



# ASP.NET Core 3

Zaawansowane programowanie

Wydanie VIII

—

Adam Freeman

Helion 

Apress®

Tytuł oryginału: Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages, 8th Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-7890-2

First published in English under the title Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages by Adam Freeman, edition: 8

Copyright © 2020 by Adam Freeman

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

Polish edition copyright © 2022 by Helion S.A.  
All rights reserved.

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

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

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

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

Helion S.A.  
ul. Kościuszki 1c, 44-100 Gliwice  
tel. 32 231 22 19, 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:  
<https://ftp.helion.pl/przyklady/aspm8.zip>

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
<https://helion.pl/user/opinie/aspm8>  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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



# Spis treści

O autorze .....	23
O recenzencie technicznym .....	24
<b>Część I Wprowadzenie do ASP.NET Core .....</b>	<b>25</b>
<b>Rozdział 1. ASP.NET Core w szerszym kontekście .....</b>	<b>27</b>
Poznajemy ASP.NET Core .....	27
Poznajemy frameworki aplikacji .....	27
Poznajemy frameworki narzędziowe .....	29
Poznajemy platformę ASP.NET Core .....	30
Co powinienem wiedzieć? .....	31
Co jest potrzebne do wykonania przykładów zamieszczonych w książce? .....	31
Jaka platforma jest odpowiednia do wykonania przykładów zamieszczonych w książce? .....	31
Co zrobić w przypadku problemów podczas wykonywania przykładów? .....	31
Co zrobić w sytuacji, gdy znajdę błąd w książce? .....	32
Jaka jest struktura książki? .....	32
Część I. Wprowadzenie do ASP.NET Core .....	32
Część II. Platforma ASP.NET Core .....	32
Część III. Aplikacje ASP.NET Core .....	33
Część IV. Funkcje zaawansowane ASP.NET Core .....	33
Czego nie znajdę w książce? .....	33
Jak mogę skontaktować się z autorem? .....	33
Co zrobić, jeśli lektura książki sprawiła mi przyjemność? .....	34
Co zrobić, jeśli ta książka mnie zdenerwowała i chciałbym się poskarżyć? .....	34
Podsumowanie .....	34

<b>Rozdział 2.</b>	<b>Rozpoczęcie pracy z ASP.NET Core</b>	<b>35</b>
	Wybór edytora kodu źródłowego	35
	Instalacja Visual Studio	36
	Instalacja Visual Studio Code	39
	Tworzenie projektu ASP.NET Core	43
	Otworzenie projektu w Visual Studio	45
	Otworzenie projektu w Visual Studio Code	46
	Uruchomienie aplikacji ASP.NET Core	46
	Poznajemy punkty końcowe	49
	Poznajemy trasy	51
	Generowanie stron WWW	51
	Połączenie wszystkiego w całość	56
	Podsumowanie	56
<b>Rozdział 3.</b>	<b>Pierwsza aplikacja ASP.NET Core</b>	<b>57</b>
	Przygotowanie sceny	57
	Ut看wienie projektu	57
	Projektowanie modelu danych	59
	Ut看wienie drugiej metody akcji i widoku	60
	Łączenie metod akcji	61
	Budowanie formularza	63
	Obsługa formularzy	64
	Dodanie widoku Thanks	67
	Wyświetlenie odpowiedzi	68
	Dodanie kontroli poprawności danych	71
	Nadanie stylu zawartości	76
	Podsumowanie	82
<b>Rozdział 4.</b>	<b>Używanie narzędzi programistycznych</b>	<b>83</b>
	Ut看wienie projektu ASP.NET Core	83
	Ut看wienie projektu za pomocą narzędzi działających w wierszu polecenia	84
	Ut看wienie projektu za pomocą Visual Studio	88
	Dodanie kodu i treści do projektu	90
	Poznajemy proces tworzenia elementów szkieletowych	91
	Budowanie i uruchamianie projektu	93
	Budowanie i uruchamianie projektu z poziomu wiersza poleceń	94
	Budowanie i uruchamianie projektu za pomocą Visual Studio Code	94
	Budowanie i uruchamianie projektu za pomocą Visual Studio	95
	Zarządzanie pakietami	95
	Zarządzanie pakietami NuGet	96
	Zarządzanie pakietami narzędziowymi	97
	Zarządzanie pakietami działającymi po stronie klienta	97
	Zarządzanie pakietami za pomocą Visual Studio	99
	Debugowanie projektu	99
	Podsumowanie	102

<b>Rozdział 5. Najważniejsze cechy języka C# .....</b>	<b>103</b>
Utworzenie przykładowego projektu .....	104
Otworzenie projektu .....	104
Włączenie frameworka MVC .....	104
Utworzenie komponentów aplikacji ASP.NET Core .....	105
Użycie operatora warunkowego null .....	108
Łączenie operatorów warunkowych null .....	109
Łączenie operatorów: warunkowego i koalescencji .....	111
Użycie automatycznie implementowanych właściwości .....	113
Użycie automatycznie implementowanych metod inicjalizacyjnych właściwości .....	114
Utworzenie automatycznie implementowanych właściwości tylko do odczytu .....	115
Interpolacja ciągu tekstowego .....	116
Użycie inicjalizatorów obiektów i kolekcji .....	118
Użycie inicjalizatora indeksu .....	119
Dopasowanie wzorca .....	120
Dopasowanie wzorca w konstrukcji switch .....	121
Użycie metod rozszerzających .....	123
Stosowanie metod rozszerzających do interfejsów .....	124
Tworzenie filtrujących metod rozszerzających .....	126
Użycie wyrażeń lambda .....	128
Definiowanie funkcji .....	129
Użycie wyrażeń lambda w postaci metod i właściwości .....	132
Użycie inferencji typów i typów anonimowych .....	134
Użycie typów anonimowych .....	135
Użycie implementacji domyślnych w interfejsach .....	137
Użycie metod asynchronicznych .....	140
Bezpośrednia praca z zadaniami .....	140
Użycie słów kluczowych async i await .....	141
Używanie wyliczenia asynchronicznego .....	143
Pobieranie nazw .....	146
Podsumowanie .....	148
<b>Rozdział 6. Testy jednostkowe w aplikacji ASP.NET Core .....</b>	<b>149</b>
Utworzenie przykładowego projektu .....	150
Otworzenie projektu .....	150
Wybór numeru portu HTTP .....	151
Włączenie frameworka MVC .....	151
Utworzenie komponentów aplikacji ASP.NET Core .....	152
Uruchomienie przykładowej aplikacji .....	154
Utworzenie projektu testów jednostkowych .....	154
Usunięcie domyślnej klasy testu .....	155
Tworzenie i wykonywanie testów jednostkowych .....	156
Wykonywanie testów w oknie Eksplorator testów w Visual Studio .....	158
Wykonywanie testów w Visual Studio Code .....	158
Wykonywanie testów w wierszu poleceń .....	159
Poprawienie testu jednostkowego .....	160

Izolowanie komponentów poddawanych testom jednostkowym .....	161
Użycie pakietu imitacji .....	166
Utworzenie obiektu imitacji .....	166
Podsumowanie .....	168
<b>Rozdział 7. SportsStore — kompletna aplikacja .....</b>	<b>169</b>
Utworzenie projektów .....	170
Utworzenie projektu testów jednostkowych .....	170
Utworzenie katalogów projektu aplikacji .....	171
Otworzenie projektów .....	171
Przygotowanie usług aplikacji i potoku żądania .....	172
Konfiguracja silnika widoku Razor .....	173
Utworzenie kontrolera i widoku .....	175
Tworzenie modelu danych .....	176
Sprawdzenie i uruchomienie aplikacji .....	176
Dodanie danych do aplikacji .....	177
Instalowanie pakietów narzędzi Entity Framework Core .....	177
Definiowanie ciągu tekstowego połączenia .....	177
Utworzenie klasy kontekstu bazy danych .....	178
Konfigurowanie Entity Framework Core .....	179
Tworzenie repozytorium .....	180
Utworzenie i zastosowanie migracji bazy danych .....	183
Tworzenie danych początkowych .....	183
Wyświetlanie listy produktów .....	186
Przygotowanie kontrolera .....	187
Uaktualnienie widoku .....	189
Uruchamianie aplikacji .....	189
Dodanie stronicowania .....	190
Wyświetlanie łączy stron .....	192
Ulepszanie adresów URL .....	200
Dodawanie stylu .....	202
Instalacja pakietu Bootstrap .....	203
Zastosowanie w aplikacji stylów Bootstrap .....	203
Tworzenie widoku częściowego .....	205
Podsumowanie .....	207
<b>Rozdział 8. SportsStore — nawigacja i koszyk na zakupy .....</b>	<b>208</b>
Dodawanie kontrolki nawigacji .....	208
Filtrowanie listy produktów .....	208
Ulepszanie schematu URL .....	213
Budowanie menu nawigacji po kategoriach .....	217
Poprawianie licznika stron .....	225
Budowanie koszyka na zakupy .....	228
Konfigurowanie Razor Pages .....	228
Utworzenie Razor Pages .....	230
Tworzenie przycisków koszyka .....	231
Włączenie obsługi sesji .....	233

Implementowanie funkcjonalności koszyka .....	234
Dokończenie pracy z Razor Page .....	238
Podsumowanie .....	244
<b>Rozdział 9. SportsStore — ukończenie koszyka na zakupy .....</b>	<b>245</b>
Dopracowanie modelu koszyka za pomocą usługi .....	245
Tworzenie klasy koszyka obsługującej magazyn danych .....	246
Rejestrowanie usługi .....	248
Uproszczenie klasy Cart frameworka Razor Pages .....	249
Kończenie budowania koszyka .....	251
Usuwanie produktów z koszyka .....	251
Dodawanie podsumowania koszyka .....	254
Składanie zamówień .....	256
Utworzenie klasy modelu .....	256
Dodawanie procesu składania zamówienia .....	258
Utworzenie kontrolera i widoku .....	258
Implementowanie mechanizmu przetwarzania zamówień .....	260
Zakończenie pracy nad kontrolerem koszyka .....	265
Wyświetlanie informacji o błędach systemu kontroli poprawności .....	268
Wyświetlanie strony podsumowania .....	270
Podsumowanie .....	271
<b>Rozdział 10. SportsStore — administracja .....</b>	<b>272</b>
Przygotowanie serwera Blazor .....	272
Utworzenie pliku importów .....	274
Utworzenie strony początkowej Razor Page .....	274
Utworzenie komponentów routingu i układu .....	275
Utworzenie komponentów Razor .....	276
Sprawdzenie konfiguracji Blazor .....	276
Zarządzanie zamówieniami .....	277
Uśprawnienie modelu .....	278
Wyświetlanie zamówień administratorowi .....	279
Dodajemy zarządzanie katalogiem .....	282
Rozszerzenie repozytorium .....	283
Dodanie kontroli poprawności modelu .....	284
Tworzenie komponentu listy .....	285
Tworzenie komponentu widoku szczegółowego .....	286
Tworzenie komponentu edytora .....	288
Usuwanie produktu .....	290
Podsumowanie .....	293
<b>Rozdział 11. SportsStore — bezpieczeństwo i wdrożenie aplikacji .....</b>	<b>294</b>
Zabezpieczanie funkcji administracyjnych .....	294
Utworzenie bazy danych dla systemu Identity .....	294
Dodanie konwencjonalnej funkcjonalności administracyjnej .....	300
Zdefiniowanie prostej polityki autoryzacji .....	302
Utworzenie kontrolera AccountController i widoków .....	303
Przetestowanie polityki bezpieczeństwa .....	307

Przygotowanie ASP.NET Core do wdrożenia .....	307
Konfiguracja obsługi błędów .....	307
Utworzenie produkcyjnych ustawień konfiguracyjnych .....	309
Utworzenie obrazu Dockera .....	310
Uruchomienie skonteneryzowanej aplikacji .....	312
Podsumowanie .....	313
<b>Część II Platforma ASP.NET Core .....</b>	<b>315</b>
<b>Rozdział 12. Poznajemy platformę ASP.NET Core .....</b>	<b>317</b>
Utworzenie przykładowego projektu .....	318
Uruchomienie przykładowej aplikacji .....	319
Poznajemy platformę ASP.NET Core .....	320
Poznajemy oprogramowanie pośredniczące i potok żądania .....	320
Poznajemy usługi .....	321
Poznajemy projekt ASP.NET Core .....	322
Poznajemy punkt wyjścia aplikacji .....	323
Poznajemy klasę Startup .....	324
Poznajemy plik projektu .....	325
Tworzenie własnego oprogramowania pośredniczącego .....	327
Definiowanie oprogramowania pośredniczącego za pomocą klasy .....	331
Poznajemy zwrotną ścieżkę dostępu potoku żądania .....	334
Skrócenie potoku żądania .....	336
Tworzenie odgałęzienia potoku żądania .....	338
Utworzenie końcowego oprogramowania pośredniczącego .....	340
Konfiguracja oprogramowania pośredniczącego .....	343
Używanie wzorca opcji z bazującym na klasie komponentem oprogramowania pośredniczącego .....	345
Podsumowanie .....	347
<b>Rozdział 13. Routing URL .....</b>	<b>348</b>
Utworzenie przykładowego projektu .....	349
Poznajemy routing URL .....	353
Dodanie komponentu oprogramowania pośredniczącego routingu i zdefiniowanie punktu końcowego .....	353
Poznajemy wzorce adresu URL .....	357
Używanie zmiennych segmentów we wzorcach URL .....	358
Generowanie adresów URL na podstawie tras .....	363
Zarządzanie dopasowaniem adresów URL .....	367
Dopasowanie wielu wartości z pojedynczego segmentu adresu URL .....	367
Używanie wartości domyślnych dla zmiennych segmentów .....	369
Używanie segmentów opcjonalnych we wzorcu adresu URL .....	369
Używanie zmiennej segmentu catchall .....	371
Ograniczenia podczas dopasowywania segmentów .....	372
Definiowanie trasy awaryjnej .....	376



Zaawansowane funkcje routingu .....	377
Tworzenie ograniczeń niestandardowych .....	378
Unikanie wyjątków związanych z niedopasowaniem trasy .....	379
Uzyskanie dostępu do punktu końcowego z poziomu komponentu oprogramowania pośredniczącego .....	382
Podsumowanie .....	384
<b>Rozdział 14. Wstrzykiwanie zależności .....</b>	<b>385</b>
Utworzenie przykładowego projektu .....	386
Utworzenie komponentu oprogramowania pośredniczącego i punktu końcowego .....	387
Konfiguracja potoku żądania .....	388
Poznajemy położenie usługi i ściśle powiązanie komponentów .....	390
Problem związany z położeniem usługi .....	391
Problem związany ze ścisłym powiązaniem komponentów .....	393
Używanie mechanizmu wstrzykiwania zależności .....	396
Używanie usługi w klasie oprogramowania pośredniczącego .....	397
Używanie usługi w punkcie końcowym .....	399
Cykl życiowy usługi .....	404
Tworzenie usługi tymczasowej .....	405
Unikanie problemów związanych z ponownym używaniem usługi tymczasowej .....	406
Używanie usługi zasięgu .....	410
Inne funkcje mechanizmu wstrzykiwania zależności .....	416
Utworzenie łańcucha zależności .....	416
Dostęp do usług w metodzie <code>ConfigureServices()</code> .....	418
Używanie funkcji fabryki usługi .....	419
Tworzenie usługi z wieloma implementacjami .....	421
Używanie usługi pozbawionej typu .....	424
Podsumowanie .....	425
<b>Rozdział 15. Używanie funkcjonalności platformy — część I .....</b>	<b>426</b>
Utworzenie przykładowego projektu .....	427
Używanie usługi konfiguracji .....	429
Poznajemy plik konfiguracyjny dla danego środowiska .....	430
Uzyskiwanie dostępu do ustawień konfiguracyjnych .....	431
Używanie danych konfiguracyjnych w usługach .....	433
Poznajemy plik ustawień początkowych .....	435
Ustalanie środowiska za pomocą kodu w klasie <code>Startup</code> .....	441
Przechowywanie kluczy tajnych użytkownika .....	442
Używanie usługi rejestrowania danych .....	445
Generowanie komunikatów .....	446
Określenie minimalnego poziomu rejestrowanych komunikatów .....	450
Używanie treści statycznej i pakietów działających po stronie klienta .....	451
Dodanie komponentu oprogramowania pośredniczącego odpowiedzialnego za obsługę treści statycznej .....	452
Używanie pakietów po stronie klienta .....	457
Podsumowanie .....	460

