



ASP.NET Core MVC 2

Zaawansowane programowanie

Wydanie VII

—

Adam Freeman

Helion 

Apress®

Tytuł oryginału: Pro ASP.NET Core MVC 2, 7th Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4600-0

Original edition copyright © 2017 by Adam Freeman
All rights reserved.

Polish edition copyright © 2018 by HELION SA.
All rights reserved.

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

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

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

Autor oraz Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/aspnm7.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/aspnm7>
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	19
	O recenzencie technicznym	21
Część I	Wprowadzenie do ASP.NET Core MVC	23
Rozdział 1.	ASP.NET Core MVC w szerszym kontekście	25
	Historia powstania ASP.NET Core MVC	25
	ASP.NET Web Forms	26
	Oryginalny framework MVC	27
	Poznajemy ASP.NET Core	28
	Co nowego w ASP.NET Core MVC 2?	28
	Najważniejsze zalety ASP.NET Core MVC	28
	Co powinienem wiedzieć?	31
	Jaka jest struktura książki?	31
	Część I. Wprowadzenie do ASP.NET Core MVC	31
	Część II. Szczegółowe omówienie frameworka ASP.NET Core MVC	31
	Gdzie znaleźć przykładowe fragmenty kodu?	32
	Podsumowanie	32
Rozdział 2.	Pierwsza aplikacja MVC	33
	Instalacja Visual Studio	33
	Instalacja .NET Core 2.0 SDK	34
	Tworzenie nowego projektu ASP.NET Core MVC	35
	Edycja kontrolera	38
	Poznajemy trasy	40
	Generowanie stron WWW	41
	Tworzenie i generowanie widoku	41
	Dynamiczne dodawanie treści	43

Tworzenie prostej aplikacji wprowadzania danych	45
Przygotowanie sceny	45
Projektowanie modelu danych	46
Utworzenie drugiej metody akcji i widoku ściśle określonego typu	47
Łączenie metod akcji	48
Budowanie formularza	49
Obsługa formularzy	51
Wyświetlenie odpowiedzi	55
Dodanie kontroli poprawności danych	57
Nadanie stylu zawartości	63
Podsumowanie	68
Rozdział 3. Wzorzec MVC, projekty i konwencje	69
Historia MVC	69
Wprowadzenie do wzorca MVC	69
Poznajemy model	70
Poznajemy kontroler	70
Poznajemy widok	71
Implementacja MVC w ASP.NET	71
Porównanie MVC z innymi wzorcami	72
Poznajemy wzorzec Smart UI	72
Architektura model-widok	73
Klasyczna architektura trójwarstwowa	74
Odmiany MVC	74
Poznajemy projekt ASP.NET Core MVC	75
Utworzenie projektu	76
Poznajemy konwencje MVC	79
Podsumowanie	80
Rozdział 4. Najważniejsze cechy języka C#	81
Utworzenie przykładowego projektu	82
Dodanie obsługi ASP.NET Core MVC	82
Utworzenie komponentów aplikacji ASP.NET Core MVC	84
Użycie operatora warunkowego null	86
Łączenie operatorów warunkowych null	87
Łączenie operatorów: warunkowego i koalescencji	88
Użycie automatycznie implementowanych właściwości	89
Użycie automatycznie implementowanych metod inicjalizacyjnych właściwości	90
Utworzenie automatycznie implementowanych właściwości tylko do odczytu	91
Interpolacja ciągu tekstowego	93
Użycie inicjalizatorów obiektów i kolekcji	94
Użycie inicjalizatora indeksu	95
Dopasowanie wzorca	96
Dopasowanie wzorca w konstrukcji switch	97

Użycie metod rozszerzających	98
Stosowanie metod rozszerzających do interfejsów	100
Tworzenie filtrujących metod rozszerzających	101
Użycie wyrażeń lambda	103
Definiowanie funkcji	104
Użycie wyrażeń lambda w postaci metod i właściwości	107
Użycie inferencji typów i typów anonimowych	109
Użycie typów anonimowych	110
Użycie metod asynchronicznych	111
Bezpośrednia praca z zadaniami	112
Użycie słów kluczowych async i await	113
Pobieranie nazw	115
Podsumowanie	117
Rozdział 5. Praca z silnikiem Razor	119
Utworzenie przykładowego projektu	120
Definiowanie modelu	121
Utworzenie kontrolera	121
Tworzenie widoku	122
Korzystanie z obiektów modelu	123
Używanie pliku poleceń importujących widoki	125
Praca z układami	126
Tworzenie układu	127
Stosowanie układu	129
Użycie pliku ViewStart	129
Użycie wyrażeń Razor	131
Wstawianie wartości danych	132
Przypisanie wartości atrybutu	133
Użycie konstrukcji warunkowych	134
Wyświetlanie zawartości tablic i kolekcji	136
Podsumowanie	138
Rozdział 6. Praca z Visual Studio	139
Utworzenie przykładowego projektu	139
Utworzenie modelu	140
Utworzenie kontrolera i widoku	141
Zarządzanie pakietami oprogramowania	142
Poznajemy NuGet	143
Poznajemy Bower	144
Poznajemy iteracyjny model programowania	148
Modyfikacje widoków Razor	148
Modyfikacje klas C#	149
Użycie funkcji połączonych przeglądarek	157

Przygotowanie kodu JavaScript i CSS do wdrożenia	161
Włączenie obsługi dostarczania treści statycznej	161
Dodanie treści statycznej do projektu	162
Uaktualnienie widoku	164
Łączenie i minimalizacja plików w aplikacjach MVC	165
Podsumowanie	169
Rozdział 7. Testy jednostkowe w aplikacji MVC	171
Utworzenie przykładowego projektu	172
Włączenie obsługi wbudowanych atrybutów pomocniczych znaczników	172
Dodanie akcji do kontrolera	173
Utworzenie formularza do wprowadzania danych	173
Uaktualnienie widoku Index	174
Testy jednostkowe w aplikacji MVC	175
Utworzenie projektu testów jednostkowych	176
Dodanie odwołania do projektu aplikacji	177
Tworzenie i wykonywanie testów jednostkowych	177
Izolowanie komponentów poddawanych testom jednostkowym	181
Usprawnianie testów jednostkowych	190
Parametryzowanie testu jednostkowego	190
Usprawnianie implementacji imitacji	194
Podsumowanie	199
Rozdział 8. SportsStore — kompletna aplikacja	201
Zaczynamy	202
Tworzenie projektu MVC	202
Tworzenie projektu testów jednostkowych	206
Sprawdzenie i uruchomienie aplikacji	207
Tworzenie modelu domeny	208
Tworzenie repozytorium	208
Tworzenie imitacji repozytorium	209
Rejestrowanie usługi repozytorium	209
Wyświetlanie listy produktów	210
Dodawanie kontrolera	211
Dodawanie i konfigurowanie widoku	212
Konfigurowanie trasy domyślnej	214
Uruchamianie aplikacji	215
Przygotowanie bazy danych	215
Instalowanie pakietu narzędzi Entity Framework Core	216
Utworzenie klas bazy danych	217
Utworzenie klasy repozytorium	217
Definiowanie ciągu tekstowego połączenia	218
Konfigurowanie aplikacji	219
Utworzenie i zastosowanie migracji bazy danych	221
Tworzenie danych początkowych	222

Dodanie stronicowania	225
Wyświetlanie łączy stron	227
Ulepszanie adresów URL	235
Dodawanie stylu	237
Instalacja pakietu Bootstrap	237
Zastosowanie w aplikacji stylów Bootstrap	237
Tworzenie widoku częściowego	240
Podsumowanie	242
Rozdział 9. SportsStore — nawigacja	243
Dodawanie kontrolki nawigacji	243
Filtrowanie listy produktów	243
Ulepszanie schematu URL	247
Budowanie menu nawigacji po kategoriach	251
Poprawianie licznika stron	258
Budowanie koszyka na zakupy	261
Definiowanie modelu koszyka	262
Tworzenie przycisków koszyka	265
Włączenie obsługi sesji	267
Implementowanie kontrolera koszyka	268
Wyświetlanie zawartości koszyka	270
Podsumowanie	272
Rozdział 10. SportsStore — ukończenie koszyka na zakupy	275
Dopracowanie modelu koszyka za pomocą usługi	275
Tworzenie klasy koszyka obsługującej magazyn danych	275
Rejestrowanie usługi	276
Uproszczenie kontrolera koszyka na zakupy	277
Kończenie budowania koszyka	278
Usuwanie produktów z koszyka	278
Dodawanie podsumowania koszyka	280
Składanie zamówień	282
Utworzenie klasy modelu	282
Dodawanie procesu składania zamówienia	284
Implementowanie mechanizmu przetwarzania zamówień	286
Zakończenie pracy nad kontrolerem koszyka	290
Wyświetlanie informacji o błędach systemu kontroli poprawności	293
Wyświetlanie strony podsumowania	294
Podsumowanie	295
Rozdział 11. SportsStore — administracja	297
Zarządzanie zamówieniami	297
Usprawnienie modelu	297
Dodanie akcji i widoku	298
Dodajemy zarządzanie katalogiem	301
Tworzenie kontrolera CRUD	302
Implementowanie widoku listy	304

Edycja produktów	305
Dodawanie nowych produktów	317
Usuwanie produktów	319
Podsumowanie	322
Rozdział 12. SportsStore — bezpieczeństwo i wdrożenie aplikacji	323
Zabezpieczanie funkcji administracyjnych	323
Utworzenie bazy danych dla systemu Identity	323
Zdefiniowanie prostej polityki autoryzacji	328
Utworzenie kontrolera AccountController i widoków	330
Przetestowanie polityki bezpieczeństwa	334
Wdrożenie aplikacji	334
Utworzenie baz danych	334
Przygotowanie aplikacji	336
Zastosowanie migracji bazy danych	339
Zarządzanie wstawieniem danych początkowych do bazy danych	340
Wdrożenie aplikacji	344
Podsumowanie	348
Rozdział 13. Praca z Visual Studio Code	349
Przygotowanie środowiska programistycznego	349
Instalacja Node.js	349
Sprawdzenie instalacji Node	351
Instalacja Git	351
Sprawdzenie instalacji Git	351
Instalacja narzędzia bower	352
Instalacja .NET Core	352
Sprawdzenie instalacji .NET Core	353
Instalacja Visual Studio Code	353
Sprawdzenie instalacji Visual Studio Code	354
Instalacja rozszerzenia Visual Studio Code C#	354
Utworzenie projektu ASP.NET Core	355
Przygotowanie projektu z użyciem Visual Studio Code	356
Zarządzanie pakietami działającymi po stronie klienta	357
Konfigurowanie aplikacji	358
Kompilacja i uruchomienie projektu	359
Odtworzenie aplikacji PartyInvites	359
Utworzenie modelu i repozytorium	360
Utworzenie bazy danych	363
Utworzenie kontrolera i widoków	365
Testy jednostkowe w Visual Studio Code	369
Utworzenie testu jednostkowego	370
Wykonanie testów	370
Podsumowanie	371

Część II	ASP.NET Core MVC 2 w szczegółach	373
Rozdział 14.	Konfigurowanie aplikacji	375
	Utworzenie przykładowego projektu	377
	Konfigurowanie projektu	378
	Dodawanie pakietów do projektu	379
	Dodawanie pakietów narzędziowych do projektu	381
	Poznajemy klasę Program	381
	Poznajemy szczegóły konfiguracji	382
	Poznajemy klasę Startup	385
	Poznajemy usługi ASP.NET	388
	Poznajemy oprogramowanie pośredniczące ASP.NET	391
	Poznajemy sposób wywoływania metody Configure()	401
	Dodawanie pozostałych komponentów oprogramowania pośredniczącego	405
	Konfigurowanie aplikacji	410
	Użycie danych konfiguracyjnych	414
	Konfigurowanie systemu rejestrowania danych	416
	Konfigurowanie mechanizmu wstrzykiwania zależności	420
	Konfiguracja usług MVC	421
	Praca ze skomplikowaną konfiguracją	423
	Utworzenie oddzielnych zewnętrznych plików konfiguracyjnych	423
	Utworzenie różnych metod konfiguracyjnych	424
	Utworzenie różnych klas konfiguracyjnych	426
	Podsumowanie	428
Rozdział 15.	Routing URL	429
	Utworzenie przykładowego projektu	430
	Utworzenie klasy modelu	431
	Utworzenie przykładowych kontrolerów	431
	Utworzenie widoku	433
	Wprowadzenie do wzorców URL	434
	Tworzenie i rejestrowanie prostej trasy	435
	Definiowanie wartości domyślnych	436
	Definiowanie osadzonych wartości domyślnych	438
	Użycie statycznych segmentów adresu URL	440
	Definiowanie własnych zmiennych segmentów	444
	Użycie własnych zmiennych jako parametrów metod akcji	447
	Definiowanie opcjonalnych segmentów URL	448
	Definiowanie tras o zmiennej długości	450
	Ograniczenia tras	452
	Ograniczanie trasy z użyciem wyrażeń regularnych	456
	Użycie ograniczeń dotyczących typu i wartości	457
	Ograniczanie trasy do zbioru wartości	458
	Definiowanie własnych ograniczeń	460

- Użycie atrybutów routingu462
 - Przygotowanie do użycia atrybutów routingu 462
 - Włączanie i stosowanie atrybutów routingu 463
 - Zastosowanie ograniczeń trasy 466
- Podsumowanie467
- Rozdział 16. Zaawansowane funkcje routingu 469**
 - Utworzenie przykładowego projektu470
 - Generowanie wychodzących adresów URL w widokach471
 - Wygenerowanie wychodzącego adresu URL 472
 - Generowanie adresów URL (nie łączy) 482
 - Dostosowanie systemu routingu484
 - Zmiana konfiguracji systemu routingu 484
 - Tworzenie własnej implementacji klasy routingu 485
 - Korzystanie z obszarów496
 - Tworzenie obszaru 496
 - Utworzenie trasy obszaru 497
 - Wypełnianie obszaru 498
 - Generowanie łączy do akcji z obszarów 500
 - Najlepsze praktyki schematu adresów URL502
 - Twórz jasne i przyjazne dla człowieka adresy URL 502
 - GET oraz POST — wybierz właściwie 503
 - Podsumowanie503
- Rozdział 17. Kontrolery i akcje 505**
 - Utworzenie przykładowego projektu506
 - Przygotowanie widoków 507
 - Poznajemy kontrolery509
 - Tworzenie kontrolera509
 - Tworzenie kontrolera POCO 510
 - Użycie klasy bazowej kontrolera 512
 - Pobieranie danych kontekstu513
 - Pobieranie danych z obiektów kontekstu 513
 - Użycie parametrów metod akcji 517
 - Generowanie odpowiedzi519
 - Wygenerowanie odpowiedzi za pomocą obiektu kontekstu 519
 - Poznajemy wyniki akcji 520
 - Wygenerowanie odpowiedzi HTML 522
 - Wykonywanie przekierowań 530
 - Zwrot różnego typu treści 537
 - Udzielanie odpowiedzi wraz z zawartością plików 540
 - Zwracanie błędów i kodów HTTP 541
 - Pozostałe klasy wyniku akcji 543
 - Podsumowanie544

Rozdział 18. Wstrzykiwanie zależności	545
Utworzenie przykładowego projektu	546
Utworzenie modelu i repozytorium	547
Utworzenie kontrolera i widoku	548
Utworzenie projektu testów jednostkowych	550
Utworzenie luźno powiązanych ze sobą komponentów	550
Analiza luźno powiązanych ze sobą komponentów	551
Wprowadzenie do wstrzykiwania zależności na platformie ASP.NET	557
Przygotowanie do użycia mechanizmu wstrzykiwania zależności	557
Konfigurowanie dostawcy usługi	559
Testy jednostkowe kontrolera wraz ze zdefiniowaną zależnością	560
Użycie łańcucha zależności	561
Użycie mechanizmu wstrzykiwania zależności dla konkretnego typu	564
Poznajemy cykl życiowy usługi	566
Użycie cyklu życiowego usługi	566
Zastosowanie cyklu życiowego zasięgu	572
Zastosowanie cyklu życiowego usługi typu singleton	573
Użycie wstrzyknięcia akcji	575
Użycie atrybutów wstrzykiwania właściwości	575
Ręczne żądanie obiektu implementacji	576
Podsumowanie	577
Rozdział 19. Filtry	579
Utworzenie przykładowego projektu	580
Włączenie szyfrowania SSL	581
Utworzenie kontrolera i widoku	581
Użycie filtrów	583
Poznajemy filtry	586
Pobieranie danych kontekstu	587
Użycie filtrów autoryzacji	587
Użycie filtru autoryzacji	588
Użycie filtrów akcji	591
Utworzenie filtru akcji	592
Utworzenie asynchronicznego filtru akcji	593
Używanie filtru wyniku	594
Utworzenie filtru wyniku	595
Utworzenie asynchronicznego filtru wyniku	596
Utworzenie filtru hybrydowego — akcji i wyniku	598
Użycie filtrów wyjątków	599
Utworzenie filtru wyjątku	601
Użycie mechanizmu wstrzykiwania zależności z filtrami	603
Spełnienie zależności filtru	603
Zarządzanie cyklem życiowym filtru	607
Użycie filtrów globalnych	610

Poznajemy i zmieniamy kolejność wykonywania filtrów	612
Zmiana kolejności filtrów	614
Podsumowanie	615
Rozdział 20. Kontrolery API	617
Utworzenie przykładowego projektu	618
Utworzenie modelu i repozytorium	618
Utworzenie kontrolera i widoków	620
Poznajemy rolę kontrolerów typu RESTful	624
Problem związany z szybkością działania aplikacji	624
Problem związany z efektywnością działania aplikacji	624
Problem związany z otwartością aplikacji	625
Poznajemy kontrolery typu API i REST	625
Utworzenie kontrolera API	626
Testowanie kontrolera API	630
Użycie kontrolera API w przeglądarce WWW	635
Poznajemy sposoby formatowania treści	637
Poznajemy domyślną politykę treści	637
Poznajemy negocjację treści	639
Określanie formatu danych akcji	641
Pobranie formatu danych z trasy lub ciągu tekstowego zapytania	642
Włączenie pełnej negocjacji treści	644
Otrzymywanie danych w różnych formatach	645
Podsumowanie	646
Rozdział 21. Widoki	647
Utworzenie przykładowego projektu	648
Tworzenie własnego silnika widoku	649
Tworzenie własnej implementacji IView	651
Tworzenie implementacji IViewEngine	652
Rejestrowanie własnego silnika widoku	653
Testowanie silnika widoku	654
Korzystanie z silnika Razor	656
Przygotowanie przykładowego projektu	656
Poznajemy widoki Razor	658
Dodawanie dynamicznych treści do widoku Razor	662
Zastosowanie sekcji układu	663
Użycie widoków częściowych	668
Dodanie treści JSON do widoku	670
Konfigurowanie silnika Razor	672
Poznajemy ekspandery widoku	673
Podsumowanie	678

