i właściwości globalne, należy dodać je do obiektu `Vue`:

```
...
Vue.getSymbol = globals.getSymbol;
...
```

Ta instrukcja udostępnia metodę `getSymbol` w całej aplikacji, co oznacza, że jest ona dostępna za pomocą obiektu `Vue`, jak pokazuje przykład z komponentu z listingu 26.27:

```
...
symbol() {
 return Vue.getSymbol(this.operation);
},
...
```

Metody i właściwości, z których chcesz skorzystać we wszystkich komponentach, muszą być dodane do obiektu `Vue.prototype`:

```
...
Vue.prototype.$calc = componentFeatures.$calc;
...
```

Ta instrukcja ustawia obiekt `$calc`, dzięki czemu można z niego skorzystać za pomocą konstrukcji `this.$calc`, co pokazuje przykład z listingu 26.27:

```
...
return this.$calc.add(this.first, this.second);
...
```

Dzięki połączeniu tych mechanizmów, jak również metod z tabeli 26.5 wtyczka może dostarczyć aplikacji Vue.js duży zakres funkcji.

## Stosowanie wtyczki

Wtyczki włączają się za pomocą metody `Vue.use`. Tą samą metodę stosowano do zarejestrowania wtyczek magazynu danych i trasowania URL we wcześniejszych rozdziałach. Jedyna różnica polega na tym, że tamte wtyczki były zawarte we własnych pakietach NPM. W listingu 26.29 importuję własny pakiet w przykładowej aplikacji i włączam go za pomocą metody `Vue.use`.

**Listing 26.29.** Włączanie wtyczki w pliku `src/main.js`

```
import Vue from 'vue'
import App from './App.vue'
import "bootstrap/dist/css/bootstrap.min.css";
```

```
Vue.config.productionTip = false
import MathsPlugin from "./plugins/math";
Vue.use(MathsPlugin);
new Vue({
 render: h => h(App)
}).$mount('#app')
```

Po włączeniu wtyczki jej funkcje są dostępne w całej aplikacji. W listingu 26.30 dodaję utworzony komponent do nadziednego szablonu komponentu App.

**Listing 26.30.** Zastosowanie komponentu z wtyczki w pliku src/App.vue

```
<template>
<div class="m-2">
 <numbers />
 <subtraction />
 <maths operation="Divide" firstVal="10" secondVal="20" />
</div>
</template>
<script>
import Numbers from "./components/Numbers";
import Subtraction from "./components/Subtraction";
export default {
 name: 'App',
 components: {
 Numbers,
 Subtraction
 }
}
</script>
```

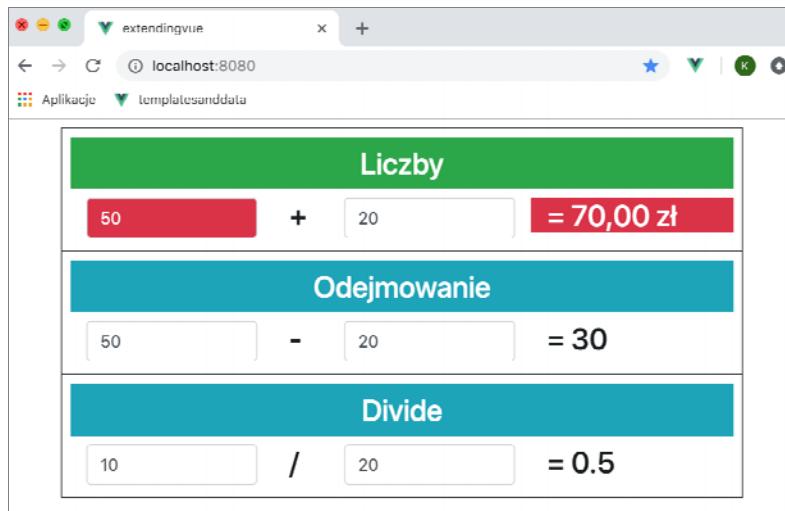
Zwróć uwagę, że nie muszę rejestrować komponentu i mogę po prostu dodać element maths do szablonu, ponieważ funkcje zdefiniowane we wtyczce są dostępne w całej aplikacji. W ramach przykładu w listingu 26.31 korzystam z jednego z filtrów w globalnej metody w komponencie Numbers.