<b>Rozdział 16. Używanie funkcjonalności platformy — część II .....</b>	<b>461</b>
Utworzenie przykładowego projektu .....	462
Używanie mechanizmu ciasteczek .....	463
Obsługa zgody na użycie ciasteczek .....	466
Zarządzanie zgodą na używanie ciasteczek .....	468
Mechanizm sesji .....	471
Konfiguracja usługi sesji i komponentu oprogramowania pośredniczącego .....	471
Używanie danych sesji .....	474
Praca z połączeniami HTTPS .....	476
Włączanie obsługi połączeń HTTPS .....	476
Wykrywanie żądań HTTPS .....	478
Wymuszanie obsługi żądań HTTPS .....	480
Włączenie protokołu HTTP Strict Transport Security .....	481
Obsługa wyjątków i błędów .....	484
Zwrot odpowiedzi HTML zawierającej informacje o błędzie .....	486
Usprawnianie odpowiedzi zawierających kody stanu .....	489
Filtrowanie żądań za pomocą nagłówka Host .....	491
Podsumowanie .....	494
<b>Rozdział 17. Praca z danymi .....</b>	<b>495</b>
Utworzenie przykładowego projektu .....	497
Buforowanie danych .....	499
Buforowanie wartości danych .....	501
Używanie współdzielonego i trwałego bufora danych .....	505
Buforowanie odpowiedzi .....	509
Entity Framework Core .....	512
Instalacja Entity Framework Core .....	513
Tworzenie modelu danych .....	514
Konfiguracja usługi bazy danych .....	515
Tworzenie i stosowanie migracji bazy danych .....	517
Przygotowanie bazy danych .....	517
Używanie danych w punkcie końcowym .....	520
Podsumowanie .....	523
<b>Część III Aplikacje ASP.NET Core .....</b>	<b>525</b>
<b>Rozdział 18. Utworzenie przykładowego projektu .....</b>	<b>527</b>
Utworzenie projektu .....	527
Dodawanie modelu danych .....	528
Dodawanie pakietów NuGet do projektu .....	528
Tworzenie modelu danych .....	529
Przygotowanie bazy danych .....	531
Konfiguracja usług Entity Framework Core i komponentu oprogramowania pośredniczącego .....	533
Tworzenie i stosowanie migracji .....	534

Dodawanie frameworka CSS .....	535
Konfigurowanie potoku żądania .....	535
Uruchomienie przykładowej aplikacji .....	537
Podsumowanie .....	537
<b>Rozdział 19. Tworzenie usługi sieciowej RESTful .....</b>	<b>538</b>
Utworzenie przykładowego projektu .....	539
Poznajemy usługi sieciowe RESTful .....	540
Poznajemy metody i adresy URL żądania .....	540
Poznajemy format JSON .....	541
Tworzenie usługi sieciowej za pomocą niestandardowego punktu końcowego .....	542
Utworzenie usługi sieciowej za pomocą kontrolera .....	545
Włączenie obsługi frameworka MVC .....	545
Tworzenie kontrolera .....	547
Usprawnienie usługi sieciowej .....	557
Używanie akcji asynchronicznych .....	558
Uniknięcie zbędnego dołączania modelu .....	560
Używanie wyniku akcji .....	561
Sprawdzanie poprawności danych .....	568
Stosowanie atrybutu kontrolera API .....	570
Pomijanie właściwości null .....	571
Podsumowanie .....	574
<b>Rozdział 20. Funkcje zaawansowane usługi sieciowej .....</b>	<b>575</b>
Utworzenie przykładowego projektu .....	576
Usunięcie bazy danych .....	577
Uruchomienie przykładowej aplikacji .....	577
Praca z powiązаныmi ze sobą danymi .....	578
Usunięcie odwołania cyklicznego w danych .....	580
Obsługa metody HTTP PATCH .....	581
Poznajemy standard JSON Patch .....	581
Instalacja i konfiguracja pakietu JSON Patch .....	582
Definiowanie metody akcji .....	583
Formatowanie treści .....	584
Poznajemy domyślną politykę treści .....	584
Poznajemy negocjację treści .....	586
Określanie formatu danych akcji .....	590
Pobranie formatu danych z adresu URL .....	592
Ograniczanie formatów otrzymywanych przez metodę akcji .....	593
Dokumentowanie i analizowanie usług sieciowych .....	595
Rozwiązywanie konfliktów metod akcji .....	595
Instalacja i konfiguracja pakietu Swashbuckle .....	596
Dopracowanie opisu API .....	599
Podsumowanie .....	603

<b>Rozdział 21. Używanie kontrolerów z widokami — część I</b> .....	<b>604</b>
Utworzenie przykładowego projektu .....	605
Usunięcie bazy danych .....	606
Uruchomienie przykładowej aplikacji .....	607
Rozpoczęcie pracy z widokiem .....	607
Konfiguracja aplikacji .....	607
Utworzenie kontrolera HTML .....	609
Tworzenie widoku Razor .....	612
Wybór widoku na podstawie nazwy .....	614
Praca z widokami Razor .....	618
Wybór typu modelu widoku .....	621
Używanie pliku poleceń importujących widoki .....	624
Poznajemy składnię Razor .....	626
Poznajemy dyrektywy .....	627
Poznajemy wyrażenia treści .....	627
Zdefiniowanie elementu treści .....	628
Zdefiniowanie wartości atrybutu .....	629
Użycie konstrukcji warunkowych .....	630
Wyświetlanie sekwencji .....	634
Używanie bloków kodu Razor .....	636
Podsumowanie .....	637
<b>Rozdział 22. Używanie kontrolerów z widokami — część II</b> .....	<b>638</b>
Utworzenie przykładowego projektu .....	639
Usunięcie bazy danych .....	640
Uruchomienie przykładowej aplikacji .....	641
Używanie obiektu ViewBag .....	641
Używanie danych tymczasowych .....	643
Praca z układami .....	646
Konfiguracja układu za pomocą obiektu ViewBag .....	648
Używanie pliku ViewStart .....	650
Nadpisanie układu domyślnego .....	651
Zastosowanie sekcji układu .....	655
Użycie widoków częściowych .....	661
Włączanie widoków częściowych .....	661
Tworzenie widoku częściowego .....	662
Zastosowanie widoku częściowego .....	662
Poznajemy kodowanie treści .....	665
Kodowanie treści HTML .....	666
Kodowanie JSON .....	668
Podsumowanie .....	669
<b>Rozdział 23. Strony Razor</b> .....	<b>670</b>
Utworzenie przykładowego projektu .....	671
Uruchomienie przykładowej aplikacji .....	672

Poznajemy strony Razor .....	672
Konfiguracja stron Razor .....	673
Tworzenie strony Razor .....	674
Poznajemy routing stron Razor .....	678
Określanie wzorca routingu na stronie Razor .....	680
Dodawanie tras stron Razor .....	682
Poznajemy klasę modelu strony .....	684
Używanie pliku klasy ukrytej .....	684
Poznajemy wynik akcji na stronie Razor .....	688
Obsługa wielu metod HTTP .....	692
Wybór metody procedury obsługi .....	694
Poznajemy widok strony Razor .....	696
Utworzenie układu dla strony Razor .....	696
Używanie widoków częściowych na stronach Razor .....	698
Tworzenie strony Razor bez modelu strony .....	699
Podsumowanie .....	701
<b>Rozdział 24. Komponenty widoku .....</b>	<b>702</b>
Utworzenie przykładowego projektu .....	703
Usunięcie bazy danych .....	705
Uruchomienie przykładowej aplikacji .....	706
Poznajemy komponent widoku .....	706
Utworzenie komponentu widoku .....	707
Zastosowanie komponentu widoku .....	708
Poznajemy wynik działania komponentu widoku .....	711
Zwrot widoku częściowego .....	712
Zwrot fragmentów kodu HTML .....	715
Pobieranie danych kontekstu .....	718
Użycie argumentów do przekazania kontekstu z widoku nadrzędnego .....	719
Tworzenie asynchronicznego komponentu widoku .....	722
Utworzenie klasy komponentu widoku .....	725
Utworzenie hybrydowej klasy kontrolera .....	727
Podsumowanie .....	730
<b>Rozdział 25. Poznajemy atrybuty pomocnicze znaczników .....</b>	<b>731</b>
Utworzenie przykładowego projektu .....	732
Usunięcie bazy danych .....	734
Uruchomienie przykładowej aplikacji .....	734
Utworzenie atrybutu pomocniczego znacznika .....	735
Zdefiniowanie klasy atrybutu pomocniczego znacznika .....	736
Rejestrowanie atrybutu pomocniczego znacznika .....	739
Użycie atrybutu pomocniczego znacznika .....	739
Zawężanie zasięgu atrybutu pomocniczego znacznika .....	741
Rozszerzenie zasięgu atrybutu pomocniczego znacznika .....	742
Zaawansowane funkcje atrybutu pomocniczego znacznika .....	744
Tworzenie elementów skrótu .....	745
Programowe tworzenie elementów .....	747

Umieszczanie treści przed elementem i po nim .....	748
Pobieranie danych kontekstu widoku .....	752
Praca z modelem widoku .....	755
Koordinacja między atrybutami pomocniczymi znaczników .....	759
Zawieszenie wygenerowania elementu .....	761
Używanie komponentów atrybutu pomocniczego znacznika .....	763
Utworzenie komponentu atrybutu pomocniczego znacznika .....	763
Zwiększenie puli elementów obsługiwanych przez komponent atrybutu pomocniczego znacznika .....	766
Podsumowanie .....	768
<b>Rozdział 26. Używanie wbudowanych atrybutów pomocniczych znaczników .....</b>	<b>769</b>
Przygotowanie przykładowego projektu .....	770
Dodawanie pliku obrazu .....	772
Instalowanie pakietu działającego po stronie klienta .....	772
Usunięcie bazy danych .....	773
Uruchomienie przykładowej aplikacji .....	773
Włączanie wbudowanych atrybutów pomocniczych znaczników .....	774
Przekształcanie znaczników <a> .....	774
Używanie znaczników <a> na stronach Razor .....	776
Używanie atrybutów pomocniczych znaczników obsługujących pliki JavaScript i CSS .....	777
Zarządzanie plikami JavaScript .....	778
Zarządzanie arkuszami stylów CSS .....	787
Praca ze znacznikiem <image> .....	790
Użycie buforowanych danych .....	792
Określenie czasu wygaśnięcia buforowanej treści .....	794
Używanie atrybutu pomocniczego znacznika <environment> .....	797
Podsumowanie .....	798
<b>Rozdział 27. Użycie atrybutów pomocniczych znaczników formularza .....</b>	<b>799</b>
Przygotowanie przykładowego projektu .....	800
Usunięcie bazy danych .....	801
Uruchomienie przykładowej aplikacji .....	802
Poznajemy wzorzec obsługi formularza .....	803
Tworzenie kontrolera do obsługi formularza .....	803
Tworzenie strony Razor przeznaczonej do obsługi formularza .....	806
Używanie atrybutów pomocniczych znaczników do usprawnienia formularzy .....	808
Praca ze znacznikami formularza HTML .....	808
Transformacja przycisków formularza .....	810
Praca ze znacznikami <input> .....	812
Transformacja atrybutu type elementu <input> .....	813
Formatowanie wartości danych .....	815
Wyświetlanie w znaczniku <input> wartości z powiązanych ze sobą danych .....	818
Praca ze znacznikiem <label> .....	823
Praca ze znacznikami <select> i <option> .....	824
Dodanie treści do znacznika <select> .....	826
Praca ze znacznikiem <textarea> .....	828

Zabezpieczenie przed atakami typu CSRF .....	830
Włączanie w kontrolerze zabezpieczenia przed atakami typu CSRF .....	831
Włączanie na stronie Razor zabezpieczenia przed atakami typu CSRF .....	833
Klienty JavaScript i tokeny zabezpieczające przed atakami typu CSRF .....	834
Podsumowanie .....	837
<b>Rozdział 28. Dołączanie modelu .....</b>	<b>838</b>
Utworzenie przykładowego projektu .....	839
Usunięcie bazy danych .....	840
Uruchomienie przykładowej aplikacji .....	840
Poznajemy dołączanie modelu .....	841
Dołączanie typów prostych .....	843
Dołączanie typów prostych na stronach Razor .....	844
Poznajemy dołączanie wartości domyślnej .....	845
Dołączanie typów złożonych .....	848
Dołączanie do właściwości .....	850
Dołączanie zagnieżdżonych typów złożonych .....	852
Selektywne dołączanie właściwości .....	856
Selektywne dołączanie właściwości w klasie modelu .....	857
Dołączanie tablic i kolekcji .....	859
Dołączanie do tablic .....	859
Dołączanie do prostej kolekcji .....	862
Dołączanie do słownika .....	864
Dołączanie kolekcji typów złożonych .....	866
Określanie źródła dołączania modelu .....	869
Określanie źródła danych dla właściwości .....	871
Użycie nagłówków jako źródła danych dla funkcji dołączania modelu .....	873
Użycie treści żądania jako źródła danych dla funkcji dołączania modelu .....	874
Ręczne dołączanie modelu .....	875
Podsumowanie .....	877
<b>Rozdział 29. Kontrola poprawności danych modelu .....</b>	<b>878</b>
Utworzenie przykładowego projektu .....	879
Usunięcie bazy danych .....	881
Uruchomienie przykładowej aplikacji .....	881
Potrzeba stosowania kontroli poprawności danych modelu .....	881
Jawna kontrola poprawności modelu .....	882
Wyświetlenie użytkownikowi błędów podczas kontroli poprawności .....	886
Wyświetlanie komunikatów kontroli poprawności .....	888
Wyświetlanie błędów kontroli poprawności na poziomie właściwości .....	893
Wyświetlanie błędów kontroli poprawności na poziomie modelu .....	895
Jawne sprawdzanie danych na stronie Razor .....	897
Definiowanie reguł poprawności za pomocą metadanych .....	901
Tworzenie własnego atrybutu kontroli poprawności .....	905
Użycie kontroli poprawności po stronie klienta .....	910
Wykonywanie zdalnej kontroli poprawności .....	913
Wykonywanie zdalnej kontroli poprawności na stronie Razor .....	916
Podsumowanie .....	918

<b>Rozdział 30. Filtry .....</b>	<b>919</b>
Utworzenie przykładowego projektu .....	920
Włączenie obsługi połączeń HTTPS .....	921
Usunięcie bazy danych .....	922
Uruchomienie przykładowej aplikacji .....	922
Użycie filtrów .....	923
Użycie filtrów na stronach Razor .....	927
Poznajemy filtry .....	929
Tworzenie własnych filtrów .....	931
Użycie filtrów autoryzacji .....	931
Używanie filtrów zasobów .....	933
Użycie filtrów akcji .....	938
Używanie filtrów strony .....	943
Używanie filtru wyniku .....	947
Użycie filtrów wyjątków .....	952
Utworzenie filtru wyjątku .....	953
Zarządzanie cyklem życiowym filtru .....	955
Utworzenie fabryki filtrów .....	957
Używanie zasięgu mechanizmu wstrzykiwania zależności do zarządzania cyklem życiowym filtrów .....	959
Użycie filtrów globalnych .....	961
Poznajemy i zmieniamy kolejność wykonywania filtrów .....	963
Zmiana kolejności filtrów .....	965
Podsumowanie .....	968
<b>Rozdział 31. Utworzenie aplikacji bazującej na formularzach .....</b>	<b>969</b>
Utworzenie przykładowego projektu .....	969
Usunięcie bazy danych .....	972
Uruchomienie przykładowej aplikacji .....	972
Utworzenie aplikacji bazującej na formularzach MVC .....	973
Przygotowanie modelu widoku i widoku .....	973
Odczyt danych .....	975
Tworzenie danych .....	977
Edycja danych .....	981
Usunięcie danych .....	984
Utworzenie bazującej na formularzach aplikacji stron Razor .....	986
Tworzenie wspólnej funkcjonalności .....	988
Definiowanie stron dla operacji CRUD .....	991
Tworzenie nowych obiektów powiązanych ze sobą danych .....	993
Dostarczanie w tym samym żądaniu powiązanych ze sobą danych .....	993
Utworzenie nowych danych w innym miejscu .....	997
Podsumowanie .....	1001