Rozdział 22. Komponenty widoku	679
Utworzenie przykładowego projektu	680
Utworzenie modeli i repozytoriów	680
Utworzenie kontrolera i widoków	682
Konfigurowanie aplikacji	685
Poznajemy komponent widoku	686
Utworzenie komponentu widoku	686
Utworzenie komponentu widoku typu POJO	687
Dziedziczenie po klasie bazowej ViewComponent	688
Poznajemy wynik działania komponentu widoku	690
Pobieranie danych kontekstu	695
Tworzenie asynchronicznego komponentu widoku	701
Utworzenie hybrydy — kontroler i komponent widoku	703
Utworzenie widoku hybrydowego	704
Użycie klasy hybrydowej	705
Podsumowanie	707
Rozdział 23. Poznajemy atrybuty pomocnicze znaczników	709
Utworzenie przykładowego projektu	710
Utworzenie modelu i repozytorium	710
Utworzenie kontrolera, układu i widoków	711
Konfigurowanie aplikacji	714
Utworzenie atrybutu pomocniczego znacznika	715
Zdefiniowanie klasy atrybutu pomocniczego znacznika	715
Rejestrowanie atrybutu pomocniczego znacznika	719
Użycie atrybutu pomocniczego znacznika	719
Zarządzanie zasięgiem atrybutu pomocniczego znacznika	721
Zaawansowane funkcje atrybutu pomocniczego znacznika	725
Tworzenie elementów skrótu	726
Umieszczanie treści przed elementem i po nim	728
Pobieranie danych kontekstu widoku	
za pomocą mechanizmu wstrzykiwania zależności	732
Praca z modelem widoku	734
Koordynacja między atrybutami pomocniczymi znaczników	736
Zawieszenie wygenerowania elementu	738
Podsumowanie	739
Rozdział 24. Użycie atrybutów pomocniczych znaczników formularza	741
Przygotowanie przykładowego projektu	742
Wyzerowanie widoków i układu	742
Praca ze znacznikami formularza HTML	744
Zdefiniowanie metody docelowej formularza	745
Użycie funkcji CSRF	746
Praca ze znacznikami <input>	747
Konfigurowanie znaczników <input>	748
Formatowanie wartości danych	750

Praca ze znacznikiem <label>	753
Praca ze znacznikami <select> i <option>	755
Użycie źródła danych do przygotowania znacznika <select>	756
Wygenerowanie znaczników <option> na podstawie typu wyliczeniowego	756
Praca ze znacznikiem <textarea>	761
Weryfikacja atrybutów pomocniczych znaczników formularza	763
Podsumowanie	763
Rozdział 25. Używanie pozostałych wbudowanych atrybutów	
pomocniczych znaczników	765
Przygotowanie przykładowego projektu	766
Używanie atrybutu pomocniczego znacznika <environment>	767
Używanie atrybutów pomocniczych znaczników	
obsługujących pliki JavaScript i CSS	767
Zarządzanie plikami JavaScript	768
Zarządzanie arkuszami stylów CSS	777
Praca ze znacznikiem <a>	780
Praca ze znacznikiem <image>	781
Użycie buforowanych danych	782
Określenie czasu wygaśnięcia buforowanej treści	785
Użycie wariantów buforowania	787
Użycie względnego adresu URL w aplikacji	788
Podsumowanie	791
Rozdział 26. Dołączanie modelu	793
Utworzenie przykładowego projektu	794
Utworzenie modelu i repozytorium	794
Utworzenie kontrolera i widoku	796
Konfigurowanie aplikacji	797
Poznajemy dołączanie modelu	799
Poznajemy dołączanie wartości domyślnej	800
Dołączanie typów prostych	802
Dołączanie typów złożonych	803
Dołączanie tablic i kolekcji	813
Określanie źródła dołączania modelu	819
Wybór standardowego źródła danych dla funkcji dołączania modelu	821
Użycie nagłówek jako źródła danych dla funkcji dołączania modelu	822
Użycie treści żądania jako źródła danych dla funkcji dołączania modelu	825
Podsumowanie	827
Rozdział 27. Kontrola poprawności danych modelu	829
Utworzenie przykładowego projektu	830
Utworzenie modelu	831
Utworzenie kontrolera	832
Utworzenie układu i widoków	832
Potrzeba stosowania kontroli poprawności danych modelu	834

Jawna kontrola poprawności modelu	835
Wyświetlenie użytkownikowi błędów podczas kontroli poprawności	838
Wyświetlanie komunikatów kontroli poprawności	840
Wyświetlanie błędów kontroli poprawności na poziomie właściwości	843
Wyświetlanie błędów kontroli poprawności na poziomie modelu	845
Definiowanie reguł poprawności za pomocą metadanych	848
Tworzenie własnego atrybutu kontroli poprawności	851
Użycie kontroli poprawności po stronie klienta	853
Wykonywanie zdalnej kontroli poprawności	856
Podsumowanie	859
Rozdział 28. Rozpoczęcie pracy z ASP.NET Core Identity	861
Utworzenie przykładowego projektu	862
Utworzenie kontrolera i widoku	863
Konfiguracja ASP.NET Core Identity	865
Utworzenie klasy użytkownika	865
Utworzenie klasy kontekstu bazy danych	867
Konfigurowanie ciągu tekstowego połączenia z bazą danych	867
Utworzenie bazy danych systemu ASP.NET Core Identity	869
Używanie systemu ASP.NET Core Identity	870
Lista kont użytkowników	870
Utworzenie konta użytkownika	872
Kontrola poprawności hasła	876
Kontrola poprawności informacji o użytkowniku	884
Ukończenie przygotowania funkcji administracyjnych	889
Implementacja funkcji usunięcia konta użytkownika	890
Implementacja funkcji edycji konta użytkownika	891
Podsumowanie	896
Rozdział 29. Użycie ASP.NET Core Identity	897
Utworzenie przykładowego projektu	897
Uwierzytelnianie użytkownika	898
Przygotowanie do implementacji uwierzytelniania	900
Dodanie funkcji uwierzytelniania użytkownika	903
Testowanie uwierzytelniania	905
Uwierzytelnianie użytkownika z uwzględnieniem roli	906
Tworzenie i usuwanie ról	907
Zarządzanie użytkownikami przypisanymi do roli	912
Używanie ról podczas autoryzacji	917
Przygotowanie bazy danych	921
Podsumowanie	924
Rozdział 30. Zaawansowane funkcje ASP.NET Core Identity	925
Utworzenie przykładowego projektu	926
Dodawanie kolejnych właściwości do klasy przedstawiającej użytkownika	926

Przygotowanie do migracji bazy danych	930
Testowanie nowych właściwości	931
Praca z oświadczeniami i polityką	931
Poznajemy oświadczenie	932
Tworzenie oświadczenia	936
Użycie polityki autoryzacji	939
Użycie polityki w celu autoryzacji dostępu do zasobu	945
Uwierzytelnianie za pomocą dostawcy zewnętrznego	950
Zarejestrowanie aplikacji w Google	951
Włączenie uwierzytelniania za pomocą Google	951
Podsumowanie	956
Rozdział 31. Konwencje dotyczące modelu i ograniczenia akcji	957
Utworzenie przykładowego projektu	957
Utworzenie modelu widoku, kontrolera i widoku	958
Używanie modelu aplikacji i konwencji modelu	960
Poznajemy model aplikacji	961
Role konwencji modelu	965
Utworzenie konwencji modelu	965
Kolejność wykonywania konwencji modelu	970
Utworzenie globalnych konwencji modelu	971
Używanie ograniczeń akcji	973
Przygotowanie przykładowego projektu	973
Poznajemy ograniczenie akcji	975
Utworzenie ograniczenia akcji	976
Spełnianie zależności w ograniczeniu akcji	981
Podsumowanie	983
Skorowidz	985

ROZDZIAŁ 8.



SportsStore — kompletna aplikacja

W poprzednich rozdziałach zbudowałeś już pierwsze proste aplikacje MVC. Zapoznałeś się z wzorcem MVC. Przedstawiłem najważniejsze funkcje C# oraz narzędzia wykorzystywane przez dobrych programistów MVC. 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 MVC, 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 MVC 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 MVC 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 MVC.

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 MVC. Oczywiście obsługa testów jednostkowych to jeden z kluczowych powodów, dla których framework ASP.NET Core MVC zyskuje coraz większą popularność.

Większości funkcji MVC, z jakich będę korzystać 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 MVC. Szczególnie powinienś zwrócić uwagę na tworzenie widoków. Jeżeli nie będziesz się ściśle stosował do przedstawianych poleceń, aplikacja może się dziwnie zachowywać.

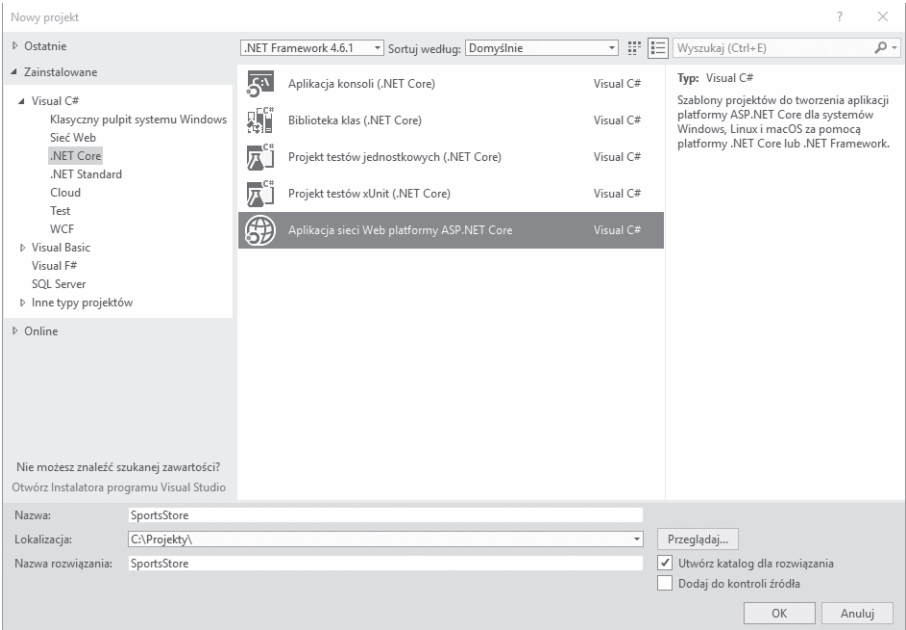
Zaczynamy

Jeżeli planujesz tworzyć aplikację SportsStore równoległe z lekturą, powinienś mieć zainstalowane oprogramowanie Visual Studio. Upewnij się o wybraniu opcji *LocalDB* podczas instalacji, ponieważ jest ona wymagana w celu trwałego przechowywania danych. Wymieniona opcja będzie użyta w trakcie instalacji Visual Studio, o ile wykonałeś kroki przedstawione w rozdziale 2.

- **Uwaga** Jeżeli chcesz po prostu zapoznać się z projektem bez jego samodzielnego utworzenia, gotowa aplikacja jest również dostępna w pliku archiwum kodu źródłowego pod adresem <ftp://ftp.helion.pl/przyklady/aspnm7.zip>. Nie musisz oczywiście przeglądać tego kodu. Starałem się, aby rysunki i listingi kodu były możliwie czytelne, dzięki czemu możesz czytać tę książkę w pociągu lub w kawiarni.

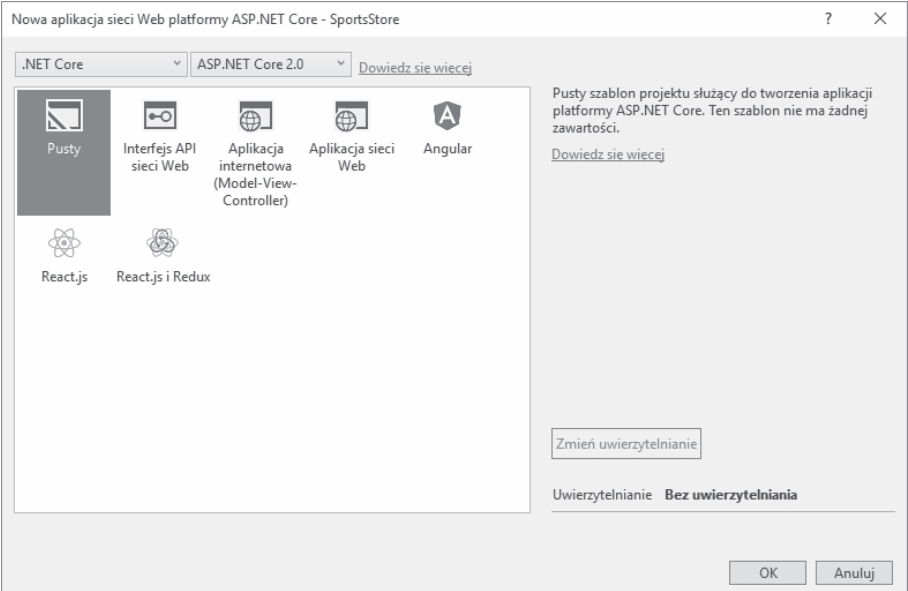
Tworzenie projektu MVC

Będziemy stosować to samo proste podejście, jakiego użyłem we wcześniejszych rozdziałach — pracę zaczniemy od pustego projektu, do którego następnie będziemy dodawać niezbędne pliki konfiguracyjne i komponenty. Z menu *Plik* w Visual Studio wybierz opcję *Nowy/Projekt...*, a następnie zaznacz szablon *Aplikacja sieci Web platformy .NET Core*, jak pokazałem na rysunku 8.1. Projektowi nadaj nazwę *SportsStore* i kliknij OK.



Rysunek 8.1. Tworzenie nowego projektu w Visual Studio

Wybierz szablon *Pusty*, jak pokazałem na rysunku 8.2, a następnie kliknij przycisk OK, aby utworzyć projekt *SportsStore*. Upewnij się o wybraniu opcji *.NET Core* i *ASP.NET Core MVC 2.0* z rozwijanych menu na górze, jak pokazałem na rysunku. Jeżeli widzisz opcję dotyczącą obsługi kontenerów Docker, upewnij się o jej odznaczeniu, a następnie kliknij przycisk OK, tworząc w ten sposób projekt aplikacji *SportsStore*.



Rysunek 8.2. Wybór szablonu dla nowego projektu w Visual Studio

Utworzenie struktury katalogu

Kolejnym krokiem jest dodanie katalogów przeznaczonych dla komponentów wymaganych podczas tworzenia aplikacji MVC, czyli modeli, kontrolerów i widoków. Dla każdego katalogu wymienionego w tabeli 8.1 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. Później będą nam potrzebne jeszcze inne katalogi, natomiast wymienione tutaj odzwierciedlają najważniejsze części aplikacji MVC i są wystarczające do rozpoczęcia pracy z projektem.

Tabela 8.1. Katalogi wymagane przez projekt *SportsStore*

Nazwa	Opis
<i>Models</i>	Ten katalog będzie zawierał klasy modelu.
<i>Controllers</i>	Ten katalog będzie zawierał klasy kontrolera.
<i>Views</i>	Ten katalog będzie zawierał wszystko to, co wiąże się z widokami, czyli między innymi poszczególne pliki widoków, plik <i>_ViewStart.cshtml</i> oraz plik poleceń importujących widoki.

Konfiguracja aplikacji

Klasa *Startup* jest odpowiedzialna za konfigurację aplikacji ASP.NET Core MVC. Na listingu 8.1 przedstawiłem zmiany konieczne do wprowadzenia w klasie *Startup*, aby włączyć obsługę frameworka MVC oraz pewnych funkcji przydatnych podczas tworzenia aplikacji.

■ **Uwaga** Klasa Startup ma bardzo ważne znaczenie na platformie ASP.NET Core. Więcej informacji na temat tej klasy przedstawię w rozdziale 14.