**Listing 26.31.** Zastosowanie możliwości wtyczki w pliku src/components/Numbers.vue

```
<template>
<div class="mx-5 p-2 border border-dark">
 <h3 class="bg-success text-white text-center p-2">Liczby</h3>
 <div class="container-fluid">
 <div class="row">
 <div class="col">
 <input v-colorize:bg-info.bg="first > 45" class="form-control"
 v-model.number="first" />
 </div>
 <div class="col-1 h3">+</div>
 <div class="col">
 <input v-colorize:bg-info="second > 30"
 class="form-control" v-model.number="second" />
 </div>
 <div v-colorize:bg.text="total > 50" class="col h3">
 = {{ total | currency }}
 </div>
 </div>
 </div>
</template>
<script>
```

```
import mixin from "../mixins/numbersMixin";
import Vue from "vue";
export default {
 computed: {
 total() {
 return Vue.sumValues(this.first, this.second);
 }
 },
 mixins: [mixin]
}
</script>
```

Efektem tego listingu jest wyświetlenie wartości sformatowanej jako kwota pieniędzy, a także nowy komponent wyświetlony użytkownikowi (rysunek 26.8)<sup>1</sup>.



Rysunek 26.8. Zastosowanie funkcji wtyczki w praktyce

## Podsumowanie

W tym rozdziale opisałem różne sposoby zwiększenia możliwości Vue.js. Pokazałem, jak utworzyć własną dyrektywę, zaznaczając przy tym, że rzadko jest to najlepsze rozwiązanie. Opisałem także, jak rozszerzać możliwości komponentów za pomocą domieszek, a także jak udostępniać zbiór powiązanych ze sobą funkcji przy użyciu wtyczek.

Na tym chciałbym zakończyć swoją opowieść o Vue.js. Rozpoczęliśmy naszą przygodę od utworzenia prostej aplikacji, a następnie zapoznałem Cię z rozmaitymi zagadnieniami związanymi z różnymi elementami wchodzącymi w skład aplikacji tworzonych w Vue.js. Pokazałem, jak tworzyć, konfigurować i wdrażać tego rodzaju aplikacje.

Życzę Ci wielu sukcesów w tworzeniu własnych projektów w Vue.js. Mam nadzieję, że czytając tę książkę, miałeś tyle radości, ile ja w trakcie jej pisania.

<sup>1</sup> Użycie na rysunku angielskiego słowa „Divide” wynika z zastosowania w kodzie literałów tekstowych w tym języku — przyp. tłum.



# Skorowidz

## A

administrowanie, 149, 173  
adres URL, 122  
akcje magazynu danych, 467  
aktualizacja, 344, 395  
    treści, 297  
alias, 289  
    trasy, 518  
analiza  
    dokumentu HTML, 56  
    stanu aplikacji, 231  
    zmian w magazynie danych, 460  
Angular, 45  
animacje, 598, 601  
    CSS, 595  
API, Application Programming Interface, 46  
    historii HTML5, 515  
    modelu DOM, 195  
aplikacja  
    Sklep sportowy, 95  
aplikacje  
    debugowanie, 231  
    planowanie układu, 547  
    Vue.js, 209  
    wielostroniczne, 44  
argument własnej dyrektywy, 621  
argumenty, 256  
asynchroniczne  
    komponenty, 494  
    wykonywanie operacji, 91

atak typu XSS, 435  
atrybuty, 55  
    bez wartości, 56  
    elementu router-link, 539  
automatyczna nawigacja, 490

## B

białe znaki, 345  
biblioteka  
    Axios, 437  
    do obsługi animacji, 598, 601  
błedy, 224, 444  
    komponentów, 400  
    ładowania, 581  
Bootstrap  
    CSS, 194  
    marginesy, 59  
    odstępy, 59  
    stylowanie formularzy, 62  
    stylowanie tabel, 60  
    tworzenie siatki, 60

## C

catchall, 516  
ceny produktów, 106  
checkbox, 33  
CORS, Cross-Origin Resource Sharing, 435  
CSS, Cascading Style Sheets, 28, 51, 58, 272  
cykl życia komponentu, 385, 489

**D**

debugger, 231  
 debugowanie aplikacji, 231  
 definiowanie  
     reguł walidacji, 353  
     strażników tras, 570  
     tras, 548  
 Docker, 188  
 dodawanie  
     argumentu, 622  
     argumentu filtru, 256  
     Bootstrapa, 28  
     elementu do tablicy, 155  
     filtru, 255  
     komponentów, 547  
     komponentów administracyjnych, 163  
 listy produktów, 106  
 nawigacji, 492, 493  
 operacji HTTP, 441  
 pakietów, 96  
 pakietu Bootstrap CSS, 240  
 propa, 368  
 przycisku wyboru, 33  
 strażnika trasy, 166  
 stylów CSS, 272  
 treści dynamicznych, 29  
 wiązania dwukierunkowego, 334  
 dołączanie stylów CSS, 97  
 DOM, Document Object Model, 46, 195, 281  
 domieszki, 626, 627  
 domknięcia, 74  
 dopasowanie adresów URL, 525, 544  
 dostęp  
     do gettera, 463  
     do komponentu, 572  
     do konfiguracji trasowania, 507  
     do magazynu danych, 459  
     do modelu DOM, 390  
     do stanu modułu, 476  
 duplikacja zdarzeń, 326  
 dynamiczne dopasowywanie tras, 522  
 dynamiczne komponenty, 481  
 dynamiczne wyświetlanie komponentów, 485  
 dyrektywa, 201, 261  
     Repeater, 285  
     v-bind, 275, 279, 281  
     v-else, 270  
     v-for, 285, 293, 301–303