<b>Część IV</b>	<b>Funkcje zaawansowane ASP.NET Core .....</b>	<b>1003</b>
<b>Rozdział 32.</b>	<b>Utworzenie przykładowego projektu .....</b>	<b>1005</b>
	Utworzenie projektu .....	1005
	Dodawanie pakietów NuGet do projektu .....	1006
	Dodawanie modelu danych .....	1007
	Przygotowanie bazy danych .....	1008
	Konfiguracja usług Entity Framework Core i komponentu oprogramowania pośredniczącego .....	1010
	Tworzenie i stosowanie migracji .....	1012
	Dodawanie frameworka CSS .....	1012
	Konfigurowanie usług i komponentu oprogramowania pośredniczącego .....	1013
	Tworzenie kontrolera i widoku .....	1014
	Tworzenie strony Razor .....	1016
	Uruchomienie przykładowej aplikacji .....	1018
	Podsumowanie .....	1019
<b>Rozdział 33.</b>	<b>Praca z serwerem Blazor — część I .....</b>	<b>1020</b>
	Utworzenie przykładowego projektu .....	1021
	Poznajemy serwer Blazor .....	1022
	Poznajemy zalety serwera Blazor .....	1024
	Poznajemy wady serwera Blazor .....	1024
	Wybór między serwerem Blazor i frameworkiem Angular, React lub Vue.js .....	1024
	Rozpoczęcie pracy z technologią Blazor .....	1025
	Konfiguracja ASP.NET Core do pracy z serwerem Blazor .....	1025
	Tworzenie komponentu Razor .....	1027
	Poznajemy podstawy funkcjonalności komponentu Razor .....	1032
	Poznajemy zdarzenia Blazor i dołączanie danych .....	1033
	Wykorzystanie mechanizmu dołączania danych .....	1041
	Używanie klasy do definiowania komponentu .....	1047
	Używanie pliku ukrytego kodu .....	1047
	Definiowanie klasy komponentu Razor .....	1048
	Podsumowanie .....	1050
<b>Rozdział 34.</b>	<b>Praca z serwerem Blazor — część II .....</b>	<b>1051</b>
	Utworzenie przykładowego projektu .....	1051
	Łączenie komponentów .....	1052
	Konfiguracja komponentu za pomocą atrybutu .....	1054
	Tworzenie niestandardowych zdarzeń i dołączania danych .....	1060
	Wyświetlanie treści potomnej w komponencie .....	1066
	Tworzenie komponentu szablonu .....	1068
	Używanie parametrów typu generycznego w komponencie szablonu .....	1070
	Parametry kaskadowe .....	1076
	Obsługa błędów .....	1079
	Błędy dotyczące połączenia .....	1080
	Obsługa niewychwyczonych błędów aplikacji .....	1082
	Podsumowanie .....	1085

<b>Rozdział 35. Funkcje zaawansowane Blazor</b> .....	<b>1086</b>
Utworzenie przykładowego projektu .....	1087
Używanie routingu komponentu .....	1088
Przygotowania dla stron Razor .....	1089
Dodawanie tras do komponentów .....	1090
Poruszanie się między komponentami stosującymi routing .....	1094
Otrzymywanie danych routingu .....	1097
Używanie układu do zdefiniowania wspólnej treści .....	1099
Metody cyklu życiowego komponentu .....	1101
Używanie metod cyklu życiowego w zadaniach asynchronicznych .....	1103
Zarządzanie interakcjami komponentu .....	1106
Używanie odniesień do komponentów potomnych .....	1106
Praca z komponentami pochodzącymi z innego kodu .....	1109
Współpraca z komponentami za pomocą JavaScriptu .....	1114
Podsumowanie .....	1122
<b>Rozdział 36. Dane i formularze Blazor</b> .....	<b>1123</b>
Utworzenie przykładowego projektu .....	1124
Usunięcie bazy danych i uruchomienie aplikacji .....	1127
Używanie komponentów formularza Blazor .....	1127
Tworzenie niestandardowych komponentów formularza .....	1129
Weryfikacja danych formularza .....	1133
Obsługa zdarzeń formularza .....	1137
Używanie Entity Framework Core z technologią Blazor .....	1139
Kwestia zasięgu kontekstu w Entity Framework Core .....	1140
Problem związany z powtórzonym zapytaniem .....	1144
Przeprowadzanie operacji CRUD .....	1150
Utworzenie komponentu List .....	1150
Utworzenie komponentu Details .....	1152
Tworzenie komponentu Editor .....	1153
Rozbudowa funkcjonalności formularza Blazor .....	1156
Tworzenie niestandardowego ograniczenia weryfikacji .....	1157
Tworzenie wysyłającego formularz przycisku, który będzie aktywny, jedynie gdy dane są poprawne .....	1160
Podsumowanie .....	1162
<b>Rozdział 37. Używanie Blazor Web Assembly</b> .....	<b>1163</b>
Utworzenie przykładowego projektu .....	1165
Usunięcie bazy danych i uruchomienie aplikacji .....	1166
Konfiguracja Blazor WebAssembly .....	1167
Tworzenie projektu współdzielonego .....	1167
Tworzenie projektu Blazor WebAssembly .....	1168
Przygotowanie projektu ASP.NET Core .....	1169
Dodawanie odwołania do pliku rozwiązania .....	1169
Otworzenie projektów .....	1169
Dokończenie konfiguracji Blazor WebAssembly .....	1169
Testowanie miejsc zarezerwowanych dla komponentów .....	1173

Tworzenie komponentu Blazor WebAssembly .....	1173
Importowanie przestrzeni nazw modelu danych .....	1173
Tworzenie komponentu .....	1174
Tworzenie układu .....	1178
Definiowanie stylów CSS .....	1179
Ukończenie aplikacji Blazor WebAssembly .....	1179
Tworzenie komponentu Details .....	1179
Tworzenie komponentu Editor .....	1181
Podsumowanie .....	1184
<b>Rozdział 38. Użycie ASP.NET Core Identity .....</b>	<b>1185</b>
Utworzenie przykładowego projektu .....	1186
Przygotowanie projektu do użycia ASP.NET Core Identity .....	1188
Przygotowanie bazy danych ASP.NET Core Identity .....	1188
Konfigurowanie ciągu tekstowego połączenia z bazą danych .....	1189
Konfigurowanie aplikacji .....	1189
Tworzenie migracji bazy danych Identity i jej przeprowadzenie .....	1191
Tworzenie narzędzi przeznaczonych do zarządzania użytkownikami .....	1191
Przygotowanie narzędzi przeznaczonych do zarządzania użytkownikami .....	1192
Wyświetlanie listy kont użytkowników .....	1193
Tworzenie użytkowników .....	1195
Edytowanie użytkowników .....	1204
Usunięcie użytkownika .....	1206
Tworzenie narzędzi przeznaczonych do zarządzania rolami .....	1207
Przygotowanie do utworzenia narzędzi przeznaczonych do zarządzania rolami .....	1209
Wyświetlanie i usuwanie ról .....	1209
Tworzenie roli .....	1211
Przypisywanie użytkownika do roli .....	1212
Podsumowanie .....	1215
<b>Rozdział 39. Stosowanie ASP.NET Core Identity .....</b>	<b>1216</b>
Utworzenie przykładowego projektu .....	1216
Uwierzytelnianie użytkowników .....	1217
Tworzenie funkcjonalności logowania .....	1217
Analiza ciasteczka ASP.NET Core Identity .....	1221
Tworzenie strony wylogowania .....	1222
Testowanie funkcjonalności uwierzytelniania .....	1223
Włączanie oprogramowania pośredniczącego odpowiedzialnego za uwierzytelnianie w ASP.NET Core Identity .....	1223
Autoryzacja dostępu do punktu końcowego .....	1227
Zastosowanie atrybutu Authorize .....	1227
Włączenie oprogramowania pośredniczącego odpowiedzialnego za obsługę uwierzytelniania .....	1228
Utworzenie punktu końcowego informującego o braku dostępu .....	1228
Przygotowanie danych .....	1229
Testowanie sekwencji uwierzytelniania .....	1231

Autoryzacja dostępu do aplikacji Blazor .....	1233
Uwierzytelnianie w komponentach Blazor .....	1234
Wyświetlanie treści uwierzytelnionym użytkownikom .....	1236
Uwierzytelnianie i autoryzowanie usług sieciowych .....	1238
Tworzenie prostego klienta JavaScript .....	1241
Ograniczanie dostępu do usługi sieciowej .....	1243
Używanie ciasteczka uwierzytelniania .....	1244
Używanie tokenu bearer .....	1247
Przygotowanie aplikacji .....	1247
Tworzenie tokenów .....	1248
Uwierzytelnianie za pomocą tokenu .....	1250
Ograniczanie dostępu za pomocą tokenów .....	1253
Używanie tokenów do żądania danych .....	1253
Podsumowanie .....	1255

## ROZDZIAŁ 7.



# SportsStore — kompletna aplikacja

W poprzednich rozdziałach zbudowałeś już pierwsze proste aplikacje ASP.NET Core. Zapoznałeś się ze wzorcami ASP.NET Core. Przedstawiłem najważniejsze funkcje C# oraz narzędzia wykorzystywane przez dobrych programistów ASP.NET Core. Teraz czas połączyć to wszystko i zbudować kompletną i realistyczną aplikację typu e-commerce.

Nasza aplikacja, SportsStore, będzie realizowała klasyczny projekt sklepu internetowego: będzie ona zawierać katalog produktów, który można przeglądać według kategorii, koszyk, do którego użytkownik może dodawać produkty i usuwać je, jak również stronę realizującą funkcje kasy, gdzie można też wprowadzić informacje dotyczące wysyłki. Utworzymy ponadto moduł administracyjny, który będzie realizował funkcje tworzenia, przeglądania, aktualizacji i usuwania (CRUD) pozwalające na zarządzanie katalogiem — będzie on chroniony, dzięki czemu tylko zalogowani administratorzy będą mogli wprowadzać zmiany.

Budowana aplikacja nie będzie tylko powierzchowną demonstracją. Zamierzam zbudować solidną i realistyczną aplikację, która korzysta z zalecanych obecnie najlepszych praktyk. Ponieważ chcę się skoncentrować na frameworku ASP.NET Core, konieczne okazało się uproszczenie integracji z systemami zewnętrznymi (na przykład bazą danych) oraz całkowite pominięcie innych (na przykład przetwarzanie płatności za dokonane zakupy).

Zauważysz, że dosyć powoli będziemy budować potrzebne nam poziomy infrastruktury. Jednak początkowa inwestycja w aplikację ASP.NET Core zwraca się nieco później, ponieważ aplikacja ta jest łatwa w utrzymaniu, jest rozszerzalna, uporządkowana i świetnie obsługuje testy jednostkowe.

### Testy jednostkowe

Sporo napisałem na temat łatwości wykonywania testów jednostkowych w ASP.NET Core oraz na temat mojego przekonania, że stosowanie tego rodzaju testów jest ważną częścią procesu tworzenia aplikacji. Przekonanie to będzie się przejawiać w całej książce, ponieważ będę opisywać szczegóły technik stosowanych w testach jednostkowych, powiązanych z kluczowymi funkcjami ASP.NET Core.

Wiem jednak, że nie jest to powszechne przeświadczenie. Jeżeli nie chcesz tworzyć testów jednostkowych, jest to Twoja decyzja. Zatem gdy będę pisać wyłącznie o testach jednostkowych, tekst będzie umieszczony w tego rodzaju ramce. Jeżeli nie jesteś zainteresowany tym tematem, po prostu pomiń ją — nie wpłynie to na samą aplikację SportsStore. Nie musisz wykonywać żadnej formy testowania automatycznego, aby skorzystać z większości udogodnień ASP.NET Core. Oczywiście obsługa testów jednostkowych to jeden z kluczowych powodów, dla których framework ASP.NET Core zyskuje coraz większą popularność.

Większości funkcji ASP.NET Core, z jakich będę korzystał podczas budowy aplikacji SportsStore, poświęciłem osobne rozdziały w dalszej części książki. Zamiast powielać potrzebne informacje, przedstawiam tyle, ile jest niezbędne w danym momencie, i wskażę rozdział zawierający dokładny opis.

Opisuję wszystkie kroki niezbędne przy budowaniu aplikacji, dzięki czemu będziesz widział, jak łączą się ze sobą poszczególne elementy ASP.NET Core. Szczególnie powinieneś zwrócić uwagę na tworzenie widoków. Jeżeli nie będziesz się ściśle stosował do przedstawianych poleceń, aplikacja może się dziwnie zachowywać.

## Utworzenie projektów

Zamierzam rozpocząć od minimalnego projektu ASP.NET Core i dodawać tylko niezbędną funkcjonalność. Przejdź do wiersza poleceń Windows PowerShell i wydaj polecenia przedstawione na listingu 7.1.

- 
- **Wskazówka** Przykładowy projekt dla tego rozdziału, a także projekty dla pozostałych rozdziałów książki zostały umieszczone w archiwum dostępnym pod adresem <https://ftp.helion.pl/przyklady/aspnm8.zip>. Jeżeli napotkasz trudności podczas pracy nad tym przykładem, przydatne w tej sytuacji informacje znajdziesz w rozdziale 1.
- 

### *Listing 7.1. Utworzenie projektu aplikacji SportsStore*

---

```
PS C:\> dotnet new globaljson --sdk-version 3.1.101 --output SportsSln/SportsStore
PS C:\> dotnet new web --no-https --output SportsSln/SportsStore --framework netcoreapp3.1
PS C:\> dotnet new sln -o SportsSln
```

```
PS C:\> dotnet sln SportsSln add SportsSln/SportsStore
```

---

Te polecenia powodują utworzenie katalogu rozwiązania *SportsSln* zawierającego katalog nowego projektu o nazwie *SportsStore*, bazującego na szablonie *web*. Katalog *SportsSln* zawiera również plik rozwiązania, do którego został dodany projekt *SportsStore*.

Zdecydowałem się na użycie odmiennych nazw dla katalogów rozwiązania i projektu, aby przykłady były łatwiejsze do wykonywania. Jeżeli projekt będziesz tworzyć za pomocą Visual Studio, domyślnie zostanie użyta ta sama nazwa dla obu wymienionych katalogów. Nie istnieje tzw. „prawidłowe” podejście w tym zakresie; projektom możesz nadawać dowolne nazwy.

## Utworzenie projektu testów jednostkowych

W celu utworzenia projektu testów jednostkowych wydaj z poziomu katalogu użytego do wydania poleceń przedstawionych na listingu 7.1 polecenia zamieszczone na listingu 7.2.

### *Listing 7.2. Utworzenie projektu testów jednostkowych*

---

```
PS C:\> dotnet new xunit -o SportsSln/SportsStore.Tests --framework netcoreapp3.1
PS C:\> dotnet sln SportsSln add SportsSln/SportsStore.Tests
PS C:\> dotnet add SportsSln/SportsStore.Tests reference SportsSln/SportsStore
```

---

Do tworzenia obiektów imitacji wykorzystam pakiet Moq. Wydanie polecenia przedstawionego na listingu 7.3 spowoduje dodanie wymienionego pakietu do projektu testów jednostkowych. To polecenie powinno zostać wydane z poziomu tego samego katalogu, który był użyty dla poleceń przedstawionych na listingach 7.1 i 7.2.

### Listing 7.3. Instalacja pakietu Moq

```
PS C:\> dotnet add SportsSln/SportsStore.Tests package Moq --version 4.13.1
```

## Utworzenie katalogów projektu aplikacji

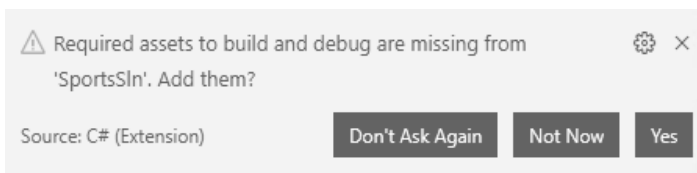
Kolejnym krokiem jest dodanie katalogów przeznaczonych dla komponentów wymaganych podczas tworzenia aplikacji ASP.NET Core. Dla każdego katalogu wymienionego w tabeli 7.1 prawym przyciskiem myszy kliknij projekt *SportsStore* w oknie *Eksplorator rozwiązań*, wybierz opcję *Dodaj/Nowy katalog...* z menu kontekstowego, a następnie podaj nazwę dla nowego katalogu.

Tabela 7.1. Katalogi wymagane przez projekt *SportsStore*

Nazwa	Opis
<i>Models</i>	Ten katalog będzie zawierał dane modelu i klasy zapewniające dostęp do informacji przechowywanych w bazie danych aplikacji.
<i>Controllers</i>	Ten katalog będzie zawierał klasy kontrolera obsługujące żądania HTTP.
<i>Views</i>	Ten katalog będzie zawierał wszystkie pliki widoków Razor, pogrupowane w oddzielnych podkatalogach.
<i>Views/Home</i>	Ten katalog będzie zawierał pliki widoków Razor przeznaczone dla kontrolera Home, które zostaną utworzone w sekcji „Utworzenie kontrolera i widoku”.
<i>Views/Shared</i>	Ten katalog będzie zawierał pliki widoków Razor, które są wspólne dla wszystkich kontrolerów.