Listing 8.1. Włączanie różnych funkcji 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;

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

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

            });
        }
    }
}
```

Metoda `ConfigureServices()` jest używana w celu skonfigurowania współdzielonych obiektów, które będą mogły być używane w aplikacji za pomocą mechanizmu wstrzykiwania zależności, którym dokładnie zajmę się w rozdziale 18. Metoda `AddMvc()` wywoływana w wymienionej wcześniej `ConfigureServices()` jest metodą rozszerzenia i odpowiada za skonfigurowanie obiektów współdzielonych używanych w aplikacjach MVC.

Metoda `Configure()` jest używana do skonfigurowania funkcji otrzymujących i przetwarzających żądania HTTP. Każda metoda wywoływana w `Configure()` jest metodą rozszerzenia przygotowującą procesor żądania HTTP, jak przedstawiłem w tabeli 8.2.

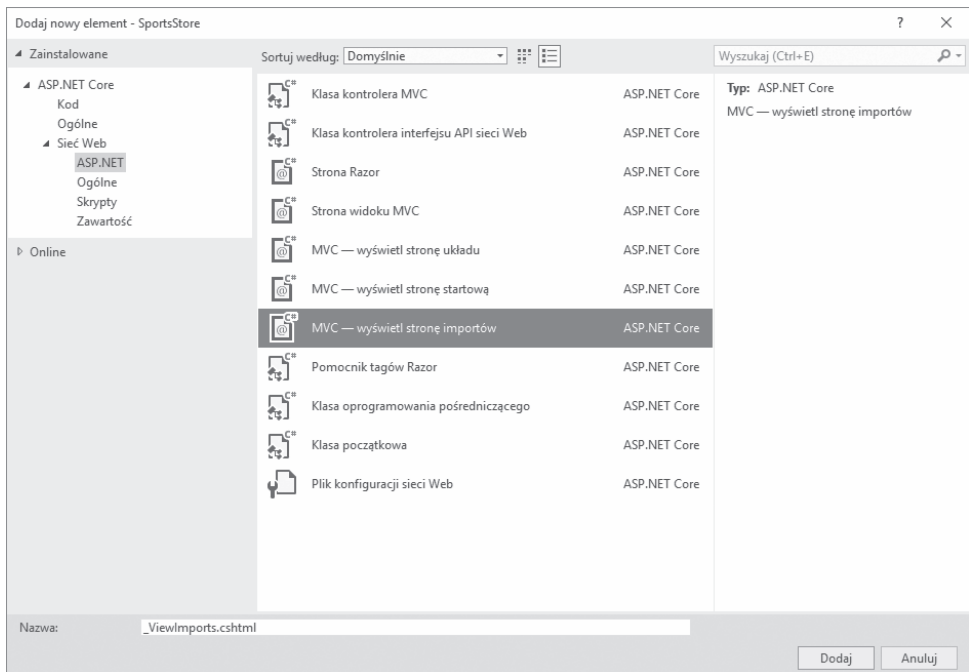
Tabela 8.2. Metody początkowe wywoływane w klasie Start

Metoda	Opis
<code>UseDeveloperExceptionPage()</code>	Metoda rozszerzenia wyświetlająca informacje szczegółowe dotyczące wyjątku zgłoszonego w aplikacji. Takie rozwiązanie okazuje się użyteczne na etapie pracy nad aplikacją. Ta metoda nie powinna być używana we wdrożonej aplikacji. W rozdziale 12. dowiesz się, jak można ją wyłączyć.
<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 żadanego zasobu.

Tabela 8.2. Metody początkowe wywoływane w klasie Start (ciąg dalszy)

Metoda	Opis
UseStaticFiles()	Ta metoda rozszerzenia włącza obsługę treści statycznej znajdującej się w katalogu <i>wwwroot</i> .
UseMvc()	Metoda rozszerzenia włączająca ASP.NET Core MVC.

Kolejnym krokiem jest przygotowanie aplikacji do obsługi widoków Razor. Prawym przyciskiem myszy kliknij katalog *Views*, wybierz *Dodaj/Nowy element...* z menu kontekstowego, a następnie zaznacz element *MVC — wyświetl stronę importów* w kategorii *ASP.NET*, jak pokazałem na rysunku 8.3.

**Rysunek 8.3.** Utworzenie pliku importującego widoki

Po kliknięciu przycisku *Dodaj* zostanie utworzony plik o nazwie *_ViewImports.cshtml*. Umieść w nim zawartość przedstawioną na listingu 8.2.

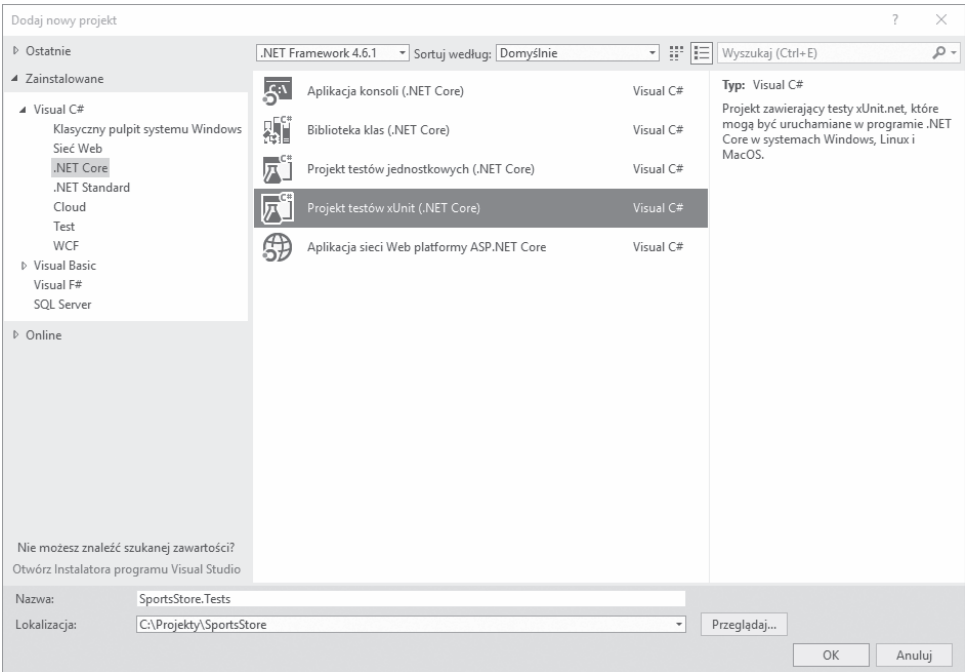
Listing 8.2. Zawartość pliku *_ViewImports.cshtml* w katalogu *Views*

```
@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*.

Tworzenie projektu testów jednostkowych

Utworzenie projektu testów jednostkowych wymaga przeprowadzenia dokładnie tego samego procesu, który przedstawiłem w rozdziale 7. Prawym przyciskiem myszy kliknij element *Rozwiązanie SportsStore* w oknie *Eksplorator rozwiązań* i wybierz opcję *Dodaj/Nowy projekt...* Zaznacz szablon *Projekt testów xUnit (.NET Core)*, jak pokazałem na rysunku 8.4 i jako nazwę nowego projektu podaj *SportsStore.Tests*. Kliknij przycisk *OK*, co spowoduje utworzenie projektu testów jednostkowych.



Rysunek 8.4. Utworzenie projektu testów jednostkowych

Gdy projekt testów jednostkowych zostanie już utworzony, prawym przyciskiem myszy kliknij projekt *SportsStore.Tests* w oknie *Eksplorator rozwiązań*, a następnie z menu kontekstowego wybierz opcję *Edytuj element SportsStore.Tests*. Wprowadź w pliku zmiany przedstawione na listingu 8.3. Modyfikacje polegają na dodaniu pakietu *Moq* wymaganego do wykonywania testów oraz na zdefiniowaniu odwołania do głównego projektu aplikacji *SportsStore*. Upewnij się o podaniu wersji pakietu *Moq* pokazanej na listingu.

Listing 8.3. Dodanie pakietu w pliku *SportsStore.Tests.csproj* w projekcie testów jednostkowych

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="..\SportsStore\SportsStore.csproj" />
  </ItemGroup>
```

```

<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
  <PackageReference Include="Moq" Version="4.7.99" />
</ItemGroup>

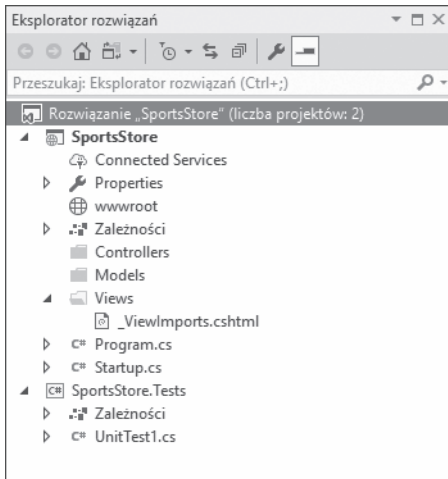
</Project>

```

Po zapisaniu zmian w pliku `.csproj` Visual Studio pobierze i zainstaluje pakiet Moq w projekcie testów jednostkowych. Ponadto zostanie utworzone odwołanie do głównego projektu SportsStore, aby znajdujące się w nim klasy mogły być używane w testach.

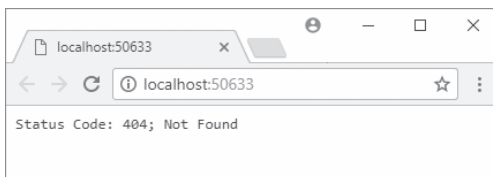
Sprawdzenie i uruchomienie aplikacji

Projekty aplikacji i testów jednostkowych są już utworzone, skonfigurowane i gotowe do rozpoczęcia pracy nad nimi. Okno *Eksplorator rozwiązań* powinno zawierać elementy pokazane na rysunku 8.5. Będziesz mieć problemy, jeśli widzisz inne elementy bądź nie znajdują się one w tych samych lokalizacjach, jak pokazałem na rysunku. Dlatego też poświęć chwilę i sprawdź, czy wszystko znajduje się na swoim miejscu.



Rysunek 8.5. Okno Eksplorator rozwiązań dla projektów aplikacji i testów jednostkowych

Jeżeli wybierzesz opcję *Rozpocznij debugowanie* z menu *Debugowanie* (lub *Uruchom bez debugowania*, o ile preferujesz iteracyjny styl programowania, który przedstawiłem w rozdziale 6.), wyświetli się strona z informacją o błędzie, widoczna na rysunku 8.6. Dzieje się tak, ponieważ zażądałeś wyświetlenia adresu URL skojarzonego z nieistniejącym kontrolerem. Rozwiązaniem tego problemu zajmiemy się już za chwilę.



Rysunek 8.6. Strona z informacją o błędzie wyświetlanym po uruchomieniu aplikacji

Tworzenie modelu domeny

Wszystkie projekty MVC zaczynają się od modelu domeny (tak naprawdę wszystko na platformie obraca się wokół modelu domeny). 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 8.4.

Listing 8.4. Zawartość pliku *Product.cs* w katalogu *Models*

```
namespace SportsStore.Models
{
    public class Product
    {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
    }
}
```

Tworzenie repozytorium

Wiemy, że potrzebny będzie mechanizm pozwalający na pobieranie obiektów *Product* z bazy danych. Zgodnie z informacjami przedstawionymi w rozdziale 3., model zawiera logikę przeznaczoną do przechowywania i pobierania danych z trwałego magazynu danych. Nie musimy się teraz przejmować, w jaki sposób cały silnik dostępu do danych będzie realizował swoje zadanie, wystarczy, że zdefiniujemy dla niego interfejs. W katalogu *Models* utwórz nowy plik interfejsu o nazwie *IProductRepository.cs*, którego zawartość jest zamieszczona na listingu 8.5.

Listing 8.5. Zawartość pliku *IProductRepository.cs* w katalogu *Models*

```
using System.Linq;

namespace SportsStore.Models
{
    public interface IProductRepository
    {
        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 *IProductRepository* 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.

Tworzenie imitacji repozytorium

Mamy już zdefiniowany interfejs, więc możemy zaimplementować mechanizm trwałego magazynu danych i dołączyć go do bazy danych. Jednak wcześniej chciałbym dodać inne komponenty aplikacji. Aby można było rozpocząć prace nad dalszymi częściami aplikacji, utworzymy imitację implementacji interfejsu `IProductRepository`, który będzie używany aż do chwili, gdy powrócimy do tematu magazynu danych. W celu utworzenia imitacji repozytorium, do katalogu *Models* dodaj nowy plik klasy o nazwie *FakeProductRepository.cs* i zdefiniuj w nim klasę przedstawioną na listingu 8.6.

Listing 8.6. Zawartość pliku *FakeProductRepository.cs* w katalogu *Models*

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models
{
    public class FakeProductRepository : IProductRepository
    {
        public IQueryable<Product> Products => new List<Product> {
            new Product { Name = "Piłka nożna", Price = 25 },
            new Product { Name = "Deska surfingowa", Price = 179 },
            new Product { Name = "Buty do biegania", Price = 95 }
        }.AsQueryable<Product>();
    }
}
```

Klasa `FakeProductRepository` implementuje interfejs `IProductRepository` i jako wartość właściwości `Products` zwraca stałej wielkości kolekcję obiektów `Product`. Metoda `AsQueryable()` jest używana w celu przeprowadzenia konwersji kolekcji obiektów na postać typu `IQueryable<T>` wymaganą do zaimplementowania interfejsu `IProductRepository`. W ten sposób można utworzyć odpowiednią imitację repozytorium bez konieczności zajmowania się obsługą rzeczywistych zapytań.

Rejestrowanie usługi repozytorium

W aplikacji MVC duży nacisk jest położony na *luźne powiązanie komponentów ze sobą*. Oznacza to możliwość wprowadzenia zmiany w jednej części aplikacji bez konieczności przeprowadzania modyfikacji jej innych fragmentów. Tego rodzaju podejście kategoryzuje fragmenty aplikacji jako *usługi* dostarczające funkcjonalności innym fragmentom aplikacji. Klasa dostarczająca usługę może być zmieniona lub zastąpiona bez konieczności wprowadzania zmian w klasach korzystających z danej usługi. Dokładnie ten temat wyjaśnię w rozdziale 18. Na potrzeby budowanej tutaj aplikacji *SportsStore* chcę utworzyć usługę

repozytorium pozwalającą kontrolerom na pobieranie obiektów implementujących interfejs `IProductRepository` bez konieczności posiadania jakichkolwiek informacji o używanej klasie. W ten sposób można rozpocząć pracę nad aplikacją, używając prostej klasy `FakeProductRepository` utworzonej w poprzedniej sekcji. Później tę klasę będzie można zastąpić rzeczywistym repozytorium, bez konieczności wprowadzania jakichkolwiek zmian w klasach uzyskujących dostęp do tego repozytorium. Rejestrowanie usług odbywa się za pomocą metody `ConfigureServices()` klasy `Startup`. Na listingu 8.7 pokazałem definicję nowej usługi dla repozytorium.

Listing 8.7. Utworzenie usługi repozytorium 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 SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<IProductRepository,
                FakeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {

            });
        }
    }
}
```

Polecenie dodane do metody `ConfigureServices()` wskazuje platformie ASP.NET, że kiedy komponent taki jak kontroler wymaga implementacji interfejsu `IProductRepository`, wówczas powinien otrzymać egzemplarz klasy `FakeProductRepository`. Zadaniem metody `AddTransient()` jest określenie, że nowy obiekt typu `FakeProductRepository` powinien zostać utworzony za każdym razem, gdy jest potrzebny interfejs `IProductRepository`. Nie przejmuj się, jeśli w tym momencie to wszystko nie ma dla Ciebie sensu. Wkrótce zobaczysz, jak tego rodzaju rozwiązanie funkcjonuje w aplikacji. Więcej informacji na ten temat przedstawię w rozdziale 18.

Wyświetlanie listy produktów

Moglibyśmy spędzić cały dzień na dodawaniu funkcji i zachowań do modelu domeny, nie korzystając wcale z projektu interfejsu użytkownika. Uważam jednak, że jest to nudne, więc zmienimy kierunek

i zaczniemy korzystać z frameworka MVC. Będziemy dodawać funkcje do modelu i repozytorium, gdy będziemy ich potrzebować.

W podrozdziale tym utworzymy kontroler i metodę akcji pozwalającą wyświetlić dane produktu z repozytorium. Na razie będzie ona korzystała z imitacji repozytorium, ale problemem tym zajmiemy się nieco później. Utworzymy również początkową *konfigurację routingu*, dzięki czemu MVC będzie w stanie przekazywać żądania do tworzonych przez nas kontrolerów.

Użycie szkieletu MVC oferowanego przez Visual Studio

W tej książce kontrolery i widoki w aplikacji MVC tworzę, klikając prawym przyciskiem myszy katalog w oknie *Eksploreator rozwiązań* i wybierając opcję *Dodaj/Nowy element...* z menu kontekstowego, a następnie wskazuję odpowiedni szablon elementu w wyświetlonym oknie dialogowym *Dodaj nowy element*. Istnieje podejście alternatywne określane mianem *szkieletu* (ang. *scaffolding*), w którym Visual Studio oferuje w menu *Dodaj* elementy przeznaczone specjalnie do tworzenia kontrolerów i widoków. Po wybraniu tego rodzaju elementu zostaniesz poproszony o wskazanie scenariusza dla tworzonego komponentu, na przykład kontroler wraz z akcjami odczytu i zapisu lub widok zawierający formularz przeznaczony do wygenerowania określonego obiektu modelu.

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 napotykanych problemów programistycznych. Moim celem w książce jest nie tylko upewnienie się, że potrafisz utworzyć aplikację opartą na frameworku MVC, ale również dokładne wyjaśnienie sposobu jej działania. Ten cel stanie się trudniejszy do zrealizowania, gdy odpowiedzialność za tworzenie komponentów zostanie zrzuczona na Visual Studio.

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. Wprawdzie 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.

Dodawanie kontrolera

W celu utworzenia pierwszego kontrolera aplikacji dodaj do katalogu *Controllers* nowy plik klasy o nazwie *ProductController.cs* i umieść w nim kod przedstawiony na listingu 8.8.

Listing 8.8. Zawartość pliku *ProductController.cs* w katalogu *Controllers*

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

namespace SportsStore.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;

        public ProductController(IProductRepository repo)
        {
            repository = repo;
        }
    }
}
```

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

To nosi nazwę *wstrzyknięcia zależności* (ang. *dependency injection*). Ten mechanizm pozwala obiektowi typu `ProductController` na uzyskanie dostępu do repozytorium aplikacji za pomocą interfejsu `IProductRepository` bez konieczności ustalenia, która implementacja klasy została skonfigurowana. Na późniejszym etapie pracy nad aplikacją imitację repozytorium zastąpimy prawdziwym repozytorium, 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 18.

Kolejnym krokiem jest dodanie do kontrolera `ProductController` metody akcji o nazwie `List()` odpowiedzialnej za wygenerowanie widoku wyświetlającego pełną listę produktów znajdujących się w repozytorium. Zmianę konieczną do wprowadzenia w pliku `ProductController.cs` przedstawiłem na listingu 8.9.

Listing 8.9. Dodawanie metody akcji w pliku `ProductController.cs` w katalogu `Controllers`

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

namespace SportsStore.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;

        public ProductController(IProductRepository repo)
        {
            repository = repo;
        }

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

Wywołanie w ten sposób metody `View()`, czyli bez podawania nazwy widoku, informuje framework, że powinna wygenerować domyślny szablon widoku dla metody akcji. Przez przekazanie listy obiektów `Product` do metody `View()` informujemy framework, że wypełniliśmy obiekt `Model` w widoku o ściśle określonym typie.

Dodawanie i konfigurowanie widoku

Teraz trzeba utworzyć widok pozwalający na wyświetlenie treści użytkownikowi. Jednak wcześniej konieczne są pewne przygotowania, dzięki którym tworzenie widoków stanie się prostsze. Pracę zaczniemy od przygotowania współdzielonego układu definiującego treść umieszczaną we wszystkich odpowiedziach

HTML wysyłanych do klienta. Wspomniany współdzielony układ to użyteczny sposób na zagwarantowanie, że widoki pozostaną spójne oraz będą zawierały odwołania do ważnych skryptów JavaScript i arkuszy stylów CSS. Więcej informacji na temat współdzielonego układu przedstawiłem w rozdziale 5.

Utwórz katalog *Views/Shared* i umieść w nim stronę układu widoku MVC o nazwie *_Layout.cshtml*. To jest nazwa domyślna stosowana przez Visual Studio dla tego rodzaju pliku. Na listingu 8.10 przedstawiłem zawartość naszego pliku *_Layout.cshtml*. Wprowadziłem tylko jedną zmianę w pierwotnej zawartości pliku: element `<title>` wyświetla nazwę budowanej aplikacji, czyli *SportsStore*.

Listing 8.10. Zawartość pliku *_Layout.cshtml* w katalogu *Views/Shared*

```
<!DOCTYPE html>

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

Kolejnym krokiem jest skonfigurowanie aplikacji, aby plik układu *_Layout.cshtml* był stosowany domyślnie. Odbyna się to za pomocą pliku typu MVC — *wyświetl stronę startową* o nazwie *_ViewStart.cshtml* utworzonego w katalogu *Views*. Zawartość domyślną wymienionego pliku, umieszczoną w nim przez Visual Studio, przedstawiłem na listingu 8.11. Działanie tego kodu polega na zastosowaniu pliku układu *_Layout.cshtml* odpowiadającego plikowi, którego zawartość przedstawiłem wcześniej, na listingu 8.10.