    v-if, 272  
     v-model, 337, 343, 348  
     v-on, 311, 374  
     v-text, 264  
     v-for  
 dyrektywy  
     globalne rejestrowanie, 619  
     jednofunkcyjne, 625  
     modyfikatory, 622  
     skrócone, 275, 315  
     tworzenie, 616  
 działanie  
     aplikacji wielostronowych, 44  
     Bootstrapa, 58  
     obiektu Vue, 193, 202  
     obietnic, 90

**E**

edytor, 23  
 produktów, 178  
 elastyczne formatowanie znaczników, 364  
 element, 54  
     router-link, 537, 539, 542  
     router-view, 124, 553, 556  
     script, 243  
     slot, 376  
     style, 243  
     template, 242  
 elementy  
     formularzy, 331  
     kontrola widoczności, 268  
     nawigacji, 550, 592  
     oznaczanie, 292  
     pobieranie indeksu, 293  
     powtarzanie, 302  
     puste, 55  
     trasowanie URL, 535  
     typu select, 341  
     ukrywanie, 274  
     ustawianie właściwości, 282  
     wyświetlanie, 267  
     zawartość tekstowa, 263

**F**

faza  
     aktualizacji, 392  
     celu, 323  
     montażu, 390

przechwytywania, 322  
 tworzenia, 389  
 zniszczenia, 398  
 filtr currency, 131  
 filtrowanie, 305  
     zakończonych zadań, 34  
 filtry, 255  
     konfiguracja, 256  
     łączenie, 257  
 formatowanie  
     adresów URL, 514  
     wartości danych, 255, 343  
     znaczników, 364  
 formularze, 62, 331  
     walidacja danych, 141, 351  
     własne wartości, 348  
 framework  
     Bootstrap CSS, 194  
     do obsługi stylów CSS, 28  
 funkcje, 68, 248  
     administracyjne, 161, 173  
     administracyjne na żądanie, 183  
     haków, 624  
     haków dyrektyw, 618  
     importowanie, 89  
     jako metody, 85  
     obsługi błędów, 403  
     projektu Vue.js, 211  
     strzałkowe, 72  
     Vue.js, 613  
     walidacji, 354  
     z parametrami, 70  
     zaawansowane, 383  
     zmiana nazw, 89  
     zwracające wyniki, 71

**G**

getter, 463  
 Git, 23  
 globalne rejestrowanie dyrektyw, 619

**H**

HMR, Hot Module Replacement, 222  
 HTML, 51, 53  
 HTTP, 221

**I**  
 implementacja  
     edytora produktów, 178  
     funkcji koszyka, 124  
     uwierzytelniania, 161  
     zarządzania zamówieniami, 169  
 importowanie  
     modułu, 89  
     pojedynczych funkcji, 90  
 indeks  
     elementu, 293  
     w dyrektywie v-for, 295  
 instalacja  
     edytora, 23  
     narzędzia Git, 231  
     Node.js, 21  
     pakietu @vue/cli, 22  
     pakietu Bootstrap, 52  
     pakietu HTTP, 429  
     przeglądarki, 24  
 instrukcje, 68  
     warunkowe, 77  
 interpolacja tekstu, 30, 239  
 izolacja komponentów, 366

**J**

JavaScript, 65, 205  
 jawną konwersję typów, 78  
 język HTML, 53

**K**

kaskadowe arkusze stylów, *Patrz style CSS*  
 katalog dist, 235  
     pakietów, 216  
     productapp, 428  
     src, 215  
 klasy  
     aktywnej trasy, 545  
     Bootstrapa, 58  
     kontekstowe, 59  
     przejść, 596, 599  
     zagnieżdzone, 551  
 klawiatura, 328  
 klucz, 291