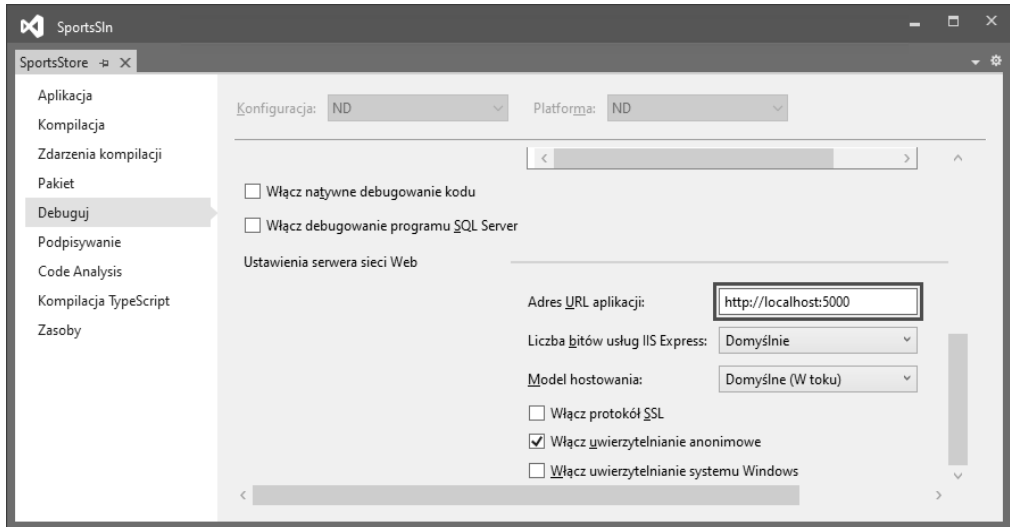
## Otworzenie projektów

Jeżeli używasz Visual Studio Code, wybierz opcję menu *File/Open Folder*, przejdź do katalogu *SportsSln*, a następnie kliknij przycisk *Wybierz folder*. Visual Studio Code utworzy projekt oraz wykryje pliki rozwiązania i projektów. Gdy na ekranie pojawi się okno dialogowe pokazane na rysunku 7.1, kliknij przycisk *Yes*, aby zainstalować zasoby niezbędne do kompilacji projektów. Gdy Visual Studio Code poprosi o wskazanie projektu do uruchomienia, wybierz *SportsStore*.



Rysunek 7.1. Dodanie brakujących zasobów w Visual Studio Code

Natomiast jeżeli używasz Visual Studio, kliknij przycisk *Otwórz projekt lub rozwiązanie* w winietce programu lub wybierz opcję menu *Plik/Otwórz/Projekt/Rozwiązanie...* W wyświetlonym oknie dialogowym zaznacz katalog *SportsSln* i kliknij przycisk *Otwórz* w celu otworzenia projektu. Gdy projekt jest otworzony w Visual Studio, wybierz opcję menu *Projekt/Właściwości SportsStore*, przejdź do sekcji *Debuguj*, a następnie w polu *Adres URL aplikacji* zmień numer portu na 5000, jak pokazałem na rysunku 7.2. Wybierz opcję menu *Plik/Zapisz wszystko*, aby w ten sposób zapisać wprowadzoną zmianę.



**Rysunek 7.2.** Określenie numeru portu HTTP aplikacji w Visual Studio

## Przygotowanie usług aplikacji i potoku żądania

Klasa *Startup* jest odpowiedzialna za konfigurację aplikacji ASP.NET Core. Na listingu 7.4 przedstawiłem zmiany konieczne do wprowadzenia w klasie *Startup*, aby skonfigurować obsługę pewnych funkcji przydatnych podczas tworzenia aplikacji *SportsStore*.

- 
- **Uwaga** Klasa *Startup* ma bardzo ważne znaczenie na platformie ASP.NET Core. Więcej informacji na temat tej klasy przedstawię w rozdziale 12.
- 

### **Listing 7.4.** Konfiguracja aplikacji w pliku *Startup.cs* w projekcie *SportsStore*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```



```

namespace SportsStore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
            });
        }
    }
}

```

Metoda `ConfigureServices()` jest używana w celu skonfigurowania obiektów określanych mianem *usług*, które będą mogły być używane w aplikacji za pomocą mechanizmu wstrzykiwania zależności, którym dokładnie zajmę się w rozdziale 14. Metoda `AddControllersWithViews()` wywoływana w wymienionej wcześniej `ConfigureServices()` jest odpowiada za skonfigurowanie obiektów współdzielonych wymaganych w aplikacjach używających frameworka MVC i silnika widoku Razor.

Framework ASP.NET Core otrzymuje żądania HTTP i przekazuje je do *potoku żądania*, wypełnionego komponentami oprogramowania pośredniczącego, które zostały zarejestrowane w metodzie `Configure()`. Poszczególne komponenty oprogramowania pośredniczącego mają możliwość analizy żądania, jego modyfikacji, wygenerowania odpowiedzi lub modyfikacji odpowiedzi wygenerowanej przez inne komponenty. Potok żądania stanowi sedno frameworka ASP.NET Core i zostanie dokładnie omówiony w rozdziale 12., w którym wyjaśnię również proces samodzielnego tworzenia komponentów oprogramowania pośredniczącego. W tabeli 7.2 wymieniałem metody użyte do skonfigurowania komponentów oprogramowania pośredniczącego, które zostało użyte w kodzie przedstawionym na listingu 7.4.

Szczególnie ważnym komponentem oprogramowania pośredniczącego jest ten dostarczający funkcjonalność routingu punktów końcowych. Odpowiada za dopasowanie żądań HTTP do funkcji aplikacji — nazywanych punktami końcowymi — i pozwala na wygenerowanie odpowiedzi udzielanych na te żądania. Wymieniony proces zostanie dokładnie omówiony w rozdziale 13. Funkcjonalność routingu punktów końcowych została dodana do potoku żądania za pomocą metod `UseRouting()` i `UseEndpoints()`. W celu zarejestrowania frameworka MVC jako źródła punktów końcowych w kodzie przedstawionym na listingu znajduje się wywołanie metody `MapDefaultControllerRoute()`.

## Konfiguracja silnika widoku Razor

Silnik widoku Razor jest odpowiedzialny za przetwarzanie plików widoku (mają rozszerzenie `.cshtml`) w celu wygenerowania odpowiedzi HTML. Konieczne jest przeprowadzenie pewnych początkowych operacji, aby skonfigurować Razor i tym samym ułatwić sobie tworzenie widoków w aplikacji.

**Tabela 7.2.** Metody oprogramowania pośredniczącego użyte w kodzie na listingu 7.4

Metoda	Opis
<code>UseDeveloperExceptionHandler()</code>	Metoda rozszerzenia wyświetlająca informacje szczegółowe dotyczące wyjątku zgłoszonego w aplikacji. Takie rozwiązanie okazuje się użyteczne na etapie pracy nad aplikacją, jak to dokładnie wyjaśnię w rozdziale 16. Ta metoda nie powinna być używana we wdrożonej aplikacji. W rozdziale 11. dowiesz się, jak można ją wyłączyć podczas przygotowywania aplikacji do wdrożenia.
<code>UseStatusCodePages()</code>	Metoda rozszerzenia dodająca prosty komunikat do odpowiedzi HTTP, która w przeciwnym razie w ogóle byłaby pozbawiona treści. Przykładem jest tutaj odpowiedź wraz z kodem stanu 404, czyli informującym o nieznalezieniu żądanego zasobu. Tę funkcjonalność dokładniej omówię w rozdziale 16.
<code>UseStaticFiles()</code>	Ta metoda rozszerzenia włącza obsługę treści statycznej znajdującej się w katalogu <code>wwwroot</code> . Kwestie związane z obsługą treści statycznej zostaną dokładniej omówione w rozdziale 15.

Do katalogu `Views` dodaj plik typu *Importy widoku Razor* o nazwie `_ViewImports.cshtml` i zawartości przedstawionej na listingu 7.5.

- **Ostrzeżenie** Zwróć uwagę na zawartość tego pliku. Bardzo łatwo popełnić błąd, którego skutkiem będzie wygenerowanie przez aplikację nieprawidłowej treści HTML.

**Listing 7.5.** Zawartość pliku `_ViewImports.cshtml` w katalogu `SportsStore/Views`

```
@using SportsStore.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Polecenie `@using` pozwala na użycie w widokach typów z przestrzeni nazw `SportsStore.Models` bez konieczności odwoływania się do niej. Polecenie `@addTagHelper` włącza wbudowane atrybuty pomocnicze znaczników, które później wykorzystamy do utworzenia elementów HTML odzwierciedlających konfigurację aplikacji `SportsStore`.

Do katalogu `SportsStore/Views` dodaj plik typu *Widok początkowy Razor* o nazwie `_ViewStart.cshtml` i zawartości przedstawionej na listingu 7.6. (Jeżeli plik utworzysz na podstawie szablonu elementu w Visual Studio, ten plik będzie zawierał już wyrażenie zamieszczone na listingu).

**Listing 7.6.** Zawartość pliku `_ViewStart.cshtml` w katalogu `SportsStore/Views`

```
@{
    Layout = "_Layout";
}
```

Plik widoku początkowego nakazuje silnikowi Razor użycie pliku układu w generowanym kodzie HTML. To pozwala zmniejszyć ilość kodu powtarzającego się w widokach. Aby utworzyć widok, do katalogu `Views/Shared` dodaj nowy plik typu *Widok Razor* — *pusty* o nazwie `_Layout.cshtml` i zawartości przedstawionej na listingu 7.7.

**Listing 7.7.** Zawartość pliku `_Layout.cshtml` w katalogu `SportsStore/Views`

```

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>

```

Ten plik definiuje prosty dokument HTML, do którego zawartość innych widoków będzie wstawiana za pomocą wyrażenia `@RenderBody`. Dokładne omówienie sposobu działania wyrażenia `Razor` znajdziesz w rozdziale 21.

## Utworzenie kontrolera i widoku

Do katalogu `SportsStore/Controllers` dodaj plik klasy o nazwie `HomeController.cs` i użyj go do zdefiniowania klasy przedstawionej na listingu 7.8. To jest minimalna wersja kontrolera o funkcjonalności wystarczającej do wygenerowania odpowiedzi.

**Listing 7.8.** Zawartość pliku `HomeController.cs` w katalogu `SportsStore/Controllers`

```

using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers
{
  public class HomeController : Controller
  {
    public IActionResult Index() => View();
  }
}

```

Metoda `MapDefaultControllerRoute()` użyta na listingu 7.4 wskazuje ASP.NET Core sposób dopasowania adresów URL do klas kontrolerów. Konfiguracja stosowana przez tę metodę deklaruje, że do obsługi żądań będzie użyta metoda akcji zdefiniowana przez kontroler `Home`.

Metoda akcji `Index()` w tym momencie nie wykonuje żadnych użytecznych zadań, a jedynie zwraca wynik wywołania metody `View()` dziedziczonej po klasie bazowej `Controller`. Ten wynik nakazuje ASP.NET Core wyświetlenie widoku domyślnie powiązanego z daną metodą akcji. W celu utworzenia widoku do katalogu `Views/Home` dodaj plik typu *Widok Razor* — *пустy* o nazwie `Index.cshtml` i zawartości przedstawionej na listingu 7.9.

**Listing 7.9.** Zawartość pliku `Index.cshtml` w katalogu `SportsStore/Views/Home`

```

<h4>Witamy w sklepie SportsStore</h4>

```

## Tworzenie modelu danych

Praktycznie wszystkie projekty ASP.NET Core mają pewnego rodzaju model danych. Ponieważ tworzymy aplikację typu e-commerce, najbardziej oczywistym modelem domeny jest produkt. Do katalogu *Models* dodaj plik klasy o nazwie *Product.cs*, a następnie zdefiniuj w nim klasę przedstawioną na listingu 7.10.

### **Listing 7.10.** Zawartość pliku *Product.cs* w katalogu *SportsStore/Models*

```
using System.ComponentModel.DataAnnotations.Schema;

namespace SportsStore.Models
{
    public class Product
    {
        public long ProductID { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }

        public string Category { get; set; }
    }
}
```

Właściwość *Price* została udekorowana atrybutem *Column* określającym typ danych SQL, który będzie używany do przechowywania wartości dla tej właściwości. Nie wszystkie typy C# można elegancko mapować na typy SQL. Dodanie takiego atrybutu gwarantuje, że baza danych użyje odpowiedniego typu dla danych aplikacji.

## Sprawdzenie i uruchomienie aplikacji

Zanim przejdziemy dalej, dobrze jest upewnić się, że aplikacja w dotychczasowej postaci jest prawidłowo kompilowana i uruchamiana. Wybierz opcję menu *Debugowanie/Uruchom bez debugowania* lub z poziomu katalogu *SportsStore* wydaj polecenie przedstawione na listingu 7.11.

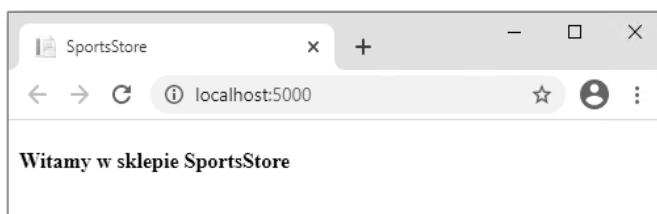
### **Listing 7.11.** Uruchomienie przykładowej aplikacji z poziomu wiersza poleceń

---

```
PS C:\> dotnet run
```

---

Po ponownym wykonaniu żądania pod adresem *http://localhost:5000* w przeglądarce WWW zostanie wygenerowana treść pokazana na rysunku 7.3.



**Rysunek 7.3.** Wynik uruchomienia budowanej aplikacji

## Dodanie danych do aplikacji

W tym momencie aplikacja SportsStore ma już podstawową konfigurację i może udzielać prostej odpowiedzi. Nadszedł więc odpowiedni moment na dodanie pewnych danych do aplikacji, aby mogła wyświetlać znacznie bardziej użyteczne informacje. Jako bazy danych użyjemy SQL Server LocalDB. Będziemy z niej korzystać za pośrednictwem Entity Framework Core (EF Core), czyli opracowanego przez Microsoft frameworka ORM dla .NET. Framework ORM pozwala nam pracować na tabelach, kolumnach i wierszach relacyjnej bazy danych z użyciem zwykłych obiektów C#.

---

■ **Ostrzeżenie** Jeżeli podczas przygotowywania środowiska pracy w rozdziale 2. nie wybrałeś opcji *LocalDB*, będziesz musiał zrobić to teraz. Budowana tutaj aplikacja SportsStore nie będzie funkcjonowała bez bazy danych.

---

## Instalowanie pakietów narzędzi Entity Framework Core

Pierwszym krokiem jest dodanie Entity Framework Core do projektu. Wiersz poleceń Windows PowerShell wykorzystaj do wydania z poziomu katalogu *SportsStore* poleceń przedstawionych na listingu 7.12.

**Listing 7.12.** Dodanie pakietów Entity Framework Core do projektu SportsStore

---

```
PS C:\> dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
PS C:\> dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

---

Te pakiety powodują zainstalowanie Entity Framework Core i pozwolą na użycie SQL Server w projekcie. Entity Framework Core wymaga również pakietu narzędziowego, który obejmuje narzędzia powłoki konieczne do przygotowania i utworzenia aplikacji ASP.NET Core. Wyдай polecenia zamieszczone na listingu 7.13 w celu usunięcia istniejącej wersji pakietu narzędziowego, o ile taka rzeczywiście istnieje, i zainstalowania wersji, z której będziemy korzystać w książce. (Skoro pakiet jest instalowany globalnie, wymienione polecenia można wydać z poziomu dowolnego katalogu).

**Listing 7.13.** Instalacja pakietu narzędziowego Entity Framework Core

---

```
PS C:\> dotnet tool uninstall --global dotnet-ef
PS C:\> dotnet tool install --global dotnet-ef --version 3.1.1
```

---

## Definiowanie ciągu tekstowego połączenia

Ustawienia konfiguracyjne, np. *ciąg tekstowy połączenia*, są przechowywane w plikach JSON. W celu zdefiniowania połączenia z bazą danych, która będzie używana do przechowywania danych aplikacji SportsStore, do pliku *appsettings.json* w katalogu *SportsStore* dodaj polecenia zamieszczone na listingu 7.14.