Listing 8.11. Domyślna zawartość pliku *_ViewStart.cshtml* w katalogu *Views*

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

Teraz musimy dodać widok, który będzie wyświetlany po wywołaniu metody akcji `List()` w celu obsługi żądania. Utwórz katalog *Views/Product* i dodaj do niego plik widoku Razor o nazwie *List.cshtml*. Następnie w nowym pliku widoku umieść kod znaczników przedstawiony na listingu 8.12.

Listing 8.12. Zawartość pliku *List.cshtml* w katalogu *Views/Product*

```
@model IEnumerable<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. Takie podejście pozostaje zgodne z omówioną w rozdziale 3. zasadą separacji zadań.

-
- **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`.
-

Konfigurowanie trasy domyślnej

Musimy teraz poinformować framework MVC, że żądania dotyczące katalogu głównego witryny (`http://nasza_witryna/`) powinny być przekazane do metody akcji `List()` z klasy `ProductController`. Możemy to zrobić przez edycję polecenia w klasie `Startup` konfigurującego klasy przeznaczone do obsługi żądań HTTP. Zmiany konieczne do wprowadzenia w pliku `Startup.cs` pokazałem na listingu 8.13.

Listing 8.13. Zdefiniowanie w pliku `Startup.cs` trasy domyślnej

```
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 SportsStore.Models;

namespace SportsStore
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddTransient<IProductRepository,
                FakeProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}
```

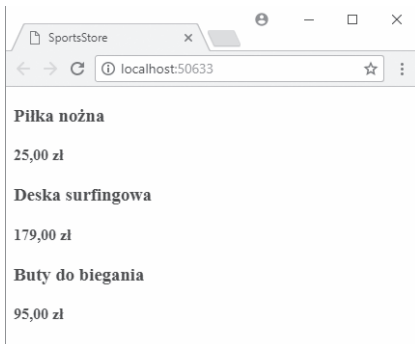
Metoda `Configure()` klasy `Startup` jest używana do skonfigurowania potoku żądania, na który składają się klasy (określane mianem *oprogramowania pośredniczącego*, ang. *middleware*) odpowiedzialne

za przeanalizowanie żądania HTTP i wygenerowanie odpowiedzi. Metoda `UseMvc()` konfiguruje oprogramowanie pośredniczące MVC, a jedna z opcji konfiguracyjnych dotyczy schematu używanego podczas mapowania adresów URL na kontrolery i metody akcji. System routingu na platformie ASP.NET przedstawię dokładniej w rozdziałach 15. i 16. Na razie wystarczy wiedzieć, że zmiana pokazana na listingu 8.13 powoduje kierowanie żądań domyślnego adresu URL do zdefiniowanej przez nas metody akcji, w omawianym przykładzie to `List()` w kontrolerze `ProductController`.

-
- **Wskazówka** Zwróć uwagę, że wartością właściwości `controller` na listingu 8.13 jest `Product`, a nie nazwa klasy `ProductController`. Jest to obowiązkowy schemat nazewnictwa ASP.NET MVC, w którym klasy kontrolerów kończą się na `Controller`; przy odwołaniu do klasy pomijamy tę część nazwy. W rozdziale 31. dokładnie wyjaśnię konwencję nazw i efekt jej stosowania.
-

Uruchamianie aplikacji

Podstawowe mechanizmy są gotowe. Mamy kontroler z metodą akcji, która jest wywoływana przez framework MVC w momencie zażądania domyślnego adresu URL. Framework MVC tworzy egzemplarz klasy `FakeRepository` i używa go do utworzenia nowego obiektu kontrolera przeznaczonego do obsługi żądania. Imitacja repozytorium dostarcza kontrolerowi pewne przykładowe dane testowe, które metoda akcji przekazuje widokowi Razor, aby odpowiedź HTML wygenerowana w przeglądarce WWW zawierała informacje szczegółowe o poszczególnych produktach. W trakcie generowania odpowiedzi framework MVC łączy dane widoku wybrane przez metodę akcji z danymi we współdzielonym układzie. W ten sposób powstaje kompletny dokument HTML, który przeglądarka WWW może przetworzyć i wyświetlić. Wynik otrzymany po uruchomieniu aplikacji pokazałem na rysunku 8.7.



Rysunek 8.7. Podstawowe funkcje aplikacji

Wzorzec używany przy tworzeniu tej aplikacji jest typowy dla frameworka ASP.NET Core MVC. Inwestujemy relatywnie dużo czasu na skonfigurowanie wszystkich elementów, ale za to bardzo szybko powstają podstawowe funkcje aplikacji.

Przygotowanie bazy danych

Możemy już wyświetlić prosty widok zawierający dane naszych produktów, ale nadal są to dane testowe zwracane przez imitację repozytorium. Zanim zbudujemy rzeczywiste repozytorium, musimy skonfigurować bazę danych i wypełnić ją danymi.

Jako bazy danych użyjemy SQL Server. 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#.

-
- **Uwaga** Jest to kolejny obszar, w którym możesz wybierać z wielu narzędzi i technologii. Można korzystać nie tylko z wielu relacyjnych baz danych, ale również z repozytoriów obiektów, magazynów dokumentów oraz kilku egzotycznych odpowiedników. Dla .NET dostępnych jest też wiele frameworków ORM, z których każda przyjmuje nieco inne podejście — któraś z nich może pasować do Twojego projektu.
-

Entity Framework wybrałem z kilku powodów. Pierwszym jest łatwość konfiguracji i wykorzystywania tego frameworka. Drugim jest pierwszorzędną integracją z LINQ, a ja lubię używać LINQ. Trzeci powód jest taki, że ten framework bardzo dobrze współpracuje z ASP.NET Core MVC — we wcześniejszych wersjach występowały wprawdzie problemy, ale bieżąca jest bardzo elegancka i ma duże możliwości.

Jedną z użytecznych funkcji w Visual Studio i SQL Server jest *LocalDB*, czyli pozbawiona funkcji administracyjnych implementacja podstawowych funkcji SQL Server przeznaczonych specjalnie na potrzeby programistów. Dzięki LocalDB można pominąć proces konfiguracji bazy danych podczas budowy projektu, a następnie dodać pełny egzemplarz SQL Server w trakcie wdrażania projektu. Większość aplikacji ASP.NET Core MVC jest wdrażana w środowiskach obsługiwanych przez profesjonalnych administratorów. Dzięki wspomnianej funkcji LocalDB zadanie konfiguracji bazy danych pozostaje w rękach administratorów baz danych, natomiast programiści zajmują się tworzeniem kodu.

-
- **Wskazówka** Jeżeli podczas instalacji Visual Studio nie wybrałeś opcji *LocalDB*, będziesz musiał zrobić to teraz. To jest fragment opcji dotyczących narzędzi danych, ewentualnie może być też zainstalowany jako część SQL Server. Jeśli zastosowałeś się do wskazówek przedstawionych w rozdziale 2., funkcja LocalDB powinna być już zainstalowana i gotowa do użycia.
-

Instalowanie pakietu narzędzi Entity Framework Core

Podstawowa funkcjonalność Entity Framework Core jest przez Visual Studio dodawana domyślnie w trakcie tworzenia nowego projektu. Potrzebny będzie jeden pakiet dodatkowy w celu dostarczenia działających w wierszu poleceń narzędzi używanych podczas tworzenia klas przygotowujących bazę danych do przechowywania danych aplikacji, znanych pod nazwą *migracji*.

Aby dodać pakiet do projektu, prawym przyciskiem myszy kliknij projekt *SportsStore.Tests* w oknie *Eksplorator rozwiązań*, a następnie z menu kontekstowego wybierz opcję *Edytuj element SportsStore.Tests*. Wprowadź w pliku zmiany przedstawione na listingu 8.14. Zwróć uwagę na wersję pakietu na listingu, a także zauważ, że nowy pakiet jest dodawany za pomocą elementu `<DotNetCliToolReference>`, a nie `<PackageReference>`, jak miało to miejsce wcześniej w przypadku istniejącego pakietu.

-
- **Uwaga** Pakiet trzeba zainstalować poprzez edycję wymienionego pliku. Tego rodzaju pakiet nie może być dodany za pomocą Menedżera pakietów NuGet lub narzędzia wiersza polecenia `dotnet`.
-

Listing 8.14. Dodanie pakietu w pliku *SportsStore.csproj* w projekcie *SportsStore*

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
```

```

<ItemGroup>
  <Folder Include="wwwroot\" />
</ItemGroup>

<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version="2.0.0" />
</ItemGroup>

</Project>

```

Po zapisaniu zmian wprowadzonych w pliku Visual Studio pobierze i zainstaluje narzędzia EF Core oraz dołączy je do projektu.

Utworzenie klas bazy danych

Klasa kontekstu bazy danych jest pomostem między aplikacją i frameworkiem EF Core, a jej działanie polega na zapewnieniu dostępu do danych aplikacji z użyciem obiektów modelu. W celu utworzenia klasy kontekstu bazy danych dla aplikacji SportsStore, do katalogu *Models* trzeba dodać nowy plik klasy o nazwie *ApplicationDbContext.cs*, a następnie zdefiniować w nim klasę przedstawioną na listingu 8.15.

Listing 8.15. Zawartość pliku *ApplicationDbContext.cs* w katalogu *Models*

```

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.DependencyInjection;

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

Utworzenie klasy repozytorium

W tym momencie to może nie być jeszcze aż tak oczywiste, ale wykonaliśmy już większość pracy niezbędnej do przygotowania bazy danych. Kolejnym krokiem jest utworzenie klasy implementującej interfejs *IProductRepository* i pobierającej dane za pomocą Entity Framework Core. Do katalogu *Models* należy dodać nowy plik klasy o nazwie *EFProductRepository.cs* i użyć go do zdefiniowania klasy repozytorium przedstawionej na listingu 8.16.

Listing 8.16. Zawartość pliku *EFProductRepository.cs* w katalogu *Models*

```

using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models
{
    public class EFProductRepository : IProductRepository
    {
        private ApplicationDbContext context;

        public EFProductRepository(ApplicationDbContext ctx)
        {
            context = ctx;
        }

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

```

Funkcjonalność będziemy dodawać wraz z implementacją kolejnych funkcji w aplikacji. Na obecnym etapie implementacja repozytorium po prostu mapuje właściwość *Products* zdefiniowaną przez interfejs *IProductRepository* na właściwość *Products* zdefiniowaną przez klasę *ApplicationDbContext*. Właściwość *Products* w klasie kontekstu zwraca obiekt typu *DbSet<Product>* implementujący interfejs *IQueryable<T>* i ułatwiający zaimplementowanie interfejsu *IProductRepository* podczas użycia Entity Framework Core. Dzięki temu mamy pewność, że zapytania wykonywane do bazy danych będą pobierały jedynie wymagane obiekty, jak to wyjaśniłem nieco wcześniej w rozdziale.

Definiowanie ciągu tekstowego połączenia

Tak zwany *ciąg tekstowy zapytania* określa położenie i nazwę bazy danych, a także zapewnia ustawienia konfiguracyjne wskazujące aplikacji sposób nawiązania połączenia z serwerem bazy danych. Ciągi tekstowe połączenia są przechowywane w pliku JSON o nazwie *appsettings.json*, który w projekcie *SportsStore* został utworzony za pomocą szablonu *Plik konfiguracji ASP.NET* w sekcji *ASP.NET Core/Sieć Web/Ogólne* okna dialogowego *Dodaj nowy element*.

Podczas tworzenia pliku *appsettings.json* Visual Studio dodaje do niego miejsce dla ciągu tekstowego połączenia. Pierwotną zawartość utworzonego pliku zastąp przedstawioną na listingu 8.17.

Listing 8.17. Przykład ciągu tekstowego połączenia w pliku *appsettings.json* w projekcie *SportsStore*

```

{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;
        Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}

```

-
- **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 8.17. Podczas definiowania ciągu tekstowego połączenia we własnym projekcie upewnij się, że wartość *ConnectionString* znajduje się w jednym wierszu.
-

W sekcji Data pliku konfiguracyjnego zdefiniowałem nazwę ciągu tekstowego połączenia (SportsStoreProducts). Wartość elementu ConnectionString określa, że funkcja *LocalDB* powinna zostać użyta dla bazy danych o nazwie SportsStore.

Konfigurowanie aplikacji

Kolejne kroki to odczyt ciągu tekstowego połączenia oraz skonfigurowanie aplikacji do jego użycia podczas nawiązywania połączenia z bazą danych. Na listingu 8.18 przedstawiłem zmiany konieczne do wprowadzenia w klasie Startup, aby można było odczytać ciąg tekstowy połączenia z pliku konfiguracyjnego i przygotować EF Core. (Odczytywanie pliku w formacie JSON to zadanie obsługiwane przez klasę Program, którą omówię dokładnie w rozdziale 14.).

Listing 8.18. 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.Configuration;
using Microsoft.EntityFrameworkCore;

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}
```

Do klasy `Startup` został dodany konstruktor wczytujący ustawienia konfiguracyjne z pliku `appsettings.json` i udostępniający je poprzez obiekt implementujący interfejs `IConfiguration`. Ten konstruktor przypisuje wspomniany obiekt właściwości o nazwie `Configuration`, aby mógł być używany w pozostałej części klasy `Startup`.

Temat odczytywania i uzyskiwania dostępu do danych konfiguracji omówię dokładnie w rozdziale 14. W metodzie `ConfigureServices()` dodałem sekwencję wywołań metod odpowiedzialną za konfigurację `Entity Framework Core`.

```
...
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        Configuration["Data:SportStoreProducts:ConnectionString"]));
...
```

Metoda rozszerzenia `AddDbContext()` przygotowuje usługi dostarczane przez `Entity Framework Core` do użycia wraz z klasą kontekstu bazy danych, którą utworzyłem na listingu 8.15. Jak wyjaśniłem w rozdziale 14., wiele metod użytych w klasie `Startup` pozwala na konfigurowanie usług i oprogramowania pośredniczącego za pomocą opcji argumentów. Argumentem metody `AddDbContext()` jest wyrażenie lambda otrzymujące obiekt opcji, który skonfiguruje bazę danych pod kątem klasy kontekstu. W omawianym przykładzie skonfigurowałem bazę danych za pomocą metody `UseSqlServer()` i określiłem ciąg tekstowy połączenia otrzymany z właściwości `Configuration`.

Następną zmianą wprowadzoną w klasie `Startup` było zastąpienie imitacji repozytorium rzeczywistym repozytorium, jak pokażę poniżej.

```
...
services.AddTransient<IProductRepository, EFProductRepository>();
...
```

Komponenty w aplikacji używające interfejsu `IProductRepository`, który w tym momencie jest po prostu kontrolerem `Product`, będą w chwili tworzenia otrzymywały obiekt `EFProductRepository` zapewniający dostęp do informacji w bazie danych. Ten mechanizm szczegółowo omówię w rozdziale 18. Efektem jest zastąpienie danych imitacji rzeczywistymi danymi pochodzącymi z bazy danych, nie trzeba przy tym wprowadzać żadnych modyfikacji w klasie `ProductController`.

Wyłączenie weryfikacji zakresu

Użycie `Entity Framework Core` wymaga pewnej zmiany konfiguracyjnej mechanizmu wstrzykiwania zależności, który dokładnie omówię w rozdziale 18. Klasa `Program` jest odpowiedzialna za uruchomienie i konfigurację `ASP.NET Core MVC`, zanim kontrola nad aplikacją zostanie przekazana klasie `Startup`. Na listingu 8.19 przedstawiłem zmianę konieczną do wprowadzenia. Bez tej zmiany nastąpi zgłoszenie wyjątku, gdy w następnej sekcji spróbujesz utworzyć schemat bazy danych.

Listing 8.19. Przygotowanie `Entity Framework Core` w pliku `Program.cs` w projekcie `SportsStore`

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace SportsStore
{
    public class Program
```

```

{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseDefaultServiceProvider(options =>
                options.ValidateScopes = false)
            .Build();
}
}

```

Konfigurację ASP.NET Core szczegółowo omówię w rozdziale 14. Powyżej przedstawiłem jedyną zmianę klasy Program wymaganą do zastosowania w aplikacji SportsStore.

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, przejdź do katalogu projektu SportsStore (czyli katalogu zawierającego pliki *Startup.cs* i *appsettings.json*), a następnie wydaj poniższe polecenie w celu wygenerowania klasy migracji, która przygotowuje bazę danych do jej pierwszego użycia.

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

Po wykonaniu powyższego polecenia zobaczysz w oknie *Eksplorator rozwiązań* w Visual Studio nowy katalog o nazwie *Migrations*. W tym katalogu Entity Framework Core przechowuje klasy migracji. Jeden z plików będzie miał nazwę w postaci długiego numeru zakończonym 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, dodany do projektu na listingu 8.14. Jedno z nowych poleceń wykorzystałem 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 8.20.

Listing 8.20. Zawartość pliku *SeedData.cs* w katalogu *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)
        {
            ApplicationDbContext context = app.ApplicationServices
                .GetRequiredService<ApplicationDbContext>();
            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. Ma to zagwarantować, że baza danych będzie miała pewną zawartość.

Metoda `EnsurePopulated()` pobiera obiekt `ApplicationContext` 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 jakiegokolwiek 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. Odbyna się to za pomocą wywołania metody `SeedData.EnsurePopulated()` w klasie `Startup`, jak pokazałem na listingu 8.21.

Listing 8.21. Umieszczenie danych początkowych w bazie danych za pomocą wywołania `EnsulePopulated()` 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 SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

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

        public IConfiguration Configuration { get; }

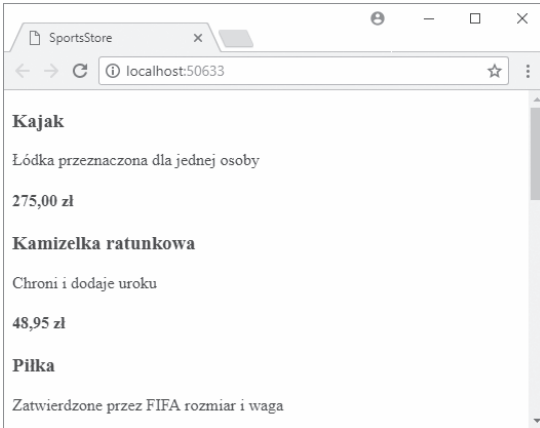
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
            SeedData.EnsurePopulated(app);
        }
    }
}

```

Uruchom aplikację. Nastąpi utworzenie bazy danych i umieszczenie w niej niezbędnych informacji, aby zapewnić aplikacji wymagane przez nią dane. (Bądź cierpliwy, ponieważ utworzenie bazy danych może zabrać dłuższą chwilę).