## kod

- asynchroniczny, 92
- HTML, 265
- źródłowy, 214
- kompilacja, 219
- kompilator Babel, 221
- komponent ProductEditor, 429
- komponent-dziecko, 365, 374
- komponent-rodzic, 369
- komponenty, 203, 359
  - administracyjne, 163, 167
  - asynchroniczne, 494
  - bez obsługi trasowania, 582
  - cykl życia, 385, 489
  - do wyświetlania produktu, 408
  - do wyświetlania zdarzeń, 446
  - domeszki, 627
  - dynamiczne, 481, 485
  - edytora produktu, 409
  - faza aktualizacji, 392
  - faza montażu, 390
  - faza tworzenia, 389
  - faza zniszczenia, 398
  - globalna rejestracja, 365
  - izolacja, 366
  - luźno powiązane, 405
  - mapowanie funkcji, 471
  - metody cyklu życia, 388
  - metody ochronne, 570
  - na żądanie, 577
  - obsługa błędów, 400, 403
  - omówienie nazw, 363
  - paczki, 500
  - pobieranie danych trasowania, 519
  - rejestracja, 204
  - rodzica i dziecka, 363
  - składniki, 242
  - sloty, 376
  - stosowanie propów, 368
  - strażnik trasy, 570, 577
  - tworzenie, 589
  - uwierzytelniania, 182
  - wdrażanie, 204
    - z komunikatem ładowania, 578
    - zmiany, 243
  - komunikacja
    - 41 między funkcjami haków, 624
    - komunikat, 201
    - konfiguracja
      - dopasowania tras URL, 513
      - filtrów, 256

## klas, 276

- komponentu, 204
- leniwego ładowania, 498
- lintera, 212
- narzędzi deweloperskich, 233
- segmentu dynamicznego, 523
- trasowania URL, 122, 592
- usługi, 440
- właściwości mode, 515
- konsolidacja aktualizacji, 393
- konteksty stylu Bootstrapa, 59
- kontener Dockera, 188, 189
- konwersja
  - liczb na łańcuchy znaków, 78
  - łańcuchów znaków na liczby, 79
- kopiowanie właściwości, 85
- koszyk zakupowy, 121, 124
  - testowanie funkcji, 132
  - utrwalanie, 132
  - widżet podsumowania, 135
  - wyswietlanie zawartości, 128

## Ł

## leniwe ładowanie, 496, 498, 577

- liczba elementów, 155
- liczby, 76
- linter, 212, 226
  - dostosowywanie reguł, 229
- lista
  - produktów, 104, 175
  - zadań, 31
- literały
  - obiektowe, 84
  - tablicowe, 80
  - tekstowe w atrybutach, 56
- lokalizacja aplikacji, 256

## Ł

## ładowanie komponentów na żądanie, 577

- łańcuchy
  - szablonowe, 75
  - znaków, 75
- łączenie
  - filtrów, 257
  - komponentów, 500
  - plików, 89
  - propów i zdarzeń, 374

**M**

magazyn danych, 101, 181, 449  
 akcje, 467  
 analiza zmian, 460  
 mapowanie funkcji, 471  
 moduły, 474  
 powiadomienia o zmianach, 468  
 rejestrowanie modułu, 475  
 rozszerzanie, 125, 162  
 stosowanie, 456  
 tworzenie, 452  
 używanie, 452  
 Vuex, 456  
 właściwości obliczane, 461

magazyn produktów, 103  
 mapowanie funkcji, 471  
 margines, 59  
 mechanizm  
 przejścia CSS, 595  
 Vue Routera, 511

metoda, 252  
 beforeRouteEnter, 572  
 beforeRouteUpdate, 533  
 get, 437

metody  
 cyklu życia, 388, 389  
 do obsługi tablic, 82  
 do obsługi zdarzeń, 313  
 HTTP, 433  
 nawigacji, 511  
 obsługi tablic, 83  
 ochronne komponentów, 570  
 typu string, 75  
 Vue, 633

model DOM, 195

moduły, 86  
 importowanie, 89  
 łączenie plików, 89  
 magazynu danych, 474  
 tworzenie, 86  
 tworzenie wielu mechanizmów, 88  
 używanie, 86  
 wieloplikowe, 90

modyfikacja  
 komponentu, 156  
 komunikatu, 201  
 zawartości tablicy, 80

**modyfikator**

number, 343  
 once, 326  
 trim, 345

**modyfikatory**  
 dyrektywy v-model, 343  
 obsługi zdarzeń, 320, 325  
 własnej dyrektywy, 622  
 zdarzeń klawiatury, 328  
 zdarzeń myszy, 327  
 mutacje, 454, 468  
 mysza, 327