**Listing 7.14.** Dodanie ustawień konfiguracyjnych w pliku *appsettings.json* w katalogu SportsStore

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```

    "Microsoft": "Warning",
    "Microsoft.Hosting.Lifetime": "Information"
  }
},
"AllowedHosts": "*",
"ConnectionStrings": {
  "SportsStoreConnection":
"Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;MultipleActiveResultSets=true"
}
}

```

Projekt zawiera również plik *appsettings.Development*. Zostały w nim zdefiniowane ustawienia konfiguracyjne, które są używane jedynie podczas pracy nad aplikacją. Wymieniony plik jest w oknie *Eksplorator rozwiązań* wyświetlany jako zagnieżdżony względem *appsettings.json*, przy czym zawsze pozostaje widoczny w Visual Studio Code. Podczas pracy nad projektem SportsStore używam jedynie pliku *appsettings.json*. Powiązania między dwoma wymienionymi plikami dokładnie omówię w rozdziale 15.

- 
- **Wskazówka** Ciąg tekstowy połączenia musi być podany w postaci pojedynczego, niezłamanego wiersza. To bez problemu można zrobić w edytorze kodu Visual Studio, natomiast jest niemożliwe na stronie książki, stąd dziwna postać tego ciągu tekstowego na listingu 7.14. Podczas definiowania ciągu tekstowego połączenia we własnym projekcie upewnij się, że wartość `SportsStoreConnection` znajduje się w jednym wierszu.
- 

Ta konfiguracja wskazuje bazę danych LocalDB o nazwie SportsStore i włącza obsługę funkcjonalności MARS (ang. *multiple active result set*) wymaganej przez niektóre zapytania do bazy danych, wykonywane przez aplikację SportsStore za pomocą Entity Framework Core.

Uważaj podczas dodawania ustawień konfiguracyjnych. Dane JSON muszą być wyrażone dokładnie w postaci pokazanej na listingu. To oznacza konieczność prawidłowego użycia znaków cytowania dla nazw właściwości i ich wartości. Jeżeli masz z tym trudności, odpowiedni plik znajdziesz w materiałach przygotowanych dla książki.

- 
- **Wskazówka** Każdy serwer bazy danych wymaga innego formatu ciągu tekstowego połączenia. Witryna internetowa <https://www.connectionstrings.com/> będzie pomocna podczas tworzenia niezbędnych ciągów tekstowych połączenia.
- 

## Utworzenie klasy kontekstu bazy danych

Entity Framework Core zapewnia dostęp do bazy danych za pomocą tzw. *klasy kontekstu*. W celu utworzenia klasy kontekstu bazy danych dla aplikacji SportsStore, do katalogu *Models* trzeba dodać nowy plik klasy o nazwie *StoreDbContext.cs*, a następnie zdefiniować w nim klasę przedstawioną na listingu 7.15.

**Listing 7.15.** Zawartość pliku *StoreDbContext.cs* w katalogu *SportsStore/Models*

```

using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models
{
    public class StoreDbContext : DbContext
    {
        public StoreDbContext (DbContextOptions<StoreDbContext>

```

```

        options) : base(options) {}

        public DbSet<Product> Products { get; set; }
    }
}

```

Klasa bazowa `DbContext` zapewnia dostęp do funkcjonalności Entity Framework Core, natomiast właściwość `Products` zapewni dostęp do obiektów typu `Product` w bazie danych. Klasa `ApplicationDbContext` dziedziczy po klasie `DbContext` i dodaje właściwości używane w celu odczytywania oraz zapisywania danych aplikacji. W tym momencie są to jedyne właściwości zapewniające możliwość uzyskania dostępu do obiektów typu `Product`.

## Konfigurowanie Entity Framework Core

Entity Framework Core trzeba skonfigurować i tym samym określić typ bazy danych, z którą będzie nawiązywane połączenie, ciąg tekstowy połączenia oraz klasę kontekstu przedstawiającą dane przechowywane w bazie danych. Zmiany konieczne do wprowadzenia w klasie `Startup` zostały zamieszczone na listingu 7.16.

### *Listing 7.16. Skonfigurowanie Entity Framework Core w pliku `Startup.cs` w katalogu `SportsStore`*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public Startup(IConfiguration config)
        {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews(); services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {

```

```

    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();

    app.UseRouting();
    app.UseEndpoints(endpoints => {
        endpoints.MapDefaultControllerRoute();
    });
}
}
}

```

Interfejs `IConfiguration` zapewnia dostęp do systemu konfiguracji ASP.NET Core obejmującego m.in. zawartość pliku `appsettings.json`. Dokładne omówienie wspomnianego systemu znajdziesz w rozdziale 15. Konstruktor otrzymuje obiekt `IConfiguration` za pomocą jego konstruktora i przypisuje ten obiekt właściwości `Configuration`, która jest używana w celu uzyskania dostępu do ciągu tekstowego połączenia.

Konfiguracja Entity Framework Core odbywa się za pomocą metody `AddDbContext()` rejestrującej klasę kontekstu i konfigurującej związek z bazą danych. Metoda `UseSqlServer()` wskazuje na użycie SQL Server i odczytywanie ciągu tekstowego połączenia za pomocą obiektu `IConfiguration`.

## Tworzenie repozytorium

Następnym krokiem jest utworzenie interfejsu repozytorium i klasy implementacji. Wzorzec repozytorium jest jednym z najczęściej stosowanych i zapewnia spójny sposób na uzyskanie dostępu do funkcjonalności oferowanej przez klasę kontekstu bazy danych. Wprawdzie nie każdy uznaje repozytorium za użyteczne rozwiązanie, ale z mojego doświadczenia wynika, że ten wzorzec pomaga zmniejszyć ilość powielanego kodu i zagwarantować spójny sposób przeprowadzania operacji w bazie danych. W katalogu *Models* utwórz nowy plik interfejsu o nazwie `IStoreRepository.cs`, którego zawartość jest zamieszczona na listingu 7.17.

**Listing 7.17.** Zawartość pliku `IStoreRepository.cs` w katalogu `SportsStore/Models`

```

using System.Linq;

namespace SportsStore.Models
{
    public interface IStoreRepository
    {
        IQueryable<Product> Products { get; }
    }
}

```

W interfejsie tym wykorzystany jest interfejs `IQueryable<T>`, który pozwala na pozyskanie sekwencji obiektów `Product` bez konieczności określania sposobu przechowywania i pobierania danych. Ten interfejs wywodzi się ze znacznie bardziej znanego `IEnumerable<T>` i przedstawia kolekcję obiektów, do których można wykonywać zapytania. Wspomniane obiekty mogą więc być zarządzane na przykład przez bazę danych.

Klasa używająca interfejsu `IStoreRepository` może uzyskać obiekty `Product` bez potrzeby znajomości jakichkolwiek szczegółów ich pochodzenia czy sposobu dostarczenia.



## Poznajemy interfejsy `IEnumerable<T>` i `IQueryable<T>`

Interfejs `IQueryable<T>` jest użyteczny, ponieważ pozwala na efektywne wykonywanie zapytań do kolekcji obiektów. W dalszej części rozdziału zaimplementuję obsługę pobierania z bazy danych podzbioru obiektów `Product`. Dzięki interfejsowi `IQueryable<T>` można pobrać z bazy danych jedynie wymagane obiekty, używając do tego standardowych poleceń LINQ. Nie trzeba przy tym mieć żadnych informacji na temat sposobu przechowywania danych przez serwer bazy danych lub przetwarzania zapytania. Bez interfejsu `IQueryable<T>` konieczne byłoby pobranie wszystkich obiektów `Product` z bazy danych, a następnie pozbycie się niepotrzebnych. Tego rodzaju operacja staje się kosztowna wraz ze wzrostem ilości danych wykorzystywanych przez aplikację. To jest więc powód, dla którego w klasach i repozytoriach związanych z obsługą bazy danych jest zwykle używany interfejs `IQueryable<T>` zamiast `IEnumerable<T>`.

Jednak należy zachować ostrożność podczas pracy z interfejsem `IQueryable<T>`, każde użycie kolekcji obiektów powoduje ponowną analizę zapytania, co oznacza jego ponowne wykonanie do bazy danych. To może zniwelować efektywność, jaką przynosi użycie interfejsu `IQueryable<T>`. Dlatego też w tego rodzaju sytuacjach można za pomocą metod rozszerzenia `ToList()` i `ToArray()` przeprowadzić konwersję `IQueryable<T>` na znacznie bardziej przewidywalną postać.

W celu utworzenia implementacji interfejsu repozytorium do katalogu *Models* należy dodać plik o nazwie *EFStoreRepository.cs* i użyć go do zdefiniowania klasy przedstawionej na listingu 7.18.

**Listing 7.18.** Zawartość pliku *EFStoreRepository.cs* w katalogu *SportsStore/Models*xxx,

```
using System.Linq;

namespace SportsStore.Models {
    public class EFStoreRepository : IStoreRepository {
        private StoreDbContext context;

        public EFStoreRepository(StoreDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.Products;
    }
}
```

Do aplikacji będziemy dodawać kolejną funkcjonalność w miarę budowania kolejnych komponentów. W tym momencie implementacja repozytorium odpowiada po prostu za mapowanie zdefiniowanej przez interfejs `IStoreRepository` właściwości `Products` na właściwość `Product` zdefiniowaną w klasie `StoreDbContext`. Właściwość `Product` w klasie kontekstu zwraca obiekt `DbSet<Product>` implementujący interfejs `IQueryable<T>` i niezwykle ułatwia implementację interfejsu repozytorium podczas pracy z Entity Framework Core.

We wcześniejszej części rozdziału wspomniałem, że ASP.NET Core obsługuje usługi pozwalające na uzyskanie dostępu do obiektów w aplikacji. Jedną z zalet usług jest to, że pozwalają klasom na używanie interfejsów bez konieczności poznania używanej klasy implementacji. Ten mechanizm zostanie dokładnie omówiony w rozdziale 14. W przypadku aplikacji *SportsStore* to oznacza, że komponenty aplikacji mogą uzyskiwać dostęp do obiektów implementujących interfejs `IStoreRepository` i nie muszą przy tym znać używanej implementacji klasy `EFStoreRepository`. Dzięki temu zyskujemy możliwość łatwej zmiany używanej przez aplikację klasy implementacji, bez konieczności wprowadzania zmian w poszczególnych komponentach. Przedstawione na listingu 7.19 polecenia dodaj do klasy

Startup w celu utworzenia usługi dla interfejsu `IStoreRepository`, który jako klasy implementacji używa `EFStoreRepository`.

- 
- **Wskazówka** Nie przejmuj się, jeśli przedstawione tutaj informacje są niezrozumiałe. To jest jedno z najbardziej dezorientujących zagadnień podczas pracy z ASP.NET Core i jego poznanie może wymagać nieco czasu.
- 

**Listing 7.19.** *Utworzenie usługi repozytorium w pliku Startup.cs w katalogu SportsStore*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public Startup(IConfiguration config)
        {
            Configuration = config;
        }
        private IConfiguration Configuration { get; set; }
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
            });
        }
    }
}
```

Metoda `AddScoped()` tworzy usługę, w której każde żądanie HTTP otrzymuje oddzielny obiekt repozytorium. Mamy tutaj typowy sposób, na który zwykle używa się Entity Framework Core.

## Utworzenie i zastosowanie migracji bazy danych

Entity Framework Core ma możliwość wygenerowania schematu bazy danych na podstawie klas modelu za pomocą funkcji o nazwie *migracji*. Podczas tworzenia migracji Entity Framework Core generuje klasę C# zawierającą polecenia SQL niezbędne do przygotowania bazy danych. Jeżeli zachodzi potrzeba modyfikacji klas modelu, wówczas można utworzyć nową migrację zawierającą polecenia SQL odzwierciedlające wprowadzone zmiany. Tym samym nie trzeba się zajmować ręcznym tworzeniem i testowaniem poleceń SQL. Zamiast tego można się całkowicie skoncentrować na klasach C# przedstawiających model aplikacji.

Polecenia EF Core są wykonywane z poziomu wiersza poleceń. Otwórz więc nowe okno wiersza poleceń Windows PowerShell, przejdź do katalogu projektu SportsStore, a następnie wydaj polecenie przedstawione na listingu 7.20 w celu wygenerowania klasy migracji, która przygotowuje bazę danych do jej pierwszego użycia.

**Listing 7.20.** Utworzenie migracji bazy danych

---

```
$ dotnet ef migrations add Initial
```

---

Po wykonaniu powyższego polecenia projekt SportsStore będzie zawierał nowy katalog o nazwie *Migrations*. W tym katalogu Entity Framework Core przechowuje klasy migracji. Jeden z plików będzie miał nazwę w postaci długiego numeru zakończonego członem *\_Initial.cs*. To jest klasa, którą wykorzystamy do utworzenia początkowego schematu dla bazy danych. Jeżeli przeanalizujesz zawartość tego pliku, zobaczysz, jak klasa modelu *Product* została użyta do utworzenia schematu.

### Co się stało z poleceniami Add-Migration i Update-Database?

Jeżeli już wcześniej pracowałeś z Entity Framework Core, zapewne poznałeś polecenie *Add-Migration* używane w celu utworzenia migracji bazy danych i polecenie *Update-Database* służące do jej zastosowania w bazie danych.

Wraz z wprowadzeniem platformy .NET Core, Entity Framework Core zawiera polecenia zintegrowane z narzędziem *dotnet* i wykorzystuje pakiet *Microsoft.EntityFrameworkCore.Tools.DotNet*. Jedno z nowych poleceń wykorzystane w rozdziale, ponieważ są one spójne i mogą być używane w dowolnym oknie powłoki (wiersza poleceń), w przeciwieństwie do poleceń *Add-Migration* i *Update-Database*, które działały jedynie z poziomu konkretnego okna w Visual Studio.

## Tworzenie danych początkowych

W celu wypełnienia bazy danych i dostarczenia pewnych przykładowych danych należy utworzyć w katalogu *Models* nowy plik klasy o nazwie *SeedData.cs*, a następnie zdefiniować w nim klasę przedstawioną na listingu 7.21.

**Listing 7.21.** Zawartość pliku *SeedData.cs* w katalogu *SportsStore/Models*

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
```

```

namespace SportsStore.Models
{
    public static class SeedData
    {
        public static void EnsurePopulated(IApplicationBuilder app)
        {
            StoreDbContext context = app.ApplicationServices
                .CreateScope().ServiceProvider.GetRequiredService<StoreDbContext>();

            if (context.Database.GetPendingMigrations().Any()) {
                context.Database.Migrate();
            }

            if (!context.Products.Any())
            {
                context.Products.AddRange(
                    new Product
                    {
                        Name = "Kajak",
                        Description = "Łódka przeznaczona dla jednej osoby",
                        Category = "Sporty wodne",
                        Price = 275
                    },
                    new Product
                    {
                        Name = "Kamizelka ratunkowa",
                        Description = "Chroni i dodaje uroku",
                        Category = "Sporty wodne",
                        Price = 48.95m
                    },
                    new Product
                    {
                        Name = "Piłka",
                        Description = "Zatwierdzone przez FIFA rozmiar i waga",
                        Category = "Piłka nożna",
                        Price = 19.50m
                    },
                    new Product
                    {
                        Name = "Flagi narożne",
                        Description = "Nadadzą twojemu boisku profesjonalny
                            wygląd",
                        Category = "Piłka nożna",
                        Price = 34.95m
                    },
                    new Product
                    {
                        Name = "Stadion",
                        Description = "Składany stadion na 35 000 osób",
                        Category = "Piłka nożna",
                        Price = 79500
                    },
                    new Product
                    {
                        Name = "Czapka",
                        Description = "Zwiększa efektywność mózgu o 75%",
                        Category = "Szachy",
                        Price = 16
                    },
                    new Product

```

```

    {
        Name = "Niestabilne krzesło",
        Description = "Zmniejsza szanse przeciwnika",
        Category = "Szachy",
        Price = 29.95m
    },
    new Product
    {
        Name = "Ludzka szachownica",
        Description = "Przyjemna gra dla całej rodziny!",
        Category = "Szachy",
        Price = 75
    },
    new Product
    {
        Name = "Błyszczący król",
        Description = "Figura pokryta złotem i wysadzana
            diamentami",
        Category = "Szachy",
        Price = 1200
    }
    };
    context.SaveChanges();
}
}
}
}
}
}

```

Metoda statyczna `EnsurePopulated()` otrzymuje argument `IApplicationBuilder`, który jest klasą używaną w metodzie `Configure()` klasy `Startup` do zarejestrowania klas oprogramowania pośredniczącego w celu obsługi żądań HTTP. `IApplicationBuilder` zapewnia również dostęp do usług aplikacji, m.in. do usługi kontekstu bazy danych `Entity Framework Core`.

Metoda `EnsurePopulated()` pobiera obiekt `StoreDbContext` za pomocą interfejsu `IApplicationBuilder` i wywołuje metodę `Database.Migrate()` w celu zagwarantowania przeprowadzenia migracji. Oznacza to utworzenie bazy danych i jej przygotowanie do przechowywania obiektów typu `Product`. Następnym krokiem jest sprawdzenie, czy w bazie danych znajdują się jakiekolwiek obiekty `Product`. Jeżeli nie ma żadnych obiektów, wówczas baza danych zostanie wypełniona kolekcją obiektów `Product`. Użyta będzie przy tym metoda `AddRange()`, a dane zostaną umieszczone w bazie danych za pomocą metody `SaveChanges()`.

Ostatnia zmiana polega na umieszczeniu w bazie danych niezbędnych danych podczas uruchomienia aplikacji. Odbywa się to za pomocą wywołania metody `SeedData.EnsurePopulated()` w klasie `Startup`, jak pokazałem na listingu 7.22.