Kiedy przeglądarka WWW zażąda domyślnego adresu URL aplikacji, konfiguracja programu nakaze frameworkowi MVC utworzenie kontrolera `ProductController` przeznaczonego do obsługi żądania. Utworzenie tego kontrolera oznacza wywołanie konstruktora klasy `ProductController`, który wymaga obiektu implementującego interfejs `IProductRepository`. Nowa konfiguracja informuje MVC, że w tym celu powinien zostać utworzony i użyty obiekt typu `EFProductRepository`. Ten obiekt wykorzystuje funkcjonalność EF Core do wczytania danych relacyjnych z serwera SQL Server oraz ich konwersji na obiekty `Product`. Cała ta funkcjonalność zostaje ukryta przed klasą `ProductController`, która po prostu otrzymuje obiekt implementujący interfejs `IProductRepository` i pracuje z otrzymanymi danymi. Wynikiem jest to, że przeglądarka WWW wyświetla przykładowe dane pochodzące z bazy danych, jak pokazałem na rysunku 8.8.



Rysunek 8.8. Użycie repozytorium w postaci bazy danych

Ten sposób przedstawienia Entity Framework bazy danych SQL Server jako serii obiektów modelu jest prosty i łatwy, a ponadto pozwala nam skoncentrować się na platformie ASP.NET Core MVC. Oczywiście pominąłem tutaj wiele informacji szczegółowych dotyczących sposobu działania frameworka Entity Framework oraz ogromną liczbę dostępnych opcji konfiguracyjnych. Bardzo lubię Entity Framework i zachęcam Cię do poświęcenia nieco czasu na jej dokładniejsze poznanie. Dobrym punktem do rozpoczęcia poznawania platformy jest poświęcona Entity Framework strona firmy Microsoft, którą znajdziesz pod adresem <https://docs.microsoft.com/pl-pl/ef/>, oraz moja książka o Entity Framework Core, która wkrótce zostanie wydana przez Apress.

Dodanie stronicowania

Jak widać na rysunku 8.8, wszystkie dane produktów pobrane z bazy danych są wyświetlane na jednej stronie. 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 metody `List()` w kontrolerze `ProductController` (listing 8.22).

Listing 8.22. Dodawanie stronicowania w metodzie `List()` kontrolera `ProductController`

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

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

        public ProductController(IProductRepository repo)
        {
            repository = repo;
        }

        public IActionResult List(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 `List()` dodałem *parametr opcjonalny*. Dzięki temu, gdy wywołamy metodę bez parametru (`List()`), nasze wywołanie będzie traktowane tak, jakbyśmy podali wartość określoną w definicji parametru (`List(1)`). W efekcie metoda akcji powoduje wyświetlenie pierwszej strony produktów, gdy framework MVC wywołuje tę metodę bez argumentu. W metodzie `List()` 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, wstrzykniemy ją do konstruktora klasy `ProductController`, a następnie wywołamy metodę `List()` dla określonej strony. Następnie porównamy obiekty `Product`, jakie otrzymamy, z tymi, których oczekiwaliśmy. Informacje na temat konfigurowania testów jednostkowych znajdziesz w rozdziale 7. Poniżej znajduje się test, jaki utworzyłem w pliku klasy o nazwie *ProductControllerTests.cs* w projekcie *SportsStore.Tests*.

```

using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests
{
    public class ProductControllerTests
    {
        [Fact]
        public void Can_Paginate()
        {
            // Przygotowanie.
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            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>();

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

            // Działanie.
            IEnumerable<Product> result =
                controller.List(2).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 łatwo dostać się do danych zwróconych z metody akcji kontrolera. Wynikiem jest obiekt `ViewResult`, wartość jego właściwości `ViewData.Model` trzeba rzutować na oczekiwany typ danych. W rozdziale 17. przedstawię różne typy wyników zwracanych przez metody akcji oraz pokażę, jak można z nimi pracować.

Wyświetlanie łączy stron

Jeżeli uruchomisz aplikację, zauważysz tylko cztery produkty na jednej stronie. Jeżeli chcesz zobaczyć inną stronę, możesz dodać do adresu URL parametr:

```
http://localhost:50633/?productPage=2
```

Prawdopodobnie będziesz musiał zmienić numer portu w tym adresie URL, aby pasował do tego, na którym działa Twój serwer ASP.NET. 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 udałoby się ich o tym poinformować, nie byłiby 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 modelu widoku, który będzie używany 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 8.23.

Listing 8.23. Zawartość pliku *PagingInfo.cs* w katalogu *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 8.24. Atrybuty pomocnicze znaczników to ważny dodatek do frameworka ASP.NET Core MVC. Więcej informacji na temat ich tworzenia i sposobu działania znajdziesz w rozdziałach od 23. do 25.

■ **Wskazówka** Katalogu *Infrastructure* używam do przechowywania klas zawierających te komponenty aplikacji, które nie są powiązane z domeną aplikacji.

Listing 8.24. Zawartość pliku klasy *PageLinkTagHelper.cs* w katalogu *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 MVC, takich jak kontrolery i widoki, może być odkrywanych automatycznie. Natomiast atrybuty pomocnicze znaczników muszą być rejestrowane. Na listingu 8.25 pokazałem nowe polecenie @addTagHelper dodane do pliku *_ViewImports.cshtml* w katalogu *Views*. To polecenie nakazuje wyszukanie klas atrybutów pomocniczych znaczników w przestrzeni nazw *SportsStore.Infrastructure*. 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 8.25. Zarejestrowanie atrybutu pomocniczego znacznika w pliku *_ViewImports.cshtml* w katalogu *Views/Shared*

```

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

```

Test jednostkowy — tworzenie łączy stron

Aby przetestować klasę atrybutu pomocniczego znacznika *PageLinkTagHelper*, wywołamy metodę *Process()* wraz z danymi testowymi i porównamy wyniki 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 apostrof. 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. Możemy zrealizować to za pomocą mechanizmu `View Bag`, ale lepszym rozwiązaniem jest opakowanie wszystkich danych wysyłanych z kontrolera do widoku pojedynczą klasą modelu 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 8.26.

Listing 8.26. Zawartość pliku *ProductsListViewModel.cs* w katalogu *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 `List()` w klasie `ProductController`, aby korzystała z klasy `ProductsListViewModel` do przekazania danych wyświetlanych produktów oraz informacji o stronicowaniu (listing 8.27).

Listing 8.27. Aktualizacja metody `List()` w pliku *ProductController.cs*

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

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

        public ProductController(IProductRepository repo)
        {
            repository = repo;
        }

        public IActionResult List(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 `ProductControllerTests` w projekcie testów:

```

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

    // Przygotowanie.
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    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.
    ProductController controller =
        new ProductController(mock.Object) {PageSize = 3};

    // Działanie.
    ProductsListViewModel result =
        controller.List(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 nasz wcześniejszy test stronicowania znajdujący się w metodzie `Can_Paginate()`. Zakłada on, że metoda akcji `List()` zwraca `ViewResult`, którego właściwość `Model` jest sekwencją obiektów `Product`, ale dane te umieściliśmy w innym typie modelu widoku. Zmieniony test wygląda następująco:

```

...
[Fact]
public void Can_Paginate()
{
    // Przygotowanie.
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    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>());

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

    // Działanie.
    ProductsListViewModel result =
        controller.List(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 samodzielne.

Teraz widok oczekuje sekwencji obiektów `Product`, więc aby obsłużyć nowy typ modelu, musimy jeszcze zmienić plik `List.cshtml`, jak pokazałem na listingu 8.28.

Listing 8.28. Zaktualizowany widok `List.cshtml` w katalogu `Views/Product`

```

@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 `List`. 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 8.29.

Listing 8.29. Dodanie do pliku `List.cshtml` łączy stronicowania

```

@model ProductsListViewModel

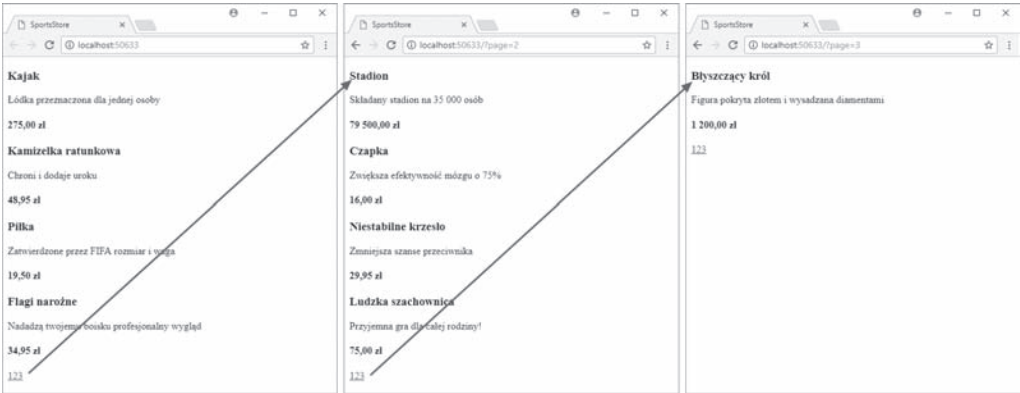
@foreach (var p in Model.Products)
{
    <div>
        <h3>@p.Name</h3>

```

```
@p.Description
<h4>@p.Price.ToString("c")</h4>
</div>
}
```

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

Po uruchomieniu aplikacji możesz zobaczyć łącza stron na dole strony (rysunek 8.9). 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 łącz widocznych na rysunku.



Rysunek 8.9. Wyświetlanie łączy nawigacji między stronami

Dlaczego po prostu nie użyjemy GridView?

Jeżeli korzystałeś wcześniej z ASP.NET, możesz uznać, że włożyliśmy sporo pracy, a uzyskaliśmy mało imponujące wyniki. Poświęciliśmy wiele miejsca, by uzyskać tylko listę stron. W przypadku Web Forms moglibyśmy uzyskać to samo przy użyciu gotowej kontrolki `GridView` lub `ListView` z ASP.NET Web Forms, dołączając ją bezpośrednio do naszej tabeli `Products`.

To, co uzyskaliśmy do tej pory, bardzo różni się jednak od przeciągnięcia `GridView` na formularz. Po pierwsze, budujemy tę aplikację na bazie solidnej architektury, która wymaga odpowiedniej separacji zadań. W przeciwieństwie do najprostszego wariantu użycia `ListView` nie mamy bezpośredniego powiązania interfejsu użytkownika z bazą danych — podejście to daje wynik najszybciej, ale z czasem sprawia bardzo dużo problemów. Po drugie, tworzyliśmy testy jednostkowe, które pozwalają nam kontrolować działanie naszej aplikacji w sposób, który jest niemal niemożliwy w przypadku użycia skomplikowanych kontrolki z Web Forms. Na koniec należy pamiętać, że spora część tego rozdziału została poświęcona tworzeniu bazowej infrastruktury, na podstawie której będzie budowana aplikacja. Musimy tylko raz zdefiniować repozytorium, a potem będziemy mogli szybko budować i testować nowe funkcje, co pokażę w kolejnych rozdziałach.

Wymienione punkty w żaden sposób nie pomniejszają faktu, że w przypadku Web Forms podobne wyniki można uzyskać po wykonaniu znacznie mniejszej ilości pracy. Jednak, jak wyjaśniłem w rozdziale 3., szybki wynik w Web Forms wiąże się z kosztem, który może być wysoki w ogromnych i skomplikowanych projektach.

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 MVC pozwala bardzo łatwo zmieniać schemat adresów URL, ponieważ wykorzystuje funkcje *routingu* ASP.NET. Wspomniany routing jest odpowiedzialny za przetwarzanie adresów URL i ustalenie docelowego fragmentu 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 8.30.

Listing 8.30. 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.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportsStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
```

```

        routes.MapRoute(
            name: "pagination",
            template: "Produkty/Strona{productPage}",
            defaults: new { Controller = "Product", action = "List" });

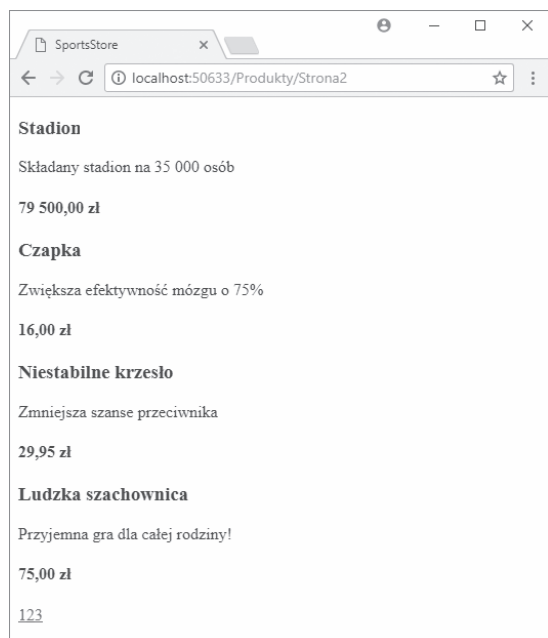
        routes.MapRoute(
            name: "default",
            template: "{controller=Product}/{action=List}/{id?}");
    });
    SeedData.EnsurePopulated(app);
}
}
}

```

Bardzo ważne jest, aby nowa trasa została dodana przed trasą domyślną (Default) zdefiniowaną w metodzie. Jak pokażę w rozdziale 15., 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 MVC 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). Nie przejmuj się, jeżeli nie wiesz, jak działa routing — wyjaśnię to szczegółowo w rozdziałach 15. i 16.

Jeżeli uruchomisz aplikację i przejdziesz do kolejnej strony, zobaczysz nowy schemat URL w działaniu (rysunek 8.10).



Rysunek 8.10. 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. Mam zamiar zaimplementować klasyczny, dwukolumnowy układ z nagłówkiem (rysunek 8.11).

Sports Store (nagłówek)	
Strona główna • Sporty wodne • Piłka nożna • Szachy • ...	• Produkt 1 • Produkt 2 • ... (główna treść)

Rysunek 8.11. Cel projektowy dla aplikacji SportsStore

Instalacja pakietu Bootstrap

W celu nadania stylów CSS w aplikacji wykorzystamy framework Bootstrap. Do zainstalowania pakietu Bootstrap wykorzystamy oferowaną przez Visual Studio obsługę Bower. W projekcie *SportsStore* utwórz nowy plik. W oknie dialogowym *Dodaj nowy element* zaznacz kategorię *ASP.NET Core/Sieć Web/Ogólne*, a następnie wybierz szablon *Plik konfiguracji programu Bower*, co spowoduje utworzenie pliku o nazwie *bower.json*, jak to pokazałem w rozdziale 6. Następnie w sekcji *dependencies* nowego pliku należy dodać pakiet Bootstrap. Zmiany konieczne do wprowadzenia w *bower.json* przedstawiłem na listingu 8.31. Jak już wcześniej wspomniałem, w przykładach przedstawionych w książce korzystam z wersji alfa frameworka Bootstrap CSS.

Listing 8.31. Dodanie pakietu Bootstrap do pliku *bower.json* w projekcie *SportsStore*

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.6"
  }
}
```

Po zapisaniu zmian w pliku *bower.json* Visual Studio użyje Bower do pobrania pakietu Bootstrap do katalogu *wwwroot/lib/bootstrap*. Zależnością frameworka Bootstrap jest biblioteka jQuery, która również zostanie automatycznie dodana do projektu.

Zastosowanie w aplikacji stylów Bootstrap

W rozdziale 5. wyjaśniłem, jak działają strony układu Razor oraz jak można je stosować. Plik *_ViewStart.cshtml* dodany do projektu na początku rozdziału określa, że układ domyślny został zdefiniowany w pliku o nazwie *_Layout.cshtml*. W tym pliku zastosujemy początkowe style Bootstrap, jak pokazałem na listingu 8.32.

Listing 8.32. Zastosowanie stylów Bootstrap CSS w pliku *_Layout.cshtml* w katalogu *Views/Shared*

```
<!DOCTYPE html>

<html>
```

```

<head>
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet"
        asp-href-include="lib/bootstrap/dist/**/*.min.css"
        asp-href-exclude="**/*-reboot*,**/*-grid*" />
  <title>SportsStore</title>
</head>
<body>
  <div class="navbar navbar-inverse bg-inverse" role="navigation">
    <a class="navbar-brand" href="#">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>

```

Znacznik `<link>` ma atrybuty `asp-href-include` i `asp-href-exclude` przedstawiające przykład wbudowanych klas atrybutu pomocniczego znacznika. W omawianym przykładzie atrybut pomocniczy znacznika wyszukuje wartość atrybutu i generuje znaczniki `<link>` dla wszystkich plików dopasowanych do podanej ścieżki dostępu (można w niej używać znaków wieloznacznych). To jest użyteczna funkcjonalność zapewniająca możliwość dodawania i usuwania plików w strukturze katalogu *wwwroot* bez obawy o uszkodzenie aplikacji. Jak wyjaśnię w rozdziale 25., należy zachować dużą ostrożność podczas stosowania znaków wieloznacznych wskazujących pliki, które mają zostać dopasowane.

Dodanie arkusza stylu Bootstrap CSS do układu oznacza możliwość użycia tych stylów w dowolnym widoku zbudowanym na bazie tego układu. Na listingu 8.33 pokazałem dodanie stylów w pliku *List.cshtml*.