**N**

**narzędzia**  
 deweloperskie, 25, 218  
 projektowe, 100  
 Vue.js, 209

**narzędzie**  
 Git, 23  
 NPM, 217

nasłuchiwanie zdarzeń, 317  
 nawigacja, 490  
 elementy HTML, 512  
 globalny strażnik, 562  
 obsługa zmian, 531

nawigowanie do adresów URL, 510  
 nazwy importowanych funkcji, 89  
 NPM, 217

**O**

**obiekt**  
 \$route, 521  
 Vue, 198, 199

**obiekty**, 82  
 globalne, 248  
 kopianie właściwości, 85  
 wyliczanie właściwości, 298

obietnica, 90, 437  
 obserwator, 396, 484

**obsługa**  
 akcji, 156  
 animacji, 598, 601  
 błędów, 444  
 komponentów, 400  
 ładowania, 581  
 danych, 150

- obsługa
    - danych odpowiedzi, 435
    - dyrektywy Repeater, 285
    - elementów formularzy, 331
    - elementów router-link, 537
    - liczb, 76
    - łańcuchów znaków, 75
    - mechanizmu wyboru produktów, 126
    - nazwanych elementów router-view, 553
    - obiektów, 82
    - starych adresów URL, 550
    - stronicowania listy, 108
    - stylów CSS, 28
    - tablic, 79
    - wartości logicznych, 74
    - wyboru kategorii, 114
    - wyszukiwania, 157
    - zdarzeń, 200, 309–313, 318, 320
    - zmian w nawigacji, 531
    - żądań HTTP, 434
  - ochrona tras, 562, 563
  - odbieranie zdarzeń, 421
  - odpowiedź HTTP, 437
  - ograniczenia zawartości elementów, 55
  - opcje
    - konfiguracji leniwego ładowania, 498
    - konfiguracyjne, 233
    - obserwatora, 398
    - reguł lintera, 226
  - operacje
    - asynchroniczne, 91, 464
    - HTTP, 444
  - operator
    - identyczności, 77
    - rozwinięcia, 81
    - równości, 77
  - operatory języka JavaScript, 76
  - opóźnianie aktualizacji, 344
  - oznaczanie elementu, 292
- P**
- pakiet
    - @vue/cli, 22
    - Bootstrap, 194
    - HTTP, 429
    - json-server, 98, 428
    - Vue Router, 508
    - Vuex, 450
  - pakiety
    - globalne, 217
    - lokalne, 217
  - parametr reszty, 70
  - parametry
    - domyślne, 70
    - metody beforeRouteUpdate, 533
  - pierwsze aplikacja, 21, 24
  - planowanie układu aplikacji, 547
  - pliki, 25
    - HTML, 206
    - JavaScript, 206
  - plik
    - jsprimer/package.json, 67
    - operations.js, 88
    - package.json, 99, 218
    - productapp/package.json, 406
    - productapp/restData.js, 428
    - ProductList.vue, 108
    - projecttools/package.json, 228, 229, 234
    - projecttools/server.js, 236
    - projecttools/vue.config.js, 233
    - sportsstore/authMiddleware.js, 99
    - sportsstore/data.js, 98, 149
    - sportsstore/data.json, 185
    - sportsstore/deploy-package.json, 188
    - sportsstore/server.js, 187
    - src/App.html, 207
    - src/App.vue, 26, 27, 33, 38, 47, 60, 62, 104, 119, 134, 203, 206, 222, 223, 234, 250, 292, 499, 507, 540, 541, 575, 602
    - src/Child.vue, 368
    - src/components/admin/Admin.vue, 168
    - src/components/admin/Administration.vue, 164, 183
    - src/components/admin/index.js, 183
    - src/components/admin/OrderAdmin.vue, 168, 170
    - src/components/admin/ProductAdmin.vue, 167, 176
    - src/components/admin/ProductEditor.vue, 177, 178
    - src/components/App.vue, 495
    - src/components/CartSummary.vue, 135
    - src/components/CategoryControls.vue, 116, 156
    - src/components/Checkout.vue, 138, 143, 144
    - src/components/Child.vue, 361, 368, 376
    - src/components/DataSummary.vue, 494
    - src/components/EditorField.vue, 409, 425