**Listing 7.22.** Umieszczenie danych początkowych w bazie danych za pomocą wywołania `EnsurePopulated()` w pliku `Startup.cs` w projekcie `SportsStore`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;

```

```

using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public Startup(IConfiguration config) =>
            Configuration = config;

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();

            services.AddDbContext<StoreDbContext>(opts =>
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]
                );
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
            });
            SeedData.EnsurePopulated(app);
        }
    }
}

```

## Wyzerowanie bazy danych

Jeżeli zachodzi potrzeba wyzerowania bazy danych, wówczas z poziomu katalogu *SportsStore* wydaj następujące polecenie:

```

...
dotnet ef database drop --force --context StoreDbContext
...

```

Po ponownym uruchomieniu ASP.NET Core baza danych zostanie utworzona od nowa z danymi początkowymi.

## Wyświetlanie listy produktów

Jak miałeś okazję zobaczyć, początkowe przygotowania projektu ASP.NET Core zabierają nieco czasu. Dobrą wiadomością jest to, że po przygotowaniu podstaw szybkość pracy wzrasta i kolejne funkcje można dodawać w znacznie krótszym czasie. W podrozdziale tym utworzymy kontroler i metodę akcji pozwalającą wyświetlić dane produktu z repozytorium.

## Użycie szkieletu oferowanego przez Visual Studio

W rozdziale 4. dowiedziałeś się, że Visual Studio pozwala na stosowanie szkieletu podczas dodawania elementów do projektu.

Jednak w tej książce nie będę używał funkcji szkieletu. Kod wygenerowany przez szkielet jest zbyt ogólny, aby mógł zostać uznany za użyteczny, a sam zestaw dostępnych scenariuszy zbyt skąpy i nie pozwala na rozwiązanie najczęściej napotykaných problemów programistycznych. Moim celem w książce jest nie tylko upewnienie się, że potrafisz utworzyć aplikację opartą na frameworku ASP.NET Core, ale również dokładne wyjaśnienie sposobu jej działania. Ten cel stanie się trudniejszy do zrealizowania, gdy odpowiedzialność za tworzenie komponentów zostanie zrzuczona na Visual Studio.

Jeżeli używasz Visual Studio, prawym przyciskiem myszy kliknij katalog w oknie *Eksplorator rozwiązań* i wybierz opcję *Dodaj/Nowy element...* z menu kontekstowego, a następnie wskaż odpowiedni szablon elementu w wyświetlonym oknie dialogowym *Dodaj nowy element*.

Mając to na uwadze, mamy kolejną sytuację, w której Twój styl programowania jest inny od mojego. Być może będziesz preferować pracę ze szkieletem generowanym przez Visual Studio. Wprowadzie tego rodzaju podejście wydaje się rozsądne, ale zachęcam Cię do poświęcenia czasu na poznanie sposobu działania generowania szkieletu. Dzięki temu będziesz wiedział, gdzie zajrzeć, jeśli otrzymasz wyniki inne niż oczekiwane.

## Przygotowanie kontrolera

Polecenia przedstawione na listingu 7.23 przygotowują kontroler do wyświetlenia listy produktów.

**Listing 7.23.** Przygotowanie kontrolera w pliku *HomeController.cs* w katalogu *SportsStore/Controllers*

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers
{
    public class HomeController : Controller
    {
        private IStoreRepository repository;

        public HomeController(IStoreRepository repo)
        {
            repository = repo;
        }

        public IActionResult Index() => View(repository.Products);
    }
}
```

Kiedy framework ASP.NET Core musi utworzyć nowy egzemplarz klasy *HomeController* w celu obsługi żądania HTTP, przeprowadza analizę konstruktora i ustala, że wymagany jest obiekt implementujący interfejs *IStoreRepository*. W celu określenia, która implementacja klasy powinna zostać użyta, ASP.NET Core sprawdzi konfigurację w klasie *Startup*. Z konfiguracji wynika, że ma być użyta klasa *EFStoreRepository*, a nowy egzemplarz powinien być tworzony za każdym razem. Framework ASP.NET Core tworzy nowy obiekt typu *EFStoreRepository* i wykorzystuje go do wywołania konstruktora *HomeController* w celu utworzenia obiektu kontrolera odpowiedzialnego za przetworzenie żądania HTTP.

To nosi nazwę *wstrzyknięcia zależności* (ang. *dependency injection*). Ten mechanizm pozwala obiektowi typu `HomeController` na uzyskanie dostępu do repozytorium aplikacji za pomocą interfejsu `IStoreRepository` bez konieczności ustalenia, która implementacja klasy została skonfigurowana. Na późniejszym etapie pracy nad aplikacją usługę można zastąpić inną klasą implementacji — na przykład nieużywającą `Entity Framework Core` — a mechanizm wstrzykiwania zależności gwarantuje, że kontroler będzie nadal działał, bez konieczności wprowadzania w nim jakichkolwiek zmian.

- 
- **Uwaga** Część programistów nie lubi wstrzykiwania zależności, uważając, że ten mechanizm niepotrzebnie komplikuje aplikację. Nie podzielam tej opinii. Jeżeli mechanizm wstrzykiwania zależności jest dla Ciebie nowością, więcej informacji na jego temat znajdziesz w rozdziale 14.
- 

### Test jednostkowy — uzyskanie dostępu do repozytorium

Test jednostkowy pozwalający na sprawdzenie, czy kontroler poprawnie uzyskuje dostęp do repozytorium, można przeprowadzić przez utworzenie imitacji repozytorium, wstrzyknięcie jej do konstruktora klasy `HomeController`, a następnie wywołanie metody akcji `Index()` w celu otrzymania odpowiedzi zawierającej listę produktów. Kolejnym krokiem jest porównanie otrzymanych obiektów `Product` z oczekiwanymi z danych testowych w implementacji imitacji. Więcej informacji na temat konfiguracji testów jednostkowych znajdziesz w rozdziale 6. Zapoznaj się z testem jednostkowym utworzonym do wymienionego celu. Ten test został zdefiniowany w pliku o nazwie `HomeControllerTests.cs`, który trzeba dodać do projektu `SportsStore.Tests`.

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;
namespace SportsStore.Tests
{
    public class ProductControllerTests
    {
        [Fact]
        public void Can_Use_Repository()
        {
            // Przygotowanie.
            Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
            mock.Setup(m => m.Products).Returns((new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"}
            }).AsQueryable<Product>());

            HomeController controller = new HomeController(mock.Object);

            // Działanie.
            IEnumerable<Product> result =
                (controller.Index() as ViewResult).ViewData.Model
                as IEnumerable<Product>;

            // Asercja.
```



```

        Product[] prodArray = result.ToArray();
        Assert.True(prodArray.Length == 2);
        Assert.Equal("P1", prodArray[0].Name);
        Assert.Equal("P2", prodArray[1].Name);
    }
}

```

Za dziwne można uznać pobieranie danych zwracanych z metody akcji. Wynikiem jest obiekt typu `ViewResult` i konieczne jest rzutowanie wartości jego właściwości `ViewData.Model` na oczekiwany typ danych. W części drugiej książki dokładnie omówię poszczególne typy wyniku (i sposoby ich działania), które mogą być zwracane przez metody akcji.

## Uaktualnienie widoku

Przedstawiona na listingu 7.23 metoda akcji `Index()` przekazuje kolekcję obiektów `Product` z repozytorium do metody `View()`. To oznacza, że wymienione obiekty będą modelem widoku, który silnik Razor wykorzysta podczas generowania treści HTML dla widoku. Wprowadź zmiany zamieszczone na listingu 7.24, aby w ten sposób móc generować treść na podstawie obiektów `Product` modelu widoku.

**Listing 7.24.** Użycie danych produktu w pliku `Index.cshtml` w katalogu `SportsStore/Views/Home`

```

@model IQueryable<Product>

@foreach (var p in Model)
{
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

```

Wyrażenie `@model` znajdujące się na początku pliku wskazuje, że widok otrzyma z metody akcji sekwencję obiektów `Product`, które będą jego danymi modelu. Wykorzystałem wyrażenie `@foreach` do przeprowadzenia iteracji przez tę sekwencję i wygenerowania prostego zbioru elementów HTML dla każdego otrzymanego obiektu `Product`.

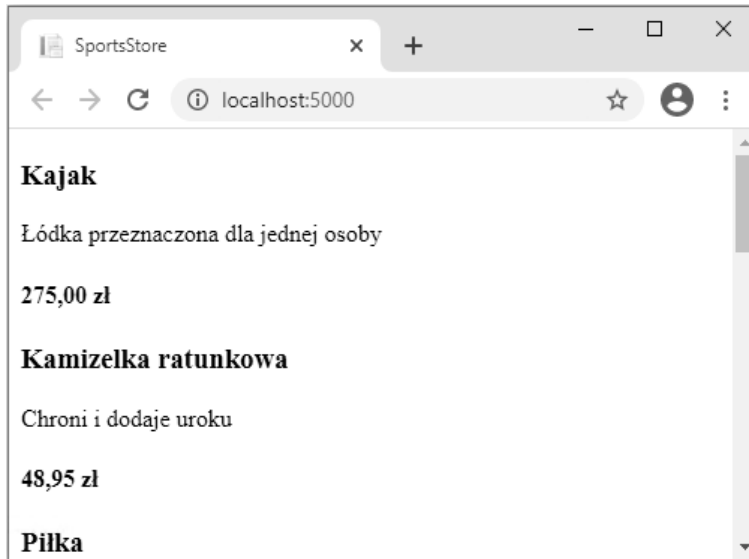
Widok nie musi wiedzieć, skąd pochodzą obiekty `Product`, w jaki sposób są pobierane, a także tego, czy przedstawiają wszystkie produkty znane aplikacji. Zamiast tego widok zajmuje się jedynie szczegółami dotyczącymi wyświetlenia poszczególnych obiektów `Product` za pomocą elementów HTML.

- **Wskazówka** W przedstawionym tu widoku do konwersji właściwości `Price` na postać ciągu tekstowego wykorzystana jest metoda formatująca `.ToString("c")`, która zwraca wartość numeryczną jako zapis waluty zgodny z ustawieniami regionalnymi serwera. Jeżeli serwer jest skonfigurowany na przykład jako pl-PL, to wywołanie `(1002.3).ToString("c")` zwróci 1 002,30 zł, a jeżeli jako en-US, to zwróci \$1,002.30.

## Uruchamianie aplikacji

Uruchom ASP.NET Core i wykonaj żądanie pod adresem `http://localhost:5000`, aby otrzymać listę produktów, jak pokazałem na rysunku 7.4. To jest typowy wzorzec stosowany w ASP.NET Core.

Na początku trzeba poświęcić nieco czasu na skonfigurowanie infrastruktury aplikacji, aby później można było szybko dodawać do niej kolejne funkcje.



Rysunek 7.4. Wyświetlenie listy produktów

## Dodanie stronicowania

Jak widać na rysunku 7.4, wszystkie dane produktów pobrane z bazy danych są wyświetlane na jednej stronie wygenerowanej na podstawie widoku *Index.cshtml*. W tym podrozdziale dodamy obsługę stronicowania, dzięki czemu będziemy mogli wyświetlić określoną liczbę produktów na stronie, a użytkownik będzie mógł przechodzić pomiędzy stronami, aby przejrzeć cały katalog. Aby zaimplementować tego rodzaju rozwiązanie, dodamy parametr do metody *Index()* w kontrolerze *HomeController* (listing 7.25).

Listing 7.25. Dodawanie stronicowania w pliku *HomeController.cs* w katalogu *SportsStore/Controllers*

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers
{
    public class HomeController : Controller
    {
        private IStoreRepository repository;
        public int PageSize = 4;

        public HomeController(IStoreRepository repo)
        {
            repository = repo;
        }
    }
}
```

```

public IActionResult Index(int productPage = 1)
    => View(repository.Products
        .OrderBy(p => p.ProductID)
        .Skip((productPage - 1) * PageSize)
        .Take(PageSize));
    }
}

```

Właściwość `PageSize` pozwala zdefiniować, że chcemy widzieć na stronie cztery produkty. Do metody `Index()` dodałem *parametr opcjonalny*. Dzięki temu, gdy wywołamy metodę bez parametru, nasze wywołanie będzie traktowane tak, jakbyśmy podali wartość określoną w definicji parametru. W efekcie metoda akcji powoduje wyświetlenie pierwszej strony produktów, gdy framework ASP.NET Core wywołuje tę metodę bez argumentu. W metodzie `Index()` pobieramy obiekty `Product`, układamy je w kolejności klucza podstawowego, pomijamy produkty znajdujące się przed naszą stroną, a następnie odczytujemy tyle produktów, ile jest wskazywanych przez wartość właściwości `PageSize`.

## Test jednostkowy — stronicowanie

Aby przetestować funkcję stronicowania, utworzymy imitację repozytorium, zażądamy określonej strony z kontrolera, a następnie sprawdzimy, czy otrzymaliśmy oczekiwany podzbiór danych. Poniżej znajduje się test, jaki utworzyłem w pliku klasy o nazwie `HomeControllerTests.cs` w projekcie `SportsStore.Tests`.

```

using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests
{
    public class ProductControllerTests
    {
        [Fact]
        public void Can_Use_Repository()
        {
            // Polecenia zostały pominięte w celu zachowania zwięzłości.
        }

        [Fact]
        public void Can_Paginate()
        {
            // Przygotowanie.
            Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
            mock.Setup(m => m.Products).Returns((new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"},
                new Product {ProductID = 3, Name = "P3"},
                new Product {ProductID = 4, Name = "P4"},
                new Product {ProductID = 5, Name = "P5"}
            })).AsQueryable<Product>();

            HomeController controller = new HomeController(mock.Object);
            controller.PageSize = 3;
        }
    }
}

```

```

// Działanie.
IEnumerable<Product> result =
    (controller.Index(2) as ViewResult).ViewData.Model
        as IEnumerable<Product>;

// Asercje.
Product[] prodArray = result.ToArray();
Assert.True(prodArray.Length == 2);
Assert.Equal("P4", prodArray[0].Name);
Assert.Equal("P5", prodArray[1].Name);
    }
}
}

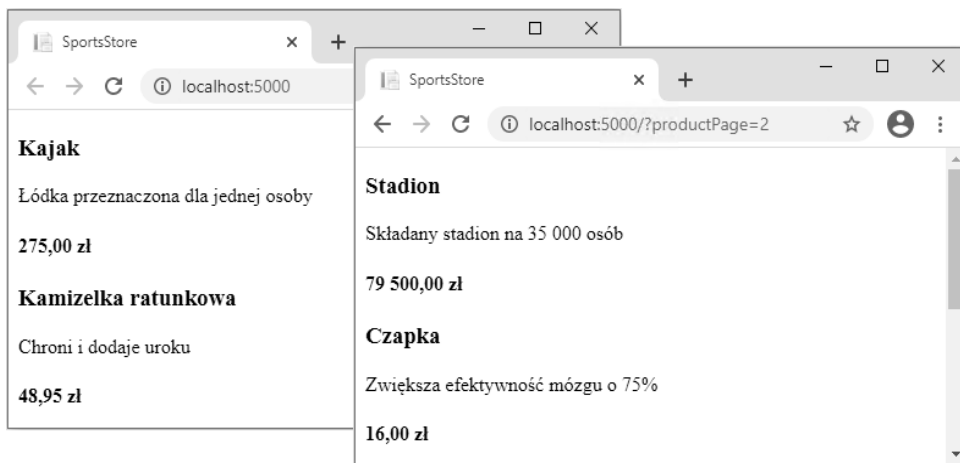
```

Zwróć uwagę, jak nowy test stosuje wzorec użyty podczas definiowania poprzedniego. Działanie opiera się na założeniu, że framework Moq dostarczy znany zestaw danych testowych.

## Wyświetlanie łączy stron

Po ponownym uruchomieniu aplikacji ASP.NET Core i wykonaniu żądania pod adresem *http://localhost:5000* zauważysz tylko cztery produkty na jednej stronie, jak możesz zobaczyć na rysunku 7.5. Jeżeli chcesz zobaczyć inną stronę, możesz dodać do adresu URL parametr:

`http://localhost:5000/?productPage=2`



Rysunek 7.5. Stronicowanie danych

Z wykorzystaniem tego typu ciągów zapytania można przechodzić pomiędzy stronami katalogu produktów. Oczywiście, tylko my wiemy o tym. Klienci nie będą wiedzieć, jakich parametrów ciągu zapytania powinni użyć, a nawet jeżeli udaloby się ich o tym poinformować, nie byłoby zadowoleni z takiego sposobu nawigacji. Niezbędne jest wygenerowanie łączy stron na dole każdej listy produktów, dzięki którym użytkownicy będą mogli przechodzić pomiędzy stronami. W tym celu zaimplementujemy atrybut pomocniczy znacznika, który wygeneruje znaczniki HTML dla potrzebnych łączy nawigacji.