Listing 8.33. Użycie Bootstrap w celu nadania stylów w pliku *List.cshtml* w katalogu *Views/Product*

```

@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="List" page-classes-enabled="true"
      page-class="btn" page-class-normal="btn-secondary"
      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 8.34.

Listing 8.34. Dodanie klas do elementów generowanych w pliku `PageLinkTagHelper.cs`

```
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 { page = 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);
}
}
}

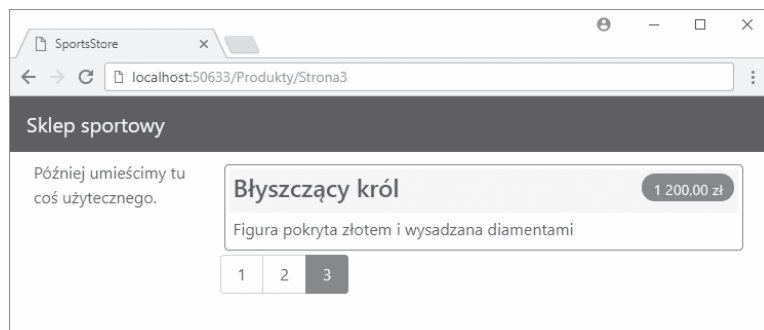
```

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 MVC.

Po uruchomieniu aplikacji zauważysz poprawę wyglądu — przynajmniej troszeczkę. Zmiany te są pokazane na rysunku 8.12.

Tworzenie widoku częściowego

Końcowym zadaniem w tym rozdziale będzie refaktoring naszej aplikacji — uprościmy widok *List.cshtml*. Utworzymy *widok częściowy*, to jest raczej fragment treści, który można dołączyć do innego widoku, a nie szablon. Widoki częściowe dokładnie omówię w rozdziale 21. 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 8.35.



Rysunek 8.12. Poprawiony układ graficzny aplikacji *SportsStore*

Listing 8.35. Zawartość pliku *ProductSummary.cshtml* w katalogu *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/List.cshtml*, aby korzystał z widoku częściowego. Zmiany są zamieszczone na listingu 8.36.

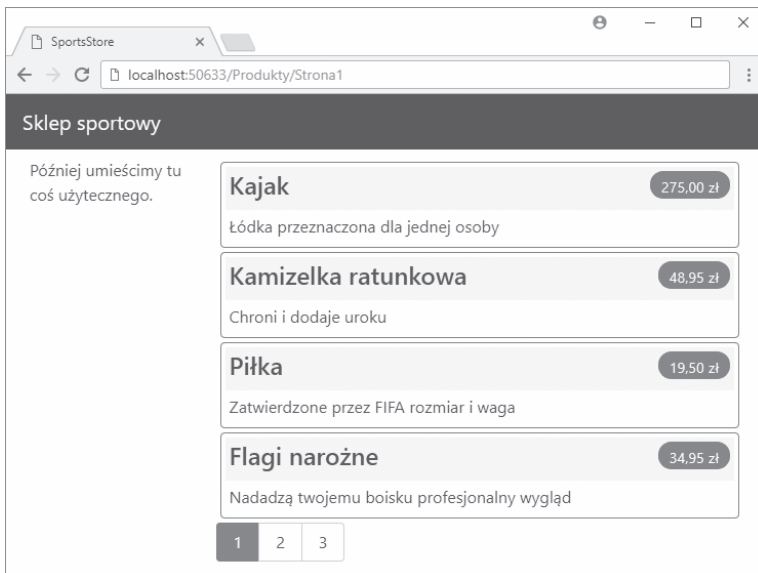
Listing 8.36. Użycie widoku częściowego w *List.cshtml*

```
@model ProductsListViewModel

@foreach (var p in Model.Products)
{
    @Html.Partial("ProductSummary", p);
}

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

Kod, który wcześniej znajdował się w pętli `foreach`, w widoku *List.cshtml*, został przeniesiony do nowego widoku częściowego. Ten widok częściowy wywołujemy przy użyciu metody pomocniczej `Html.Partial()`. Jej argumentami są nazwa widoku oraz obiekt 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). Jak możesz zobaczyć na rysunku 8.13, widok częściowy nie zmienia wyglądu aplikacji. Zmieniła się jedynie lokalizacja, w której silnik Razor odnajduje treść używaną do wygenerowania odpowiedzi przekazywanej do przeglądarki WWW.



Rysunek 8.13. Użycie widoku częściowego

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, `ProductController`, 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.

Skorowidz

.NET Core
instalacja, 352
sprawdzenie instalacji, 353

A

adaptacja wzorca RESTful, 628
adres URL, 235
dopasowanie, 439
generowanie, 471, 482
logowania, 900
najlepsze praktyki, 502
przychodzący, 486
segmenty, 434
w łączach, 479
wychodzący, 474, 493
względny, 788

akcja

Create(), 317
Delete(), 320
Edit(), 305
Error(), 407
GoogleLogin(), 953
Header(), 822
Logout(), 917

akcje

ograniczenie, 975, 978
utworzenie ograniczenia, 976

aktualizowanie repozytorium
produktów, 308

aktywowanie kontroli

poprawności, 316

analiza komponentów, 551

API, application programming
interface, 28

API kontrolera MVC, 511

ApiControllerers

kontrolery typu API, 625
kontrolery typu REST, 624
utworzenie kontrolera
i widoków, 620
utworzenie modelu
i repozytorium, 618

aplikacja wprowadzania danych,
45

formularz, 49
kontrola poprawności danych,
57

łączenie metod, 48

metody, 47

model danych, 46

nadanie stylu zawartości, 63

scena, 45

widok, 47

ListResponses, 67

MyView, 63

RsvpForm, 65

Thanks, 66

wyróżnianie pól, 61

wyświetlenie odpowiedzi, 55

aplikacje

bezpieczeństwo, 323–348

efektywność działania, 624

kompilacja, 359

konfigurowanie, 337, 375, 410,
685

otwartość, 625

szybkość działania, 624

użycie obszarów, 500

wdrażanie, 323–348, 344

architektura

model-widok, 73

model-widok-model widoku,
75

model-widok-prezenter, 74
trójwarstwowa, 74

argumenty UIHint, 749

arkusz stylów CSS, 62, 162, 777

ASP.NET Core, 28

ASP.NET Core Identity, 323,
861–956

funkcje zaawansowane,
925–956

używanie systemu, 870

ASP.NET Core MVC, 25, 28

ASP.NET Web Forms, 26

atak typu CSRF, 745

atrybut

action, 790

ActionContext, 576

ActionName, 964

ActionNamePrefix, 966

AdditionalActions, 971

archive, 790

Area, 499

asp-action

asp-action, 472, 475, 744, 780

asp-antiforgery, 744, 747

asp-append-version, 768, 777,
781

asp-area, 744, 780

asp-controller, 744, 780

asp-fallback-href, 777

asp-fallback-href-exclude, 777

atrybut
 asp-fallback-href-include, 777
 asp-fallback-href-test-class, 777
 asp-fallback-href-test-property, 777
 asp-fallback-href-test-value, 777
 asp-fallback-src, 768
 asp-fallback-src-exclude, 768, 776
 asp-fallback-src-include, 768, 776
 asp-fallback-test, 768
 asp-for, 747, 753, 755, 761
 asp-format, 747, 750
 asp-fragment, 780
 asp-host, 480, 780
 asp-href-exclude, 777
 asp-href-include, 777
 asp-items, 755, 760
 asp-protocol, 480, 780
 asp-route, 476, 481, 744, 780
 asp-route-*, 501, 744, 780
 asp-src-exclude, 768, 772
 asp-src-include, 768, 769
 asp-validation-summary, 840
 Authorize, 589, 898, 906, 917
 Bind, 811, 812
 cite, 790
 Compare, 849
 Consumes, 645
 ControllerContext, 517, 576
 data-val, 855
 DisplayFormat, 751
 enabled, 782
 expires-after, 782
 expires-on, 782
 expires-sliding, 782, 786
 formation, 790
 FormatFilter, 643
 FromBody, 820, 827
 FromForm, 820
 FromHeader, 820
 FromQuery, 820
 FromRoute, 820
 FromServices, 575
 href, 472, 790
 HtmlTargetElement, 722, 724
 HttpGet, 629
 HttpPost, 536

manifest, 790
 name, 806
 names, 767
 NonAction, 964
 NonController, 510, 964
 priority, 782
 Produces, 641, 645
 Profile, 592
 Range, 849
 Remote, 858
 RegularExpression, 849
 Required, 849
 RequireHttps, 585
 Route, 464, 476, 524
 ServiceType, 609
 show-for-action, 739
 src, 790
 srcset, 790
 StringLength, 849
 TypeFilter, 607
 ValidateAntiForgeryToken, 746
 vary-by, 782
 vary-by-cookie, 782
 vary-by-header, 782
 vary-by-query, 782
 vary-by-route, 782
 vary-by-user, 782
 ViewComponentContext, 576
 ViewContext, 576, 733
 ViewDataDictionary, 576
 atrybuty
 C#, 606
 dotyczące źródła danych, 820
 klasy FormTagHelper, 744
 klasy ScriptTagHelper, 768
 kontrolni poprawności, 849
 metody HTTP, 629
 pomocnicze znaczników, 472, 541, 662, 709, 715, 747
 formularza, 741
 funkcje zaawansowane, 725
 kolejność wykonywania, 725
 model widoku, 734
 rejestrwanie, 719
 rozszerzenie zasięgu, 724
 weryfikacja, 763
 włączenie, 730
 zarządzanie zasięgiem, 721
 zawężanie zasięgu, 722

routingu, 429, 462
 wstrzykiwania właściwości, 575, 576
 automatyczna kompilacja klas, 149, 151
 automatycznie implementowane właściwości, 89
 tylko do odczytu, 91
 autoryzacja, 328, 330, 917, 939, 949
 dostępu do zasobu, 945, 947
 oświadczenia, 932

B

baza danych
 dane początkowe, 222, 340
 dodawanie pakietów, 364
 konfiguracja, 335
 migracja, 221, 326, 339, 365, 930
 rozbudowa, 287
 SQLite, 364
 system Identity, 323
 tworzenie, 215, 334, 363, 869
 wyzerowanie, 287, 872
 zdefiniowanie ustawień, 337
 bezpieczeństwo aplikacji, 323–348
 bezpieczne konwencje modelu, 968
 biblioteka jQuery, 237, 670, 766, 839
 błąd, 541
 kontrolni poprawności, 846
 Bootstrap, 237
 Bower, 144, 352, 357
 broker typu, 554
 konfiguracja, 556
 testowanie kontrolera, 556
 buforowanie, 774, 787, 782
 Bundler & Minifier, 165

C

C#, 81
 CDN, content delivery network, 775
 ciąg tekstowy
 połączenia, 218, 324, 335, 867
 zapytania, 218, 642
 Ciężar ViewState, 26

- Cities
 - konfigurowanie aplikacji, 714
 - utworzenie kontrolera, 711
 - utworzenie modelu
 - i repozytorium, 710
 - utworzenie układu i widoków, 711
 - wyzerowanie widoków
 - i układu, 742
 - ConfiguringApps
 - dodawanie pakietów, 379
 - klasa Program, 381
 - klasa Startup, 385
 - konfiguracja usług MVC, 421
 - konfigurowanie aplikacji, 387, 410
 - konfigurowanie projektu, 378
 - pakiety narzędziowe, 381
 - utworzenie projektu, 377
 - ControllersAndActions
 - generowanie odpowiedzi, 519
 - kontrolery, 509
 - pobieranie danych kontekstu, 513
 - utworzenie widoków, 507
 - ConventionsAndConstraints
 - utworzenie kontrolera, 958
 - utworzenie modelu widoku, 958
 - CRUD, 302
 - CSRF, cross-site request forgery, 745
 - CSS, 161, 237, 777
 - cykl życiowy
 - strony, 26
 - filtru, 607
 - usługi, 566, 569, 573
 - zasięgu, 572
 - czas wygaśnięcia buforowanej treści, 785
- ## D
- dane
 - buforowane, 782
 - konfiguracyjne
 - pobieranie, 414
 - ponowne wczytywanie, 412
 - systemu rejestrowania
 - danych, 417
 - stronicowania, 232
 - z obiektów kontekstu, 513
 - debuger, 153
 - punkt przzerwania, 153
 - definiowanie
 - ciągu tekstowego połączenia, 218
 - funkcji, 104
 - intencji, 520
 - modelu, 121
 - opcjonalnych segmentów
 - URL, 448
 - osadzonych wartości
 - domyślnych, 438
 - reguł poprawności, 848
 - tras o zmiennej długości, 450
 - wartości domyślnych, 436
 - własnych ograniczeń, 460
 - własnych prefiksów, 807
 - własnych zmiennych
 - segmentów, 444
 - deklarowanie zależności, 562
 - DependencyInjection
 - analiza komponentów, 551
 - komponenty, 550
 - kontroler, 548
 - model, 547
 - repozytorium, 547
 - test jednostkowy, 550
 - widok, 548
 - dołączanie
 - akcji, 298
 - kontrolek nawigacji, 243
 - nowych produktów, 317
 - pakietu Bootstrap, 959
 - stylów CSS, 237, 315
 - treści, 43
 - dynamicznych, 662
 - statycznych, 162
 - dołączanie
 - do kolekcji, 815
 - do tablic, 813
 - kodu HTML, 805
 - kolekcji typów
 - niestandardowych, 816
 - modelu, 52, 54, 793, 799
 - określanie źródła, 819
 - użycie nagłówków, 822
 - użycie treści żądania, 825
 - selektywne właściwości, 811
 - typów prostych, 802
 - typów złożonych, 803, 823
 - wartości domyślnej, 800
 - domyślna polityka treści, 637
 - domyślne
 - konwencje aplikacji, 964
 - metody konfiguracyjne, 384
 - dopasowanie
 - adresów URL, 435, 439
 - ograniczenia, 453
 - zmienna przechwytyjąca
 - segmentu, 451
 - zmienne opcjonalne, 449
 - wzorca, 96, 97
 - dostarczanie
 - treści statycznej, 161
 - zawartości plików, 540
 - danych kontekstu, 700
 - dostęp
 - do kontrolera, 924
 - do kontrolera API, 628
 - do własnej zmiennej
 - segmentu, 445, 447
 - do zasobu, 945
 - do żądań HTTPS, 584
 - dostosowanie
 - modelu aplikacji, 963, 964
 - systemu routingu, 484
 - dynamiczne dodawanie treści, 43
 - dyrektywa @model, 660
 - dziedziczenie, 688
 - dziennik zdarzeń, 419
- ## E
- edycja
 - konta użytkownika, 891, 895
 - kontrolera, 38
 - produktów, 305, 308
 - użytkowników, 916
 - edytor
 - kodu źródłowego, 154
 - Visual Studio Code, 361
 - efektywność działania aplikacji, 624
 - ekspandery widoku, 673, 677
 - Entity Framework Core, 216
- ## F
- fabryka, 570
 - Filters
 - utworzenie kontrolera, 581
 - utworzenie widoku, 581

filtrowanie listy produktów, 243
filtrujące metody rozszerzających,
101

filtry, 579, 586

- akcji, 587, 591–593
- asynchroniczne akcji, 593
- asynchroniczne wyniku, 596
- autoryzacji, 587
- globalne, 610
- hybrydowe, 598
- kolejność wykonywania, 612
- wyjątku, 587, 599, 601
- wyniku, 587, 594
- z zależnościami, 604
- zarządzanie cyklem
życiowym, 607
- zastosowanie, 606, 609

Font Awesome, 280

format

- JSON, 410, 538, 631
- XML, 640

formatowanie

- danych, 132, 642, 645, 750
- treści, 637
- za pomocą klasy modelu, 751

formularz, 744

- obsługa, 51
- tworzenie, 49

framework

- Bootstrap, 237
- imitacji, 195
- Moq, 196
- MVC, 27
- xUnit.net, 176

funkcje, 104

- administracyjne, 323, 889
- fabryki, 570
- połączonych przeglądarek,
157, 408
- uwierzytelniania użytkownika,
903

G

generowanie

- adresów URL, 479–482
- wychodzących, 471, 483, 493
- listy kategorii, 252, 253
- łączy, 500
- odpowiedzi, 519, 539
- JSON, 538
- HTML, 522

stron WWW, 41

treści, 391, 393

widoku, 41

znacznika <option>, 756–759

Git, 351

instalacja, 351

sprawdzenie instalacji, 351

globalne konwencje modelu, 971,
972

Google, 951

włączenie uwierzytelniania,
951

GridView, 234

GUID, globally unique identifier,
568

H

hasła, 876

I

Identity

- baza danych, 323
- dane początkowe, 326
- kontroler AccountController,
330
- polityka autoryzacji, 328
- wstawianie danych
początkowych, 340

identyfikacja kontrolerów, 510

identyfikator GUID, 568

imitacje, 194, 195

repozytorium, 209

tworzenie obiektu, 197

implementacja

- imitacji, 194
- IView, 651
- IViewEngine, 652
- klasy kontroli poprawności,
886
- klasy routingu, 485
- kontrolera koszyka, 268
- mechanizmu przetwarzania
zamówień, 286
- MVC, 71
- uwierzytelniania, 900
- widoku listy, 304

importowanie widoku, 125, 867

inferencja typów, 109

informacje

dotyczące konfiguracji
aplikacji, 375

o ASP.NET Core Identity, 861

o atrybutach pomocniczych
znaczników formularza, 709,
741

o dołączaniu modelu, 793

o filtrach, 579

o funkcjach systemu routingu,
469

o komponentach widoku, 679

o kontrolerach, 505

o kontrolerach API, 617

o kontroli poprawności
danych modelu, 829

o modelu aplikacji, 961

o ograniczeniach akcji, 973

o silniku widoku Razor, 119

o systemie routingu, 429, 430

o użytkownika, 884, 886, 888,
894

o widokach, 647

inicjalizatory

- indeksu, 95
- obiektów i kolekcji, 94

instalacja

- .NET Core, 352
- SDK, 34
- w systemie Linux, 352
- w systemie macOS, 352
- w systemie Windows, 352
- Entity Framework Core, 216
- Git, 351
- w systemie Linux, 351
- w systemie macOS, 351
- w systemie Windows, 351
- narzędzia bower, 352
- Node.js, 349
- w systemie Linux, 350
- w systemie macOS, 350
- w systemie Windows, 350
- pakietu Bootstrap, 237
- rozszerzenia Visual Studio
Code C#, 165, 355
- Visual Studio, 33
- Visual Studio Code, 353
- w systemie Linux, 354
- w systemie macOS, 354
- w systemie Windows, 353