- src/components/FilteredData.vue, 572, 581
- src/components/ListMaker.vue, 590, 604
- src/components/ListMakerControls.vue, 591, 607
- src/components/MessageDisplay.vue, 401, 583
- src/components/Numbers.vue, 589, 609, 610, 614, 616, 620, 622, 627, 634
- src/components/OrderThanks.vue, 139
- src/components/PageControls.vue, 113, 151
- src/components/Preferences.vue, 547
- src/components/ProductDisplay.vue, 408, 422, 430, 442, 451, 472, 475, 483, 510, 519, 530, 536
- src/components/ProductEditor.vue, 48, 409, 424, 429, 512, 520, 524, 527, 529, 531
- src/components/ProductList.vue, 105, 107, 111, 126
- src/components/Search.vue, 160
- src/components/ShoppingCart.vue, 129
- src/components/ShoppingCartLine.vue, 128
- src/components/SimpleDisplay.vue, 589, 594, 598, 599
- src/components/Store.vue, 103, 106, 116, 136, 160
- src/components/Subtraction.vue, 628
- src/components/ValidationError.vue, 141
- src/directives/colorize.js, 616, 623, 624
- src/main.js, 28, 69, 97, 102, 131, 199, 220
- src/math/index.js, 91
- src/mixins/numbersMixin.js, 626
- src/plugins/math/componentFeatures.js, 631
- src/plugins/math/directives.js, 630
- src/plugins/math/filters.js, 630
- src/plugins/math/globals.js, 630
- src/plugins/math/Operation.vue, 631
- src/ProductEditor.vue, 416
- src/restDataSource.js, 440
- src/router/basicRoutes.js, 561, 575, 577, 580, 583
- src/router/index.js, 122, 140, 165, 166, 168, 177, 505, 518, 519, 522, 525, 526, 548, 555, 562, 563, 564, 568, 592
- src/store/auth.js, 162, 182
- src/store/cart.js, 125, 133
- src/store/index.js, 49, 102, 108, 115, 118, 125, 138, 153, 157, 174, 182, 453, 465, 490, 578
- src/store/orders.js, 137, 169
- src/store/preferences.js, 478
- src/validationRules.js, 353
- templatesanddata/package.json, 240
- pobieranie danych, 439
  - aplikacji, 483
  - trasowania, 519
- pochylenie, 595
- podpowiedzi wstępnego pobierania, 497
- polecenia narzędzia NPM, 217
- polimorfizm, 70
- powiadomienia o zmianach, 468
- powtarzanie
  - elementów, 302
  - treści, 302
- poziomy reguł lintera, 227
- projekt
  - Sklep sportowy, 95
  - Vue.js, 211, 214
- prop, 368, 372
- propagacja zdarzeń, 320, 324
- protokół CORS, 435
- przechowywanie danych, 38, 39
- przeglądarka, 24
- przejścia, 587, 593, 603
  - dla zmian w kolekcji, 604
  - początkowe, 597
- przekazywanie
  - argumentów, 464
  - funkcji przez argument, 71
  - propa, 585
- przekierowanie, 565
  - do trasy nazwanej, 566
  - żądania, 564
- przesłanianie usług, 413
- przestrzeń nazw modułów, 478
- przesunięcie, 595
- przetwarzanie danych, 438
- przycisk wyboru, 33

## R

- React, 45
- reagowanie na zmiany, 357, 609
- reaktywność, 250, 390
- reguły
  - lintera, 229
  - walidacji, 353
- rejestrowanie
  - dyrektyw, 619
  - komponentów, 138, 204, 365
  - modułu, 475
  - wtyczek, 633

renderowanie po stronie serwera, 45  
 REST, Representational State Transfer, 98, 433  
 restartowanie aplikacji, 29  
 REST-owa usługa sieciowa, 117, 162, 427, 433  
 routing, 122, 503  
 rozszerzanie  
     możliwości Vue.js, 613  
     magazynu danych, 125  
     Vue Devtools, 231

**S**

segmenty  
     dynamiczne, 526  
     opcjonalne, 526  
 sekwencka przejścia, 597  
 selektor CSS, 58, 273  
 serwer  
     deweloperski HTTP, 221  
     HTTP, 236, 428  
 siatka, 60  
 skalowanie, 149, 595  
 Sklep sportowy, 95  
     administrowanie, 149, 173  
     dodawanie  
         funkcji administracyjnych, 173  
         pakietów, 96  
         zamówień, 137  
     edytor produktów, 178  
     formularz walidacji, 141  
     funkcje administracyjne, 161  
     koszyk zakupowy, 121  
     lista produktów, 104, 106  
     magazyn danych, 101  
     magazyn produktów, 103  
     obsługa  
         danych, 150  
         rozliczenia, 137  
         wyszukiwania, 157  
     przetwarzanie cen, 106  
     REST-owa usługa sieciowa, 98, 117  
     rozliczenie, 121  
     skalowanie, 149  
     stronicowanie listy produktów, 108  
     style CSS, 97  
     uruchamianie aplikacji, 101  
     wdrażanie, 181, 185, 188  
     wybór kategorii, 114  
     wybór produktów, 126  
     wyświetlanie zawartości koszyka, 128