## Dodawanie modelu widoku

Aby zapewnić możliwość prawidłowego działania atrybutów pomocniczych znaczników, będziemy musieli przekazać informacje o liczbie dostępnych stron, bieżącej stronie oraz całkowitej liczbie produktów w repozytorium. Najprostszym sposobem zrealizowania tego zadania jest utworzenie klasy modelu widoku, która będzie używana do przekazywania danych między kontrolerem i widokiem. W projekcie *SportsStore* utwórz katalog *Models/ViewModels*, a następnie umieść w nim plik klasy *PagingInfo.cs* i wykorzystaj go do zdefiniowania klasy przedstawionej na listingu 7.26.

### Listing 7.26. Zawartość pliku *PagingInfo.cs* w katalogu *SportsStore/Models/ViewModels*

```
using System;

namespace SportsStore.Models.ViewModels
{
    public class PagingInfo
    {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages =>
            (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
    }
}
```

## Dodanie klasy atrybutu pomocniczego znacznika

Skoro mamy model widoku, możemy przystąpić do utworzenia klasy atrybutu pomocniczego znacznika. W projekcie *SportsStore* utwórz nowy katalog o nazwie *Infrastructure* i umieść w nim plik klasy *PageLinkTagHelper.cs*, w którym będzie zdefiniowana klasa przedstawiona na listingu 7.27. Atrybuty pomocnicze znaczników to ważny dodatek do frameworka ASP.NET Core. Więcej informacji na temat ich tworzenia i sposobu działania znajdziesz w rozdziałach od 25. do 27.

- 
- **Wskazówka** Katalogu *Infrastructure* używam do przechowywania klas zawierających te komponenty aplikacji, które nie są powiązane z podstawową funkcjonalnością aplikacji. We własnych projektach nie musisz stosować się do tej konwencji.
- 

### Listing 7.27. Zawartość pliku klasy *PageLinkTagHelper.cs* w katalogu *SportsStore/Infrastructure*

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure
{
    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper
    {
        private IUrlHelperFactory urlHelperFactory;
    }
}
```

```

public PageLinkTagHelper(IUrlHelperFactory helperFactory)
{
    urlHelperFactory = helperFactory;
}

[ViewContext]
[HtmlAttributeNotBound]
public ViewContext ViewContext { get; set; }

public PagingInfo PageModel { get; set; }

public string PageAction { get; set; }

public override void Process(TagHelperContext context,
    TagHelperOutput output)
{
    IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
    TagBuilder result = new TagBuilder("div");
    for (int i = 1; i <= PageModel.TotalPages; i++)
    {
        TagBuilder tag = new TagBuilder("a");
        tag.Attributes["href"] = urlHelper.Action(PageAction,
            new { productPage = i });
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
    }
    output.Content.AppendHtml(result.InnerHtml);
}
}
}

```

Ten atrybut pomocniczy znacznika umieszcza w elemencie <div> znaczniki <a> odpowiadające stronom produktów. Nie zamierzam w tym miejscu zagłębiać się w szczegóły dotyczące działania atrybutów pomocniczych znaczników. Wystarczy wiedzieć, że to jeden z najużyteczniejszych sposobów umieszczania logiki C# w widokach. Kod atrybutu pomocniczego znacznika nie jest zbyt czytelny, ponieważ języki C# i HTML nie łączą się zbyt elegancko. Jednak użycie atrybutów pomocniczych znaczników to preferowane podejście w zakresie umieszczania kodu C# w widoku, ponieważ atrybut pomocniczy znacznika może być bardzo łatwo przetestowany za pomocą testu jednostkowego.

Większość komponentów ASP.NET Core, takich jak kontrolery i widoki, może być odkrywanych automatycznie. Natomiast atrybuty pomocnicze znaczników muszą być rejestrowane. Na listingu 7.28 pokazałem nowe polecenie `@addTagHelper` dodane do pliku `_ViewImports.cshtml` w katalogu `Views`. To polecenie nakazuje wyszukanie klas atrybutów pomocniczych znaczników w projekcie `SportsStore`. Dodałem także wyrażenie `@using`, aby w widokach można było odwoływać się do klas modelu bez konieczności podawania w pełni kwalifikowanej nazwy wraz z przestrzenią nazw.

**Listing 7.28.** Zarejestrowanie atrybutu pomocniczego znacznika w pliku `_ViewImports.cshtml` w katalogu `Views`

```

@using SportsStore.Models
@using SportsStore.Models.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, SportsStore

```

## Test jednostkowy — tworzenie łączy stron

Aby przetestować klasę atrybutu pomocniczego znacznika `PageLinkTagHelper`, wywołamy metodę `Process()` wraz z danymi testowymi i dostarczymy obiekt `TagHelperOutput`, który będzie analizowany w celu porównania wyniku z oczekiwanym kodem HTML. W projekcie `SportsStore.Tests` utwórz nowy plik o nazwie `PageLinkTagHelperTests.cs` i umieść w nim przedstawiony poniżej fragment kodu.

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests
{
    public class PageLinkTagHelperTests
    {
        [Fact]
        public void Can_Generate_Page_Links()
        {
            // Przygotowanie.
            var urlHelper = new Mock<IUrlHelper>();
            urlHelper.SetupSequence(x =>
                x.Action(It.IsAny<UrlActionContext>()))
                .Returns("Test/Page1")
                .Returns("Test/Page2")
                .Returns("Test/Page3");

            var urlHelperFactory = new Mock<IUrlHelperFactory>();
            urlHelperFactory.Setup(f =>
                f.GetUrlHelper(It.IsAny<ActionContext>()))
                .Returns(urlHelper.Object);

            PageLinkTagHelper helper =
                new PageLinkTagHelper(urlHelperFactory.Object)
            {
                PageModel = new PagingInfo
                {
                    CurrentPage = 2,
                    TotalItems = 28,
                    ItemsPerPage = 10
                },
                PageAction = "Test"
            };

            TagHelperContext ctx = new TagHelperContext(
                new TagHelperAttributeList(),
                new Dictionary<object, object>(), "");

            var content = new Mock<TagHelperContent>();
            TagHelperOutput output = new TagHelperOutput("div",
                new TagHelperAttributeList(),
                (cache, encoder) => Task.FromResult(content.Object));
```

```

// Działanie.
helper.Process(ctx, output);

// Asercje.
Assert.Equal(@"<a href=""Test/Page1"">1</a>"
+ @"<a href=""Test/Page2"">2</a>"
+ @"<a href=""Test/Page3"">3</a>",
output.Content.GetContent());
}
}
}

```

Trudność w tym teście polega na utworzeniu obiektów niezbędnych do utworzenia i użycia atrybutu pomocniczego znacznika. Tego rodzaju atrybut wykorzystuje obiekty `IUrlHelperFactory` do wygenerowania adresów URL dla różnych fragmentów aplikacji. W tym teście użyłem frameworka `Moq` do przygotowania implementacji tego interfejsu oraz powiązanego interfejsu `IUrlHelper` dostarczającego dane testowe.

Kluczowy fragment testu sprawdza dane wyjściowe atrybutu pomocniczego znacznika za pomocą literału ciągu tekstowego zawierającego cudzysłów. Język `C#` ma duże możliwości w zakresie pracy z tego rodzaju ciągami tekstowymi, o ile ciąg tekstowy będzie poprzedzony znakiem `@` i ujęty w cudzysłów, a nie apostrofy. Pamiętaj, aby nie dzielić literału ciągu tekstowego na oddzielne wiersze, chyba że porównywany ciąg tekstowy również jest podzielony w taki sposób. Na przykład literał użyty w powyższej metodzie testowej został opakowany kilkoma wierszami, ponieważ nie mieści się na stronie drukowanej książki. Nie dodałem znaku nowego wiersza, ponieważ to spowodowałoby niezaliczenie testu.

## Dodawanie danych modelu widoku

Nie jesteśmy w pełni gotowi do użycia naszego atrybutu pomocniczego znacznika. Musimy jeszcze przekazać obiekt klasy `PagingInfo` do widoku. W tym celu dodaj nowy plik klasy, o nazwie `ProductsListViewModel.cs`, do katalogu `Models/ViewModels` w projekcie `SportsStore`. Kod tej klasy przedstawiłem na listingu 7.29.

**Listing 7.29.** Zawartość pliku `ProductsListViewModel.cs` w katalogu `SportsStore/Models/ViewModels`

```

using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels
{
    public class ProductsListViewModel
    {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
}

```

Teraz możemy zaktualizować metodę `Index()` w klasie `HomeController`, aby korzystała z klasy `ProductsListViewModel` do przekazania danych wyświetlanych produktów oraz informacji o stronicowaniu (listing 7.30).

**Listing 7.30.** Aktualizacja metody akcji w pliku `HomeController.cs` w katalogu `SportsStore/Controllers`

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

```



```

using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers
{
    public class HomeController : Controller
    {
        private IStoreRepository repository;
        public int PageSize = 4;

        public HomeController(IStoreRepository repo)
        {
            repository = repo;
        }

        public ViewResult Index(int productPage = 1)
            => View(new ProductsListViewModel
            {
                Products = repository.Products
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo
                {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            });
    }
}

```

Zmiany te powodują przekazanie obiektu `ProductsListViewModel` jako danych modelu dla widoku.

## Test jednostkowy — dane stronicowania w widoku modelu

Musimy upewnić się, że kontroler przesyła do widoku prawidłowe dane stronicowania. Poniżej przedstawiłem test jednostkowy dodany do klasy `HomeControllerTests` w projekcie testów:

```

...
[Fact]
public void Can_Send_Pagination_View_Model() {

    // Przygotowanie.
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    })).AsQueryable<Product>());

    // Przygotowanie.
    HomeController controller =
        new HomeController(mock.Object) {PageSize = 3};
}

```

```

// Działanie.
ProductsListViewModel result =
    controller.Index(2).ViewData.Model as ProductsListViewModel;

// Asercje.
PagingInfo pageInfo = result.PagingInfo;
Assert.Equal(2, pageInfo.CurrentPage);
Assert.Equal(3, pageInfo.ItemsPerPage);
Assert.Equal(5, pageInfo.TotalItems);
Assert.Equal(2, pageInfo.TotalPages);
}
...

```

Musimy również zmienić nasze wcześniejsze testy jednostkowe, aby odzwierciedlić nowy wynik działania metody akcji `Index()`. Zmodyfikowane testy wyglądają następująco:

```

...
[Fact]
public void Can_Use_Repository()
{
    // Przygotowanie.
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"}
    })).AsQueryable<Product>();

    HomeController controller = new HomeController(mock.Object);

    // Działanie.
    ProductsListViewModel result =
        controller.Index().ViewData.Model as ProductsListViewModel;

    // Asercje.
    Product[] prodArray = result.Products.ToArray();
    Assert.True(prodArray.Length == 2);
    Assert.Equal("P1", prodArray[0].Name);
    Assert.Equal("P2", prodArray[1].Name);
}

[Fact]
public void Can_Paginate()
{
    // Przygotowanie.
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    })).AsQueryable<Product>();

    HomeController controller = new HomeController(mock.Object);
    controller.PageSize = 3;

    // Działanie.
    ProductsListViewModel result =
        controller.Index(2).ViewData.Model as ProductsListViewModel;

```

```
// Asercje.
Product[] prodArray = result.Products.ToArray();
Assert.True(prodArray.Length == 2);
Assert.Equal("P4", prodArray[0].Name);
Assert.Equal("P5", prodArray[1].Name);
}
...
```

Zwykle tworzę wspólną metodę konfiguracji testu, aby uniknąć powielania kodu w tego typu metodach testowych. Jednak tu zamieszczam testy jednostkowe w osobnych ramkach; musimy tworzyć testy samodzielnie.

Teraz widok oczekuje sekwencji obiektów Product, więc aby obsłużyć nowy typ modelu, musimy jeszcze zmienić plik *Index.cshtml*, jak pokazałem na listingu 7.31.

**Listing 7.31.** Zaktualizowany plik *Index.cshtml* w katalogu *SportsStore/Views/Home*

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

Zmieniłem dyrektywę `@model`, aby poinformować Razor, że teraz pracujemy na innym typie danych. Trzeba również zmodyfikować pętlę `foreach`, ponieważ źródłem danych jest właściwość `Products` w danych modelu.

## Wyświetlanie łączy stron

Mamy już wszystko przygotowane, aby dodać łączy stron do widoku *Index*. Utworzyliśmy model widoku, który zawiera dane stronicowania, zaktualizowaliśmy kontroler, aby dane te zostały przekazane do widoku, a następnie zmieniliśmy dyrektywę `@model`, aby pasowała do nowego typu modelu widoku. Pozostało nam dodanie elementu HTML, który będzie przetwarzany przez atrybut pomocniczy znacznika w celu utworzenia łączy stron. Zmiany konieczne do wprowadzenia przedstawiłem na listingu 7.32.

**Listing 7.32.** Dodanie łączy stronicowania do pliku *Index.cshtml* w katalogu *SportsStore/Views/Home*

```
@model ProductsListViewModel

@foreach (var p in Model.Products)
{
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="Index"></div>
```

Po ponownym uruchomieniu aplikacji i wykonaniu zapytania pod adresem `http://localhost:5000` możesz zobaczyć łącza stron na dole strony (rysunek 7.6). Styl strony jest nadal bardzo prosty, ale zmienimy to pod koniec rozdziału. Na tym etapie ważniejsze jest, że łącza te pozwalają na przechodzenie pomiędzy stronami w katalogu i przeglądanie dostępnych produktów. Gdy silnik Razor znajduje atrybut `page-model` znacznika `<div>`, nakazuje klasie `PageLinkTagHelper` transformację tego znacznika, co prowadzi do wygenerowania sekwencji łączy widocznych na rysunku.

## Ulepszanie adresów URL

Nasze łącza stron działają, ale nadal korzystają z ciągu zapytania do przekazywania danych do serwera w następujący sposób:

---

```
http://localhost/?productPage=2
```

---

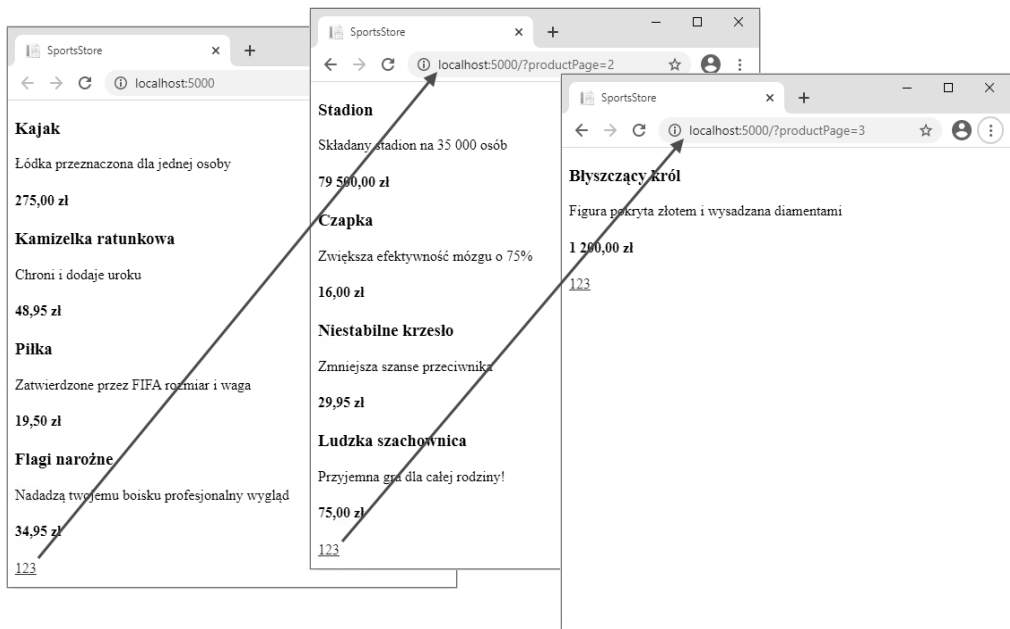
Możemy zrobić to lepiej, tworząc schemat oparty na wzorcu *składanych adresów URL*. Składany adres URL to taki, który ma sens dla użytkownika, tak jak poniższy:

---

```
http://localhost/Strona2
```

---

Na szczęście funkcjonalność routingu ASP.NET Core pozwala bardzo łatwo zmieniać schemat adresów URL w aplikacji. Wystarczy więc po prostu dodać nową trasę podczas rejestrowania oprogramowania pośredniczącego w metodzie `Configure()` klasy `Startup`. Odpowiednie zmiany konieczne do wprowadzenia przedstawiłem na listingu 7.33.



**Rysunek 7.6.** Wyświetlanie łączy nawigacji między stronami

**Listing 7.33.** Dodawanie nowej trasy w pliku *Startup.cs* w projekcie *SportsStore*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public Startup(IConfiguration config) =>
            Configuration = config;

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();

            services.AddDbContext<StoreDbContext>(opts =>
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]
                )
            );
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("pagination",
                    "Products/Page{productPage}",
                    new { Controller = "Home", action = "Index" });
                endpoints.MapDefaultControllerRoute();
            });

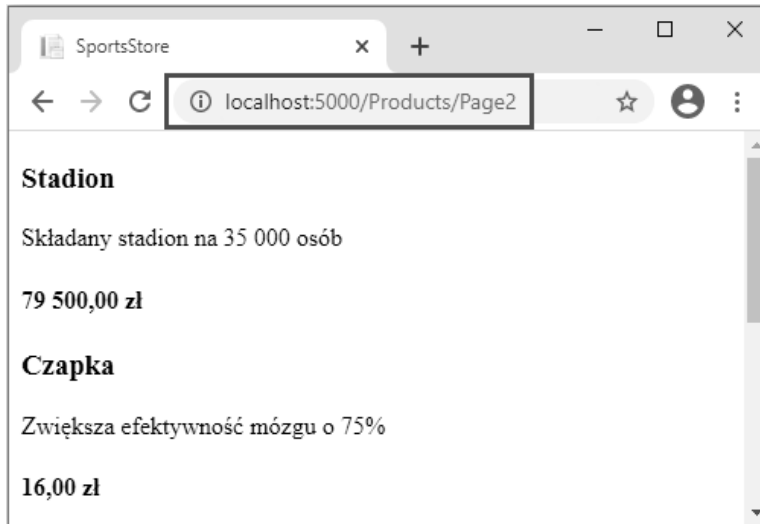
            SeedData.EnsurePopulated(app);
        }
    }
}

```

Bardzo ważne jest, aby nowa trasa została dodana przed wywołaniem metody `MapDefaultControllerRoute()`. Jak pokażę w rozdziale 13., trasy są przetwarzane w kolejności definiowania, a nasza nowa trasa musi mieć większy priorytet niż istniejąca.