IntelliSense, 527

interfejs

- IActionConstraint, 976
- IActionConstraintFactory, 981
- IActionResult, 520
- IConfiguration, 415
- IConfigurationBuilder, 412
- IEnumerable<T>, 208
- IEqualityComparer<T>, 182
- IFilterDiagnostics, 603
- IHostingEnvironment, 403
- ILoggingBuilder, 417
- IModelBindingMessage
 - ↳ Provider, 842
- IModelValidator, 851, 852
- IMvcBuilder, 422
- IPasswordValidator<T>, 879
- IPrincipal, 899
- IQueryable<T>, 208
- IRepository, 577
- IRoute, 486
- IRouteConstraint, 460
- IServiceProvider, 571
- IUserValidator<T>, 886
- IViewComponentResult, 690
- programowania aplikacji, 28

interfejsy

- konwencji modelu aplikacji, 965
- metody rozszerzające, 100

interpolacja ciągu tekstowego, 93

InvitesProjects

- baza danych, 363
- kompilacja projektu, 359
- konfigurowanie aplikacji, 359
- kontroler, 365
- model, 360
- odtworzenie aplikacji, 360
- repozytorium, 360
- testy jednostkowe, 369
- tworzenie pliku, 358
- uruchomienie projektu, 359
- widoki, 365

iteracyjny model programowania,

148

izolowanie komponentów, 181,

183, 194

J

JavaScript, 161, 768

wdrożenie, 161

jawna kontrola poprawności

modelu, 835

język C#, 81

jQuery, 237, 670, 766, 839

JSON, javascript object notation,
410, 412, 538, 631

K

katalogi projektu, 78

klasa, *Patrz także* plik

- ActionConstraintContext, 976
- ActionDescriptor, 979
- ActionExecutingContext, 591
- ActionModel, 963
- ActionNamePrefixAttribute, 966
- ActionSelectorCandidate, 979
- AddActionAttribute, 969
- AnchorTagHelper, 780
- AppIdentityDbContext, 922
- ApplicationModel, 961
- Assert, 179
- Attribute, 966
- AuthenticationProperties, 955
- AuthorizationFilterContext, 588
- AuthorizationHandlerContext, 943
- AuthorizationOptions, 940
- AuthorizationPolicyBuilder, 941
- BlockUserRequirement, 943
- Claim, 933
- ClaimsIdentity, 933
- ClaimsPrincipal, 933
- ContentViewComponent
 - ↳ Result, 690
- Controller, 512
- ControllerContext, 516
- ControllerModel, 962
- DictionaryStorage, 562
- EnvironmentTagHelper, 767
- FilterContext, 587
- FormTagHelper, 744
- HomeController, 960
- HtmlContentView
 - ↳ ComponentResult, 690
- HtmlTargetElement, 723
- HttpContext, 517
- HttpRequest, 514
- HttpResponse, 519

IdentityResult, 874

IdentityRole, 907

IdentityUser, 866

LinkTagHelper, 777, 779

ModelStateDictionary, 836

ModelValidationContext, 852

ParameterModel, 963

PasswordOptions, 878

PasswordValidator, 881

PocoController, 510

Program, 381

PropertyModel, 962

RazorPage, 661

RazorPage<T>, 660

ReservationController, 628

ResultExecutingContext, 595

ResultFilterAttribute, 595

RoleManager<T>, 906, 909

RouteContext, 487

routing, 488

ScriptTagHelper, 768, 770, 775

SelectTagHelper, 760

ShoppingCart, 98

SimpleRepository, 154

Startup, 151, 385

TagHelper, 716

TagHelperContent, 727

TagHelperContext, 716

TagHelperOutput, 718, 728

UrlResolutionTagHelper, 790,
791

UserManager<T>, 915

UserValidator<T>, 888

ValidationMessage, 763

ValidationSummary, 763

ValidationSummaryTag

↳ Helper, 840

ViewComponent, 256, 688,
690, 691

ViewComponentContext, 695

ViewContext, 650

ViewDataDictionary, 651

ViewEngineResult, 650

ViewLocationExpander
↳ Context, 675

ViewResult, 522

ViewViewComponentResult,
690

VirtualPathContext, 495

WebHostBuilderContext, 411

- klasy
 - atrybutu pomocniczego znacznika, 715
 - bazy danych, 217
 - błędów, 838
 - hybrydowe, 705
 - konfiguracyjne, 426
 - kontekstu bazy danych, 867
 - kontroli poprawności hasła, 882
 - kontrolera, 79
 - weryfikacji atrybutów pomocniczych znaczników, 763
 - wyniku akcji, 543
- koalescencja, 88
- kod
 - 404, 543
 - wbudowany, 662
- kody HTTP, 541
- kolejność
 - tras, 442
 - wykonywania konwencji modelu, 970
- kompilacja aplikacji, 359
- komponenty
 - asynchroniczne widoku, 701
 - luźno powiązane, 551
 - rozdzielenie, 552
 - widoku, 281, 662, 679, 686
 - widoku typu POCO, 687
- komunikat
 - błędu, 800, 844, 877, 884
 - systemu dołączania modelu, 844
 - potwierdzający, 312
 - weryfikacji danych, 842
 - własny, 419
- konfiguracja
 - aplikacji, 325, 337, 359, 410, 685, 714
 - ASP.NET Core Identity, 865
 - bazy danych, 335
 - brokera typu, 556
 - domyślnych komunikatów, 842
 - dostawcy usługi, 559, 565
 - funkcji połączonych przeglądarek, 157
 - importu widoków, 867
 - mapowania typu, 563
 - mechanizmu wstrzykiwania zależności, 420
 - projektu, 378
 - produkcyjnej bazy danych, 337
 - repozytorium, 363
 - serializera JSON, 639
 - silnika Razor, 672, 674
 - systemu rejestrowania danych, 416
 - systemu routingu, 484
 - trasy domyślnej, 214
 - usług MVC, 421
 - usługi aplikacji, 346
 - znaczników <input>, 748
- konstrukcje warunkowe, 134
- konto użytkownika, 872, 875
 - edycja, 891
 - usuwanie, 890
- kontrola poprawności, 293, 316
 - danych modelu, 57, 313, 829, 834
 - definiowanie reguł poprawności, 848
 - domeny adresu e-mail, 952
 - hasła, 876–882
 - implementacja własnej klasy, 886
 - informacji o użytkowniku, 884, 888
 - jawna modelu, 835
 - konflikty, 855
 - po stronie klienta, 316, 853, 856
 - w JQuery, 856
 - własne atrybuty, 851
 - wyświetlanie
 - błędów, 838, 840, 843, 845
 - komunikatów, 840
 - podsumowania, 841
 - zdalna, 856
- kontroler, 38, 70, 509
 - AccountController, 330, 902
 - AdminController, 921, 924
 - API, 626
 - metody akcji, 628
 - testowanie, 630
 - trasy, 628
 - użycie w przeglądarce WWW, 635
 - wyniki akcji, 629
 - zależności, 628
- ClaimsController, 935
- CRUD, 302
- CustomerController, 464–466
- ErrorController, 336
- HomeController, 683, 745, 918, 973
- PocoController, 510
- RoleAdminController, 921
- kontrolery
 - identyfikacja, 510
 - koszyka, 268
 - metody View(), 523
 - nawigacji, 251
 - pobieranie danych kontekstu, 515
 - tworzenie, 509
 - typu
 - API, 917, 625
 - POCO, 510, 515
 - REST, 625
 - RESTful, 624
 - użyteczne właściwości, 513
- kontrolki nawigacji, 243
- konwencje
 - aplikacji, 964
 - dla klas kontrolerów, 79
 - dla układów, 80
 - dla widoków, 79
 - modelu, 957, 965
 - globalne, 971
 - kolejność wykonywania, 970
 - usunięcie, 972, 974
 - MVC, 79
- koordynacja atrybutów pomocniczych znaczników, 738
- koszyk na zakupy, 261–273
 - definiowanie modelu, 262
 - dopracowanie modelu, 275
 - kontroler, 290
 - implementowanie, 268
 - uproszczenie, 277
 - podsumowanie, 280
 - przechowywanie szczegółów, 267
 - testowanie, 263
 - tworzenie, 261
 - tworzenie przycisków, 265
 - usuwanie produktów, 278
 - wyświetlanie zawartości, 270

L

LanguageFeatures
 obsługa ASP.NET Core MVC,
 82
 utworzenie kontrolera
 i widoku, 85
 utworzenie modelu, 84
 licznik stron, 258
 LINQ, 116
 lista
 kategorii, 252, 253
 produktów, 210, 243
 LTS, long term support, 350

Ł

łącza stron, 500
 testowanie, 229
 wyświetlanie, 227, 233
 łączenie
 metod akcji, 48
 metod HTTP i adresu URL,
 626
 operatorów, 88
 operatorów warunkowych
 null, 87
 plików, 165, 166
 statycznych segmentów URL,
 442

M

magazyn danych, 275
 mapowanie typu, 563
 mechanizm przetwarzania
 zamówień, 286
 menedżer pakietów NuGet, 380
 menu
 Debug, 150
 Debugowanie, 154
 nawigacji, 251
 metadane, 848
 metoda
 @RenderBody(), 128
 AddAuthentication.
 ➔ AddGoogle(), 952
 AddIdentity(), 907
 AddScoped, 567
 addSingleton, 567
 AddTransient, 567

Configure(), 387, 401, 473
 ConfigureServices(), 386, 388,
 560
 Content(), 693
 Create(), 318, 803
 ExecuteResultAsync(), 523
 Filter(), 105
 FilterByPrice(), 103
 GetProducts(), 94
 GetView(), 653
 GetVirtualPath(), 493
 GoogleLogin(), 955
 HTTP
 DELETE, 626
 GET, 626
 PATCH, 626
 POST, 626
 PUT, 626
 HttpClient.GetAsync(), 113
 InvokeAsync(), 701
 IUrlHelper(), 249
 Process(), 718
 SignOutAsync(), 904
 StatusCode(), 542
 string.Format(), 93
 TransformAsync(), 938
 Url.Action(), 482, 483, 495,
 541
 VerifyGet(), 199
 Where(), 186
 metody
 asercji, 179
 asynchroniczne, 111
 filtru, 104
 interfejsu
 IConfigurationBuilder, 412
 ILoggingBuilder, 417
 IMvcBuilder, 422
 IServiceProvider, 571
 klasy
 AuthorizationHandler
 ➔ Context, 943
 AuthorizationPolicy
 ➔ Builder, 941
 ClaimsIdentity, 933
 ClaimsPrincipal, 933
 ModelStateDictionary, 836
 RoleManager<T>, 909
 TagHelperContent, 727, 728
 TagHelperOutput, 718

userManager<T>, 915
 ViewComponent, 691
 ViewEngineResult, 650
 konfiguracyjne, 424
 ASP.NET Core, 384
 LINQ, 116
 pomocnicze HTML, 717
 rozszerzające, 98
 filtrujące, 101
 rozszerzenia, 567, 571
 View kontrolera, 523
 migracja bazy danych, 221, 326,
 365, 930
 minimalizacja plików, 165, 166
 model, 70
 aplikacji, 961
 dostosowanie, 963
 domeny, 208
 koszyka, 262, 275
 widoku, 227
 dodawanie danych, 231
 ModelValidation
 utworzenie kontrolera, 832
 utworzenie modelu, 831
 utworzenie układu i widoków,
 832
 modyfikacje
 klas C#, 149
 odpowiedzi, 399
 widoków Razor, 148
 żądań, 396
 MVC
 implementacja w ASP.NET, 71
 konfiguracja usług, 421
 kontroler, 70
 konwencje, 79
 model, 70
 odmiany wzorca, 74
 pliki projektu, 77–79
 projekt, 75
 testy jednostkowe, 175
 tworzenie projektu, 202
 widok, 71
 MvcModels
 konfigurowanie aplikacji, 797
 utworzenie kontrolera
 i widoku, 796
 utworzenie modelu
 i repozytorium, 794

N

nadanie stylu widokowi, 63
 ListResponses, 67
 MyView, 63
 RsvpForm, 65
 Thanks, 66
 nagłówek żądania, 823
 narzędzie
 Bower, 352, 357
 Git, 351
 NuGet, 143
 nawigacja, 243–273
 menu, 251
 tworzenie kontrolera, 251
 negocjacja treści, 639, 644
 niewłaściwe rozdzielanie zadań, 26
 Node.js
 instalacja, 349
 sprawdzenie instalacji, 351
 nowy projekt, 35
 początkowa konfiguracja, 36
 początkowa struktura plików,
 37
 szablon, 35
 NuGet, 143, 196
 dodawanie pakietu, 380

O

obiekt
 HttpContext, 577
 HttpResponse, 519
 ModelState, 836
 MvcOptions, 422
 RouteOptions, 484
 Task, 688
 ViewBag, 529
 ViewResult, 523
 obiekty do generowania
 odpowiedzi, 539
 obsługa
 attributów pomocniczych
 znaczników, 172
 dostarczania treści statycznej,
 161
 formularzy, 51
 funkcji połączonych
 przeglądarek, 408
 mechanizmu wstrzykiwania
 zależności, 567
 plików statycznych, 162

sesji, 267
 stron wyjątków
 programisty, 152
 treści statycznej, 409
 uwierzytelniania użytkownika,
 903
 wyjątków, 406
 żądań POST, 310
 obszary, 496, 497
 tworzenie, 496
 wypełnianie, 498
 odpowiedź, 55
 HTML, 522
 ograniczenia
 akcji, 957, 973–978
 dostępu do kontrolera, 924
 dostępu do żądań, 584
 trasy, 452
 do zbioru wartości, 458
 wyrażenia regularne, 456
 ograniczona kontrola nad HTML,
 26
 okno
 Eksplorator rozwiązań, 207
 Eksplorator testów, 180
 Lokalne, 156
 określanie formatu danych
 akcji, 641
 opcja
 RespectBrowserAcceptHeader,
 645
 operacje CRUD, 302
 operator
 koalescencji, 88
 warunkowy null, 86
 oprogramowanie
 pośredniczące, 391
 dołączanie komponentów, 405
 generowanie treści, 391
 modyfikujące odpowiedź, 399
 modyfikujące żądanie, 396
 skrcające potok żądań, 394
 użycie usług, 393
 oświadczenia, 931, 936
 otwartość aplikacji, 625

P

pakiet, 142
 Bootstrap, 148, 237, 582
 Bower, 145, 352

Font Awesome, 280
 jQuery, 670
 Microsoft.AspNetCore.All, 143
 NuGet, 143, 196
 pakiety
 działające po stronie
 klienta, 357
 .NET, 143
 narzędziowe, 381
 parametry metod akcji, 447, 517
 parametryzowanie testu
 jednostkowego, 190
 pętla
 foreach, 969
 Razor @foreach, 669
 pierwsza aplikacja, 33
 pliki
 JavaScript, 768
 konfiguracyjne
 łączenie i minimalizacja, 167
 zewnętrzne, 423
 plik
 .bowerrc, 358
 .csproj, 379
 _AdminLayout.cshtml, 301,
 312, 315
 _Layout.cshtml, 237, 251, 666,
 683
 _ViewImports.cshtml, 229,
 690, 719, 911
 AccessDenied.cshtml, 920
 AccountController.cs, 331,
 901, 919, 953
 ActionNamePrefixAttribute.cs,
 965
 AddActionAttribute.cs, 967,
 968
 AddressSummary.cs, 808, 812
 AdminController.cs, 329, 873,
 890
 AdminControllers.cs, 310, 870
 AlternateRepository.cs, 555
 AppIdentityDbContext.cs, 324
 ApplicationDbContext.cs, 287
 appsettings.development.json,
 424
 appsettings.json, 324, 412, 868
 appsettings.production.json,
 337
 AppUser.cs, 866, 927

- BlockUsersRequirement.cs, 942
- bower.json, 145, 280, 582, 766, 864, 959
- BrowserTypeMiddleware.cs, 396
- Cart.cs, 262
- CartController.cs, 271, 277
- CartItemViewModel.cs, 271
- CartSummaryView
 - Component.cs, 281
- Checkout.cshtml, 285, 293
- City.cs, 752, 754
- CityController.cs, 703
- CitySummary.cs, 689, 691, 698
- ClaimsController.cs, 932
- ClaimTypeTagHelper.cs, 934
- ColorExpanders.cs, 676
- ConfiguringApps.csproj, 378, 381
- ContentController.cs, 641, 645
- ContentMiddleware.cs, 391, 393
- CountryNames.cs, 756
- Create.cshtml, 684, 757, 809
- CustomerController.cs, 451, 465
- CustomHtmlResult.cs, 521
- CustomUserValidator.cs, 886, 952
- DebugDataView.cs, 651
- DebugDataViewEngine.cs, 652
- Default.cshtml, 254, 702
- DerivedController.cs, 512
- DictionaryResult.cshtml, 507
- DictionaryStorage.cs, 562
- DisplaySummary.cshtml, 809
- DI.Tests.cs, 553
- DocumentAuthorization.cs, 948
- DocumentController.cs, 946
- DocumentControllers.cs, 949
- Edit.cshtml, 307, 314, 894
- EFProductRepository.cs, 309, 319
- EFRepository.cs, 362
- ErrorController.cs, 336
- ErrorMiddleware.cs, 399
- ExampleController.cs, 528, 533, 542
- FakeProductRepository.cs, 209
- FilterDiagnostics.cs, 603
- first.css, 163
- FormButtonTagHelper.cs, 726
- FormTagHelper.cs, 732
- GetLegacyUrl.cshtml, 490
- GuestResponse.cs, 46, 361
- HeaderModel.cs, 823
- HomeController.cs, 38, 47, 85, 365, 419
- HomeController.cshtml, 53
- HomeControllerTests.cs, 183, 194, 370
- IdentitySeedData.cs, 326, 340
- IModelStorage.cs, 561
- Index.cshtml, 85, 123, 174
- IOrderRepository.cs, 288
- IProductRepository.cs, 309
- IRepository.cs, 361, 547
- LanguageFeatures.csproj, 112
- LegacyController.cs, 490
- LegacyRoute.cs, 486
- List.cshtml, 233, 250
- ListResponses.cshtml, 368
- LocationClaimsProvider.
 - cshtml, 936
- Login.cshtml, 332, 902, 953
- LoginModel.cs, 330
- MakeBooking.cshtml, 838, 840, 844, 847
- MemoryRepository.cs, 562
- MyAsyncMethods.cs, 112
- MyExtensionMethods.cs, 99–104
- MyView.cshtml, 44, 45, 366
- NavigationMenuView
 - Component.cs, 251, 253, 256
- Order.cs, 283, 297
- OrderController.cs, 284, 290, 298
- PageLinkTagHelper.cs, 228, 239, 249
- PageSize.cs, 701
- PocoController.cs, 510, 511
- PocoController.cs, 515
- PocoViewComponent.cs, 687
- Product.cs, 84, 87, 90, 91
- ProductController.cs, 211, 231, 259
- ProductsListViewModel.cs, 231
- ProductTestData.cs, 192
- ProductTests.cs, 178
- ProductTotalizer.cs, 564
- Program.cs, 381, 382
- ProtectedDocument.cs, 945
- Repository.cs, 795
- ReservationController.cs, 627
- Result.cs, 958
- Result.cshtml, 433, 481, 959
- RoleAdminController.cs, 907, 913
- RoleUsersTagHelper.cs, 910
- RsvpForm.cshtml, 47, 59, 65, 367
- SeedData.cs, 222, 342
- SelectiveTagHelper.cs, 739
- SelectOptionTagHelper.cs, 759
- SessionCart.cs, 275
- ShoppingCart.cs, 98, 100
- ShortCircuitMiddleware.cs, 394, 397
- SimpleForm.cshtml, 508, 541
- SimpleRepository.cs, 184
- Startup.cs, 83, 120, 214, 337, 565, 657
- StartupDevelopment.cs, 426
- TableCellTagHelper.cs, 730
- TimeViewComponent.cs, 783
- TypeBroker.cs, 554
- UptimeService.cs, 388
- UrlExtensions.cs, 265
- UserAgentAttribute.cs, 976, 980, 982
- UserAgentComparer.cs, 981
- UserViewModels.cs, 900, 912
- ViewResultDiagnostics.cs, 610
- Views/Home/ListResponses.
 - cshtml, 56
- Views/Home/Thanks.cshtml, 54
- ViewStart, 129
- ViewStart.cshtml, 129
- WeekDayConstraint.cs, 460
- pobieranie
 - danych kontekstu, 513, 587, 695, 732
 - formatu danych, 642
 - nazw, 115
- pola z błędami, 61
- polecenia importujące widoki, 125

polecenie
 @foreach, 137
 @model, 123
 case, 98
 dotnet add package, 380
 switch, 97
 polityka
 autoryzacji, 939, 942, 947, 949
 bezpieczeństwa, 334
 treści, 637
 połączenie z bazą danych, 867
 porty HTTP, 623
 poziomy rejestrowanie danych, 418
 prefiks, 807
 asp-route-, 476
 programowanie sterowane
 testami, 189
 projekt, 75, 202, 356
 ApiController, 618
 Cities, 710, 742, 766
 ConfiguringApps, 377
 ControllersAndActions, 506
 ConventionsAndConstraints,
 957
 DependencyInjection, 546
 Filters, 580
 InvitesProjects, 356
 LanguageFeatures, 82
 ModelValidation, 830
 MvcModels, 794
 Razor, 120
 SportsStore, 201
 UrlsAndRoutes, 430, 470
 Users, 862, 897, 926
 UsingViewComponents, 680
 Views, 648
 WorkingWithVisualStudio,
 139, 172
 projektowanie
 modelu danych, 46
 testów jednostkowych, 176, 550
 przechowywanie odpowiedzi, 53
 przekazywanie
 danych, 529
 danych do widoku, 526
 kontekstu z widoku
 nadrzędnego, 698
 przekierowania, 530
 do adresu URL, 533
 do innej metody akcji, 534
 do literału adresu URL, 531

metod akcji, 535
 trwałe, 531, 532
 tymczasowe, 530
 wyników akcji, 531
 z użyciem tras, 533
 przestrzeń nazw
 Microsoft.AspNetCore.Mvc, 689
 przesyłanie danych edycji, 311
 przetwarzanie
 danych, 132
 właściwości modelu, 800
 przychodzące adresy URL, 486
 przycisk zamówienia, 284
 przyciski koszyka na zakupy, 265
 przypisanie wartości atrybutu, 133
 punkt przerywania, 153