zamówienia, 121  
 zarządzanie zamówieniami, 169  
 sloty  
     nazwane, 377  
     o ograniczonym zasięgu, 378  
 słowo kluczowe  
     async, 92  
     await, 92  
     const, 73  
     debugger, 234  
     let, 72  
     new, 506  
     return, 245  
 sortowanie, 305  
 SPA, 44  
 SSR, server-side rendering, 45  
 stan, 454, 468  
 stosowanie  
     aliasu trasły, 289, 518  
     argumentów własnej dyrektywy, 621  
     biblioteki do obsługi animacji, 598  
     domieszki, 627  
     funkcji, 68  
     funkcji walidacji, 354  
     komponentów, 359  
     komponentów asynchronicznych, 494  
     lintera, 226  
     magazynu danych, 449, 456  
     modułów, 86  
     modułów magazynu danych, 474, 475  
     modyfikatorów  
         dyrektywy v-model, 343  
         obsługi zdarzeń, 320  
         własnej dyrektywy, 622  
     obietnic, 91  
     propa, 369  
     propów, 368  
     przejścia, 593, 601–604  
     przestrzeni nazw modułów, 478  
     REST-owych usług, 427  
     slotów komponentów, 376  
     slotów nazwanych, 377  
     slotów o ograniczonym zasięgu, 378  
     systemu trasowania, 507  
     właściwości obliczanych, 303  
     wtyczki, 633  
     wyrażeń regularnych, 525  
     wyrażeń własnych dyrektyw, 620  
     zdarzeń początkowych, 608  
     zdarzeń przejść, 606

strażnik  
 nawigacji, 562  
 trasy, 166, 577  
 stronicowanie, 151  
 danych, 303  
 struktura  
 dyrektywy v-bind, 275  
 dyrektywy v-on, 311  
 podkatalogów, 24  
 projektu, 213  
 style  
 Bootstrapa, 58  
 CSS, 28, 272  
 stylowanie  
 elementów  
 HTML, 29  
 łącza routera, 542  
 nawigacji, 543  
 formularzy, 62  
 tabel, 60  
 szablon, 205  
 szyna zdarzeń, 420, 421  
 lokalna, 424

## Ś

środowisko programistyczne, 21, 46

## T

tabele  
 naprzemienne kolorowanie wierszy, 296  
 tablice, 79, 287  
 modyfikacja zawartości, 80  
 odczyt, 80  
 przeglądanie zawartości, 81  
 stron, 155  
 wbudowane metody, 82  
 wykrywanie zmian, 296  
 zamiana obiektu, 298  
 testowanie  
 aplikacji, 186  
 funkcji koszyka, 132  
 klas zagnieżdżonych, 551  
 usługi sieciowej, 100  
 transformacja, 219  
 CSS, 595

trasowanie  
 do wyświetlania komponentów, 507  
 URL, 122, 503, 535, 601  
 konfiguracja, 592  
 zaawansowane, 559  
 trasy  
 definiowanie, 548  
 dynamiczne dopasowywanie, 522  
 nazwane, 528, 566  
 ochrona, 562  
 typu catchall, 517  
 zagnieżdzone, 546  
 treści  
 dynamiczne, 29  
 zastępcze edytora, 177  
 trwałe przechowywanie danych, 38  
 tworzenie  
 domieszek komponentów, 626  
 dwukierunkowych wiązań modeli, 333  
 elementów nawigacji, 550, 592  
 funkcji, 68  
 globalnego filtru, 131  
 komponentów, 138, 589  
 komponentów rozliczenia, 138  
 kontenera Dockera, 188, 189  
 listy produktów, 104  
 lokalnych szyn zdarzeń, 424  
 magazynu danych, 101, 452  
 magazynu produktów, 103  
 modułów, 86  
 obiektów komponentów, 390  
 obiektu Vue, 198  
 pliku danych, 185  
 pliku paczki, 188  
 projektu, 24, 210  
 projektu Sklep sportowy, 95  
 reaktywnych usług, 415  
 REST-owej usługi sieciowej, 98  
 siatki, 60  
 tras nazwanych, 528  
 tras zagnieżdżonych, 546  
 trasy typu catchall, 517  
 usługi, 411, 419  
 usługi HTTP, 440  
 usługi obsługi błędów, 444  
 wersji produkcyjnej, 234  
 wiązania, 334  
 wiązania do tablicy, 346