To jedyne, co musimy zrobić w celu zmiany schematu URL dla stronicowania produktów. Framework ASP.NET Core jest ściśle zintegrowany z funkcjami routingu, więc taka zmiana jest automatycznie stosowana podczas przetwarzania adresów URL używanych przez aplikację, także tych generowanych przez atrybuty pomocnicze znaczników (tego rodzaju znacznik tworzy łącza nawigacyjne na naszej stronie).

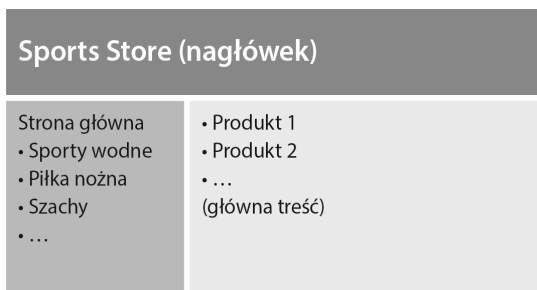
Po ponownym uruchomieniu aplikacji i wykonaniu żądania pod adresem `http://localhost:5000`, gdy przejdziesz do kolejnej strony, zobaczysz nowy schemat URL w działaniu (rysunek 7.7).



Rysunek 7.7. Nowy schemat URL wyświetlany w przeglądarce

## Dodawanie stylu

Do tej pory zbudowaliśmy całkiem niezłą infrastrukturę i nasza aplikacja zaczyna nabierać kształtu, ale nie zwracaliśmy uwagi na projekt graficzny. Wprawdzie książka nie jest poświęcona CSS ani projektowaniu dla WWW, ale w tym podrozdziale zajmiemy się szatą graficzną aplikacji SportsStore, gdyż teraz jej słaby wygląd przesłania techniczną doskonałość programu. W tym podrozdziale mam zamiar zaimplementować klasyczny, dwukolumnowy układ z nagłówkiem (rysunek 7.8).



Rysunek 7.8. Cel projektowy dla aplikacji SportsStore

## Instalacja pakietu Bootstrap

W celu nadania stylów CSS w aplikacji wykorzystamy framework Bootstrap. Jak wspomniałem w rozdziale 4., do instalacji pakietów działających po stronie klienta użyjemy LibMan. Jeżeli narzędzia LibMan nie zainstalowałeś podczas wykonywania przykładów zamieszczonych w rozdziale 4., użyj wiersza poleceń Windows PowerShell do wydania poleceń przedstawionych na listingu 7.34. Pierwsze z nich powoduje usunięcie istniejącego pakietu LibMan, zaś drugie instaluje ten pakiet w wersji wymaganej przez przykłady zamieszczone w książce.

### *Listing 7.34. Instalacja pakietu narzędzia LibMan*

```
PS C:\> dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
PS C:\> dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.0.96
```

Gdy masz zainstalowane narzędzie LibMan, z poziomu katalogu *SportsStore* wydaj polecenia przedstawione na listingu 7.35, aby w ten sposób zainicjalizować przykładowy projekt i zainstalować pakiet Bootstrap.

### *Listing 7.35. Inicjalizacja przykładowego projektu*

```
PS C:\> libman init -p cdnjs
PS C:\> libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

## Zastosowanie w aplikacji stylów Bootstrap

Układ widoku Razor dostarcza wspólną treść, której dzięki temu nie trzeba powielać w wielu widokach. Elementy przedstawione na listingu 7.36 dodaj do pliku *\_Layout.cshtml* w katalogu *Views/Shares*. Powodują one dodanie arkusza stylów Bootstrap CSS do treści przekazywanej przeglądarce WWW, a także zdefiniowanie nagłówka używanego w aplikacji SportsStore.

### *Listing 7.36. Zastosowanie stylów Bootstrap CSS w pliku \_Layout.cshtml w katalogu SportsStore/Views/Shared*

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width">
  <title>SportsStore</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <a class="navbar-brand ml-2">Sklep sportowy</a>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      Później umieścimy tu coś użytecznego.
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Dodanie arkusza stylu Bootstrap CSS do układu oznacza możliwość użycia tych stylów w dowolnym widoku zbudowanym na bazie tego układu. Na listingu 7.37 pokazałem dodanie stylów w pliku *Index.cshtml*.

**Listing 7.37.** Użycie Bootstrap w celu nadania stylów w pliku *Index.cshtml* w katalogu *SportStore/Views/Home*

```
@model ProductsListViewModel

@foreach (var p in Model.Products)
{
    <div class="card card-outline-primary m-1 p-1">
        <div class="bg-faded p-1">
            <h4>
                @p.Name
                <span class="badge badge-pill badge-primary" style="float:right">
                    <small>@p.Price.ToString("c")</small>
                </span>
            </h4>
        </div>
        <div class="card-text p-1">@p.Description</div>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="Index" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-outline-dark"
    page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>
```

Musimy jeszcze zmienić styl przycisków wygenerowanych przez klasę *PageLinkTagHelper*. Nie chcę wiązać klas Bootstrap z kodem C#, ponieważ to znacznie utrudnia wielokrotne użycie atrybutu pomocniczego znacznika w innych częściach aplikacji lub zmianę wyglądu przycisków. Dlatego też zdefiniowałem własne atrybuty dla znacznika `<div>` wskazujące wymagane klasy do użycia. Odpowiadające im właściwości dodałem do klasy atrybutu pomocniczego znacznika. Następnie użyłem wspomnianych atrybutów do nadania stylu znacznikom `<a>`, jak pokazałem na listingu 7.38.

**Listing 7.38.** Dodanie klas do elementów generowanych w pliku *PageLinkTagHelper.cs* w katalogu *SportsStore/Infrastructure*

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure
{
    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper
    {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory)
        {
            urlHelperFactory = helperFactory;
        }
    }
}
```



```

    }

    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewContext { get; set; }

    public PagingInfo PageModel { get; set; }

    public string PageAction { get; set; }

    public bool PageClassesEnabled { get; set; } = false;
    public string PageClass { get; set; }
    public string PageClassNormal { get; set; }
    public string PageClassSelected { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
        TagBuilder result = new TagBuilder("div");
        for (int i = 1; i <= PageModel.TotalPages; i++)
        {
            TagBuilder tag = new TagBuilder("a");
            tag.Attributes["href"] = urlHelper.Action(PageAction,
                new { productPage = i });
            if (PageClassesEnabled)
            {
                tag.AddCssClass(PageClass);
                tag.AddCssClass(i == PageModel.CurrentPage
                    ? PageClassSelected : PageClassNormal);
            }
            tag.InnerHtml.Append(i.ToString());
            result.InnerHtml.AppendHtml(tag);
        }
        output.Content.AppendHtml(result.InnerHtml);
    }
}

```

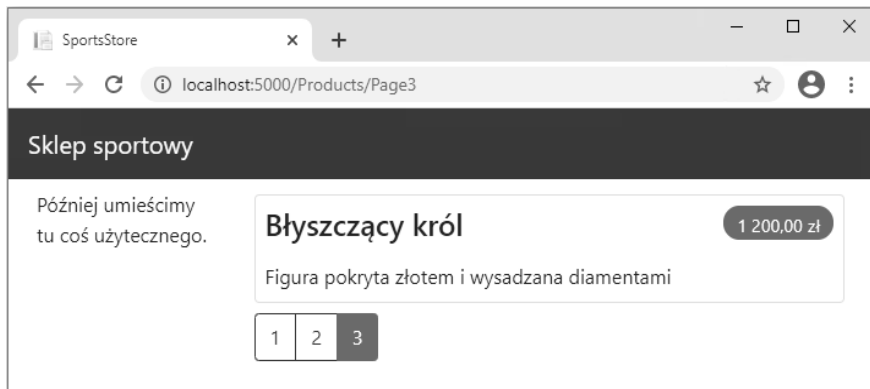
Wartości atrybutów są automatycznie używane do zdefiniowania wartości właściwości atrybutu pomocniczego znacznika. Stosowane jest mapowanie między formatem nazwy atrybutu HTML (`page-class-normal`) i formatem nazwy właściwości C# (`PageClassNormal`). Dzięki temu atrybut pomocniczy znacznika może odpowiednio reagować w zależności od atrybutu elementu HTML, co daje znacznie elastyczniejszy sposób generowania treści w aplikacji ASP.NET Core.

Po ponownym uruchomieniu aplikacji i wykonaniu żądania pod adresem <http://localhost:5000> zauważysz poprawę wyglądu — przynajmniej troszeczkę. Zmiany te są pokazane na rysunku 7.9.

## Tworzenie widoku częściowego

Końcowym zadaniem w tym rozdziale będzie refaktoring naszej aplikacji — uprościmy widok *Index.cshtml*. Utworzymy *widok częściowy*, to jest raczej fragment treści, który można dołączyć do innego widoku, a nie szablon. Widoki częściowe dokładnie omówię w rozdziale 22. Teraz wystarczy wiedzieć, że pomagają zmniejszyć ilość powielonego kodu, szczególnie jeżeli używamy tych samych danych w kilku miejscach aplikacji. Zamiast kopiować i wklejać ten sam kod znaczników Razor

w wielu widokach, można go zdefiniować raz w widoku częściowym. Aby dodać widok częściowy, do katalogu *Views/Shared* dodaj plik widoku Razor o nazwie *ProductSummary.cshtml* i umieść w nim kod przedstawiony na listingu 7.39.



**Rysunek 7.9.** Poprawiony układ graficzny aplikacji *SportsStore*

**Listing 7.39.** Zawartość pliku *ProductSummary.cshtml* w katalogu *SportsStore/Views/Shared*

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <span class="card-text p-1">@Model.Description</span>
</div>
```

Teraz musimy zmodyfikować widok *Views/Products/Index.cshtml*, aby korzystał z widoku częściowego. Zmiany są zamieszczone na listingu 7.40.

**Listing 7.40.** Użycie widoku częściowego w pliku *Index.cshtml* w katalogu *SportsStore/Views/Home*

```
@model ProductsListViewModel

@foreach (var p in Model.Products)
{
  <partial name="ProductSummary" model="p" />
}

<div page-model="@Model.PagingInfo" page-action="Index" page-classes-enabled="true"
  page-class="btn" page-class-normal="btn-outline-dark"
  page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>
```

Kod, który wcześniej znajdował się w pętli `foreach`, w widoku `Index.cshtml`, został przeniesiony do nowego widoku częściowego. Ten widok częściowy wywołujemy za pomocą elementu `<partial>` używając atrybutów `name` i `model` pozwalających na określenie nazwy widoku oraz obiektu modelu widoku. Korzystanie z widoków częściowych jest dobrą praktyką, ponieważ pozwala to na użycie tego samego kodu znaczników w każdym widoku, który musi wyświetlać pewne dane (w omawianym przykładzie to będzie podsumowanie produktów).

Po ponownym uruchomieniu aplikacji i wykonaniu żądania pod adresem `http://localhost:5000` możesz zobaczyć, że widok częściowy nie zmienia wyglądu aplikacji. Zmieniła się jedynie lokalizacja, w której silnik Razor odnajduje treść używaną do wygenerowania odpowiedzi przekazywanej do przeglądarki WWW.

## Podsumowanie

W tym rozdziale zbudowaliśmy większość podstawowej infrastruktury dla aplikacji SportsStore. Nie posiada ona zbyt wielu funkcji, które można pokazać klientowi, ale „pod maską” mamy już początki modelu domeny oraz repozytorium produktów obsługujące bazę SQL Server za pośrednictwem Entity Framework Core. Mamy jeden kontroler, `HomeController`, który pozwala wygenerować stronicowaną listę produktów, skonfigurowaliśmy też przyjazny schemat adresów URL.

Jeżeli uważasz, że w tym rozdziale było zbyt dużo konfiguracji i za mało wyników, to w następnym znajdziesz wyrównanie. Mamy zbudowane podstawowe elementy, więc możemy pójść dalej i dodać wszystkie funkcje użytkownika — nawigację według kategorii, koszyk na zakupy i proces składania zamówienia.



# 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**

# ASP.NET Core MVC: od teraz napiszesz każdą aplikację!

ASP.NET Core jest doskonałym narzędziem dla profesjonalnych programistów. Zapewnia kompletne środowisko wyposażone w bogaty zestaw funkcjonalności, pozwalające wykorzystywać najnowsze technologie w tworzonych aplikacjach. Kolejna wersja tej lubianej platformy, ASP.NET Core 3, imponuje potencjałem i wspaniałymi możliwościami. Przed rozpoczęciem tworzenia aplikacji jednak warto się zapoznać z nowymi elementami ASP.NET Core 3, by się dowiedzieć, jakie zaawansowane techniki oferuje i jakie korzyści można dzięki nim osiągnąć.

Ta książka jest gruntownie uaktualnionym przewodnikiem po ASP.NET Core 3, przeznaczonym dla profesjonalnych programistów, którzy chcą w swoich projektach wykorzystać w pełni potencjał technologii Microsoftu. Wydanie zawiera kompletne i praktyczne omówienie ASP.NET Core 3. Znalazły się tu przydatne wskazówki dotyczące narzędzi potrzebnych podczas tworzenia nowoczesnych, skalowalnych aplikacji internetowych. Nowe funkcje — takie jak MVC 3, strony Razor, serwer Blazor i technologia Blazor WebAssembly — zostały dokładnie omówione i zaprezentowane na przykładach. Poruszono tutaj również takie tematy jak komponenty oprogramowania pośredniczącego, wbudowane usługi czy dołączanie modelu w żądaniu, a także zaprezentowano wiele bardziej zaawansowanych zagadnień, jak routing punktów końcowych i mechanizm wstrzykiwania zależności.

W tej książce między innymi:

- solidne podstawy platformy ASP.NET Core
- korzystanie z funkcji wbudowanych w ASP.NET Core 3
- używanie szablonów
- usługi sieciowe typu RESTful, aplikacje internetowe i aplikacje działające po stronie klienta
- stosowanie modeli programistycznych

**Adam Freeman** jest doświadczonym programistą, autorem wielu świetnie przyjętych książek o programowaniu. Tworzył duże systemy rozproszone (platformy e-commerce). Zajmował stanowiska kierownicze w wielu firmach, w tym w Netscape, Sun Microsystems, na giełdzie NASDAQ i w bankach o międzynarodowym zasięgu. Obecnie jest na emeryturze i poświęca czas na pisanie i bieganie na długich dystansach.

	<i>Sprawdź nasze szkolenia!</i>	<b>KOD KORZYŚCI</b> <i>Sięgnij po więcej!</i>	
 <a href="http://helion.pl">helion.pl</a>	 <b>SZKOLENIA</b> AKADEMIA IT & BUSINESS	ISBN 978-83-283-7890-2	
 <b>HELION SA</b> ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	<a href="http://HELIONSZKOLENIA.PL">HELIONSZKOLENIA.PL</a>	 9 788328 378902	
<b>INFORMATYKA W NAJLEPSZYM WYDANIU</b>		Cena: 199,00 zł	