## tworzenie

- wierszy tabeli, 288
- własnych dyrektyw, 616
- własnych zdarzeń, 373
- wtyczki, 632
- wtyczki Vue.js, 629
- zadań, 36

## typy, 72

- prymitywne, 74

**U**

## układ aplikacji, 547

## ukrywanie elementów, 274, 582

## upraszczanie wiązań dwukierunkowych, 337

## URL

- trasowanie, 503

## uruchamianie

- aplikacji, 101, 189

- REST-owego serwera, 432

## usługa, 411

- HTTP, 440

- obsługi błędów, 444

- reaktywna, 415

- sieciowa, 98

- szyny zdarzeń, 420

## ustawianie

- atributów, 279, 280

- pojedynczych stylów, 277

- właściwości elementu, 282

- właściwości HTMLElement, 281

## usuwanie białych znaków, 345

## uwierzytelnianie, 161, 182

**V**

## Vue Router, 511

## Vue.js, 19, 43

**W**

## walidacja, 141, 144, 351–354

## wartości logiczne, 74

## wdrażanie, 185, 233

- aplikacji, 188

- komponentu, 204

- sklepu sportowego, 181

## wersje pakietów, 216

## wiązania

- do elementów typu select, 341

- do pól tekstowych, 338

- do przycisków, 339

- do różnych typów danych, 346

- do tablicy, 346

- dwukierunkowe, 333

- z elementami formularzy, 338

## wiązanie danych, 239, 247, 248

- wybór komponentów, 486

- wywoływanie metod, 254

## widżet, 135

## własne

- dyrektywy, 616, 622

- elementy HTML, 371

- wartości dla przycisków, 349

- wartości formularza, 348

- zdarzenia, 373

## właściwości, 58

- elementów, 283

- HTMLElement, 281

- obiektów zdarzeń, 313

- obiektu, 298, 300

- \$route, 521

- validacji, 142

- obliczane, 249, 250, 251, 303

- odpowiedzi Axios, 438

- w magazynie danych, 461

- zdefiniowane w obiekcie wiązania, 619

## wstrzykiwanie zależności, 405, 411, 412, 417

## wtyczki, 629, 632, 633

## wybór

- fragmentów zawartości, 270

- kategorii, 114

- komponentów, 486

- produktów, 126

- tablicy elementów, 346

- wyswietlanych elementów, 272

- zdarzenia nawigacji, 541

## wyliczanie właściwości obiektu, 298

## wyłączanie podpowiedzi, 497

## wyrażenia

- lambda, 72

- regularne, 525

- złożone, 247

## wysyłanie zdarzeń, 420

## wyszukiwanie produktów, 157

## wyświetlanie

- błędów, 224, 447
- kodu HTML, 265
- komponentów, 485, 507
- komponentów-dzieci, 410
- listy zadań, 31
- produktu, 408
- trasowanego komponentu, 123
- wartości danych, 244
- wybranych elementów, 267, 268
- zawartości koszyka, 128
- zdarzeń, 446
- komponentu, 487
- wartości danych, 246
- wywołanie zwrotne, 395
- wypoływanie metod, 254

**X**

## XSS, Cross-Site Scripting, 435

**Z**

## zadania

- tworzenie, 36
- zakończone, 34
- zamiana treści zastępczych, 26
- zamówienia, 169
- zarządzanie
  - propagacją zdarzeń, 320
  - zamówieniami, 169, 172
  - zdarzeniami obserwatora, 484
- zasada działania
  - SPA, 44
  - dyrektyw, 618
  - obietnic, 90

## zasięg CSS, 367

- zawartość tekstowa elementu, 263
- zdarzenia, 200, 309, 312
  - duplikacja, 326
  - faza celu, 323
  - faza przechwytywania, 322
  - klawiatury, 328
  - metody, 313
  - modyfikatory, 320, 325
  - myszy, 327
  - nasłuchiwanie, 317
  - nawigacji, 541
  - obserwatora, 484
  - obsługa, 311
  - odbieranie, 421
  - propagacja, 320, 324
  - przejść, 606
  - tworzenie, 373
  - właściwości obiektów, 313
  - wysyłanie, 420
  - wyświetlanie, 446

## złożoność aplikacji, 46

## zmiana

- adresu URL, 509
  - rozmiaru strony, 113
  - strażnika, 568
  - treści, 111
- zmienne, 72
- jako właściwości obiektów, 84

**ż**

## żądania HTTP, 221, 428, 436

# PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**  
<http://program-partnerski.helion.pl>

GRUPA  
**Helion**

## KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE  
ZGODNIE Z METODĄ

## BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

**WWW.HELIONSZKOLENIA.PL**