R

Razor, 119, 656, 658
 dodanie treści JSON, 670
 dynamiczne treści, 662
 generowanie sekcji
 opcjonalnych, 666
 konfigurowanie silnika, 672
 modyfikacje widoków, 148
 obiekt modelu, 123
 sekcje układu, 663
 sprawdzanie istnienia sekcji, 666
 tworzenie widoku
 częściowego, 668
 układy, 126
 właściwości, 661
 wyrażenia, 131
 zastosowanie widoku
 częściowego, 668
 refaktoring aplikacji, 240
 reguły poprawności, 848
 rejestrowanie
 aplikacji w Google, 951
 atrybutu pomocniczego
 znacznika, 719
 danych, 412, 416–418
 silnika widoku, 653
 trasy, 435
 usługi, 276, 388
 usługi repozytorium, 209
 repozytorium, 208, 217
 konfiguracja, 363
 produktów, 308
 zamówień, 288

REST, representational state
 transfer, 625
 role, 907
 konwencji modelu, 965
 testowanie, 912
 testowanie użytkowników, 916
 tworzenie, 907, 912
 usuwanie, 907, 912
 zarządzanie użytkownikami,
 912
 routing, 469–503, *Patrz także* trasy
 do kontrolerów MVC, 490
 dostosowanie systemu, 484
 funkcje zaawansowane,
 469–503
 oparty na konwencji, 429
 URL, 429–467
 zmiana konfiguracji systemu,
 484
 rozdzielanie komponentów, 552
 rozszerzenie Bundler & Minifier,
 165

S

schemat URL, 247, 434, 443, 502
 segmenty
 adresu URL, 434
 mieszane, 441
 statyczne, 440
 catchall, 451, 452
 ograniczenia
 tras, 455
 typu i wartości, 457
 znienie
 opcjonalne, 448
 przechwytyjące, 451
 wielokrotnie wykorzystane,
 478
 własne, 444, 447
 sekcje, 662
 serializer JSON, 639
 serwer Kestrel, 385
 sesja
 włączenie obsługi, 267
 sieć CDN, 775
 silne typowanie, 527
 silnik widoku
 rejestrowanie, 653
 testowanie, 654
 tworzenie, 649

- Razor, 119, 656, 658
 - dodanie treści JSON, 670
 - dynamiczne treści, 662
 - generowanie sekcji
 - opcjonalnych, 666
 - konfigurowanie silnika, 672
 - modyfikacje widoków, 148
 - obiekt modelu, 123
 - sekcje układu, 663
 - sprawdzanie istnienia sekcji, 666
 - tworzenie widoku
 - częściowego, 668
 - układy, 126
 - właściwości, 661
 - wyrażenia, 131
 - zastosowanie widoku
 - częściowego, 668
 - singleton, 573, 982
 - składanie zamówień, 282, 284
 - skręcanie potoku żądań, 394
 - Słaba abstrakcja, 26
 - słowo kluczowe
 - async, 113
 - await, 113
 - new, 551
 - var, 109, 111
 - Smart UI, 72
 - SportsStore, 201–348
 - administracja, 297–322
 - aktualizowanie repozytorium produktów, 308
 - aktywowanie kontroli
 - poprawności, 316
 - autoryzacja, 328
 - baza danych, 215
 - część administracyjna, 347
 - dane modelu widoku, 231
 - dodanie
 - akcji, 298
 - stronicowania, 225
 - kontrolera, 211
 - nowych produktów, 317
 - stylu, 237
 - widoku, 212, 298
 - edycja produktów, 305
 - klasa atrybutu pomocniczego
 - znacznika, 228
 - komunikaty potwierdzające, 312
 - konfigurowanie
 - aplikacji, 203, 219, 325, 337
 - trasy domyślnej, 214
 - usługi, 346
 - kontrola poprawności modelu, 313
 - kontroler
 - AccountController, 330
 - ErrorController, 336
 - kontrolki nawigacji, 243
 - koszyk na zakupy, 261, 275–295
 - lista produktów, 210
 - łącza stron, 227, 233
 - metody, 204
 - model
 - widoku, 227
 - domeny, 208
 - nawigacja, 243–273
 - obsługa żądań POST, 310
 - repozytorium, 208
 - składanie zamówień, 282
 - style Bootstrap, 237
 - system Identity, 323
 - testy jednostkowe, 201, 206
 - układ graficzny, 240
 - uruchomienie aplikacji, 207, 215
 - usuwanie produktów, 319
 - wdrożenie, 334, 344
 - widok częściowy, 240
 - zabezpieczanie funkcji
 - administracyjnych, 323
 - zarządzanie
 - katalogiem, 301
 - zamówieniami, 297
 - sprawdzenie typu, 96
 - SQLite, 364
 - strona
 - podsumowania, 294
 - wyjątków programisty, 151, 152
 - stronicowanie, 225
 - struktura plików, 37
 - style, 237
 - Bootstrap, 237
 - CSS, 315
 - RESTful, 628
 - system
 - Identity, 323, 326
 - kontroli poprawności, 293
 - rejestrowania danych, 412, 416, 417
 - routingu, 429
 - szablon projektu, 35, 76
 - Aplikacja internetowa, 38
 - szkielet MVC, 211
 - szybkość działania aplikacji, 624
 - szyfrowanie SSL, 581
- ## Ś
- środowisko hostingu, 402
- ## T
- tablice
 - wyświetlanie zawartości, 136
 - TDD, test-driven development, 189
 - test jednostkowy, 171, 175, 177
 - akcja Index(), 303
 - akcji, 522, 538
 - aktualizowanie testów, 245
 - atrybutu pomocniczego
 - znacznika, 720
 - dane stronicowania, 232
 - filtrów, 247, 589
 - generowanie
 - listy kategorii, 253
 - widoku, 524
 - kategorie, 256
 - kody HTTP, 543
 - komponentów, 181
 - komponentu widoku, 699
 - kontrolera, 522, 560
 - koszyk na zakupy, 263
 - metoda akcji Edit(), 306
 - tablice
 - obiekt ViewBag, 530
 - obiekty modelu widoku, 527
 - parametryzowany, 190
 - projektowanie, 206
 - przekierowania, 532, 535
 - przesyłanie danych edycji, 311
 - przetwarzanie zamówień, 291
 - stronicowanie, 226
 - tworzenie łączy stron, 229
 - usprawnienia, 190
 - usuwanie produktów, 321
 - w Visual Studio Code, 369
 - zliczanie produktów, 260

- testowanie
 - kontrolera, 553, 556
 - kontrolera API, 630
 - operacji
 - DELETE, 634
 - GET, 631
 - PATCH, 633
 - POST, 632
 - PUT, 633
 - polityki bezpieczeństwa, 334
 - ról, 912
 - silnika widoku, 654
 - uwierzytelniania, 905
 - użytkowników, 916
 - widoku, 43
 - właściwości, 931
- testy automatyczne, 26
- trasy, 40, 248, 434
 - dla obszarów, 497
 - domyślne, 214
 - dostęp do kontrolera API, 628
 - generowanie adresu URL, 480
 - kolejność, 442
 - nazwane, 482
 - o zmiennej długości, 450
 - ograniczenia, 452, 458
 - osadzone wartości
 - domyślne, 438
 - pobieranie formatu
 - danych, 642
 - rejestrowanie, 435
 - skomplikowane, 465
 - tworzenie, 435
 - w kontrolerze
 - CustomerController, 465, 466
 - wartości domyślne, 436
 - własne ograniczenia, 460, 461
 - zastosowanie ograniczeń, 466
- treści
 - dynamiczne, 662
 - statyczne, 162, 409
- tworzenie
 - arkusza stylów, 62
 - asynchronicznego
 - filtru akcji, 593
 - filtru wyniku, 596
 - komponentu widoku, 701, 703
 - atributu
 - kontroli poprawności, 851
 - pomocniczego
 - znacznika, 715
 - automatycznie
 - implementowanych
 - właściwości, 91
 - bazy danych, 334, 363, 869
 - bezpiecznej konwencji
 - modelu, 968
 - danych początkowych, 222
 - ekspandera widoku, 673
 - elementów skrótu, 726
 - filtru
 - akcji, 592
 - hybrydowego, 598
 - wyjątku, 601
 - wyniku, 595
 - filtrujących metod
 - rozszerzających, 101
 - formularza, 49, 173
 - globalnych konwencji modelu, 971
 - imitacji repozytorium, 209
 - implementacji
 - IView, 651
 - IViewEngine, 652
 - klasy
 - atributu pomocniczego
 - znacznika, 228
 - bazy danych, 217
 - komponentu widoku, 281
 - konfiguracyjnej, 426
 - kontekstu, 324
 - kontekstu bazy danych, 867
 - modelu, 431
 - repozytorium, 217
 - konta użytkownika, 881, 872, 875
 - kontrolera, 121, 431, 509, 711
 - AccountController, 330
 - API, 626
 - CRUD, 302
 - ErrorController, 336
 - nawigacji, 251
 - POCO, 510
 - konwencji modelu, 965
 - metod konfiguracyjnych, 424
 - metody akcji Edit, 305
 - migracji bazy danych, 365
 - modelu, 84, 360, 680, 710
 - akcji, 970
 - domeny, 208
 - nowego projektu, 35
 - obiektu imitacji, 197
 - obszaru, 496
 - ograniczenia akcji, 976
 - oprogramowania
 - pośredniczącego, 391, 394
 - oświadczeń tożsamości, 936, 939
 - pliku, 358
 - arkusza stylów, 162
 - JavaScript, 164
 - konfiguracyjnego Bower, 145
 - konfiguracyjnego JSON, 412
 - ViewStart, 130
 - projektu, 76
 - ASP.NET Core, 356
 - MVC, 202
 - przycisków koszyka, 265
 - punktu przerwania, 154
 - repozytorium, 208, 288, 360, 680, 710
 - ról, 907, 912
 - silnika widoku, 649
 - struktury katalogu, 203
 - testów
 - automatycznych, 26
 - jednostkowych, 176, 206, 369, 550
 - trasy, 435
 - obszaru, 497
 - układu, 127
 - usługi aplikacji Azure, 345
 - widoku, 41, 85, 122, 254, 433, 548, 686, 711
 - częściowego, 240
 - edycji, 307
 - hybrydowego, 704
 - typu POCO, 687
 - własnych komunikatów, 419
 - wymagań polityki autoryzacji, 942
 - zewnętrznych plików
 - konfiguracyjnych, 423
 - typ wyliczeniowy
 - ModelState, 837
 - ValidationSummary, 841
 - typy
 - anonimowe, 109
 - filtrów, 587
 - właściwości C#, 749

U

- uaktualnienie widoku, 164
- układy, 126
 - stosowanie, 129
 - tworzenie, 127
- URL
 - dopasowanie adresów, 439
 - generowanie adresów, 471
 - łączenie statycznych segmentów, 442
 - opcjonalne segmenty, 448
 - statyczne segmenty adresu, 440
 - ulepszanie adresów, 235
 - ulepszanie schematu, 247
 - własne zmienne segmentu, 444
 - wzorce, 434
- UrlsAndRoutes
 - utworzenie klasy modelu, 431
 - utworzenie kontrolerów, 431
 - utworzenie widoku, 433
 - wzorce URL, 434
- Users
 - baza danych, 921
 - dane wyjściowe aplikacji, 936
 - dodawanie właściwości, 926
 - implementacja
 - uwierzelniania, 900
 - informacje o użytkowniku, 884
 - klasa kontekstu bazy danych, 867
 - lista kont użytkowników, 870
 - migracja bazy danych, 930
 - testowanie właściwości, 931
 - utworzenie
 - klasy użytkownika, 865
 - konta użytkownika, 872
 - kontrolera i widoku, 863, 909
 - uwierzelnianie użytkownika, 898
- UsingViewComponents
 - konfigurowanie aplikacji, 685
 - utworzenie kontrolera i widoków, 682
 - utworzenie modeli i repozytoriów, 680
- usługa
 - typu RESTful, 626
 - typu singleton, 573
- usługi, 388
 - cykl życiowy, 566
 - rejestrowanie, 276, 388
 - specjalne, 401
 - w oprogramowaniu
 - pośredniczącym, 393
 - wbudowane, 390
 - własne, 388
- usuwanie
 - konta użytkownika, 890
 - konwencji modelu, 972, 974
 - modeli, 967
 - produktów, 319
 - ról, 907, 912
- utrata ważności buforowanych danych, 785
- uwierzelnianie, 36
 - dwupoziomowe, 905
 - poprzez dostawcę zewnętrznego, 950
 - użytkownika, 898, 903
 - z uwzględnieniem roli, 906
- użycie
 - API kontrolera MVC, 511
 - ASP.NET Core Identity, 870, 897
 - atrybutów
 - routingu, 462
 - wstrzykiwania właściwości, 575
 - atrybutu
 - Area, 499
 - Bind, 811, 812
 - HtmlTargetElement, 722, 724
 - pomocniczego znacznika, 472, 719
 - automatycznie
 - implementowanych metod, 90
 - właściwości, 89
 - brokera typu, 554
 - buforowanych danych, 782
 - treści, 786
 - cyklu życiowego
 - usługi, 566, 569, 573
 - zasięgu, 572, 573
 - danych konfiguracyjnych, 414
 - debugera, 153
 - dołączania modelu, 52
- filtrów, 583
 - akcji, 591
 - autoryzacji, 587, 588
 - globalnych, 610
 - wyjątków, 599
 - wyniku, 594, 597
- funkcji
 - CSRF, 745
 - fabryki, 570
 - połączonych przeglądarek, 157, 158
- inferencji typów, 109
- inicjalizatora
 - indeksu, 95
 - kolekcji, 94
 - obiektów, 94
- interpolacji ciągu tekstowego, 93
- iteracyjnego modelu
 - programowania, 148
- klasy
 - bazowej kontrolera, 512
 - Controller, 513
 - hybrydowej, 705
 - Program, 381
 - Startup, 385, 387
- komunikatów błędów, 845
- konstrukcji warunkowych, 134
- kontrolera API, 635
- użycie
 - kontroli poprawności, 853
 - konwencji modelu, 960, 966
 - łańcucha zależności, 561
 - mechanizmu wstrzykiwania zależności, 564, 603
- metod
 - akcji, 541, 804
 - asynchronicznych, 111
 - rozszerzających, 98, 100
- modelu aplikacji, 960
- narzędzia NuGet, 143
- obiektów do generowania odpowiedzi, 539
- obiektu
 - kontekstu
 - HttpResponse, 519
 - ModelState, 836
 - modelu widoku, 526
 - ViewBag, 529, 568
- obszarów, 497, 500
- ograniczeń akcji, 973

użycie

- okna Lokalne, 156
- operatora warunkowego null, 86
- parametrów metod akcji, 517
- pliku ViewStart, 129
- polityki, 945
 - autoryzacji, 939
- prefiksu asp-route-, 476
- segmentu catchall, 452
- serwera Kestrel, 385
- statycznych segmentów adresu URL, 440
- ścieżki dostępu, 525
- środowiska hostingu, 402
- tras nazwanych, 482
- treści żądania, 825
- typów anonimowych, 110
- układu, 129
- usług w oprogramowaniu pośredniczącym, 393
- wariantów buforowania, 787
- widoków częściowych, 669
- wielu przeglądarek WWW, 160
- własnych zmiennych segmentu, 447
- właściwości dla repozytorium, 552
- właściwości TempData, 536
- wstrzyknięcia akcji, 575
- wyniku akcji, 521
- wrażenia
 - @model, 123
 - nameof, 116
- wrażień
 - lambda, 103, 107
 - Razor, 131
 - regularnych, 456
 - względego adresu URL, 788
 - wzorca POST-przekierowanie-GET, 535
- użytkownik przypisany do roli, 916

V

Views

- implementacja IView, 651
- implementacja IViewEngine, 652

- silnik Razor, 658, 656
- tworzenie silnika widoku, 649
- Visual Studio, 33, 139
 - Code, 349
 - rozszerzenie C#, 355
 - testy jednostkowe, 369
 - tworzenie projektu, 356

W

- wartość null, 86
- wczytywanie plików konfiguracyjnych, 423
- wdrożenie
 - aplikacji, 323–348
 - CSS, 161
 - kodu JavaScript, 161
- weryfikacja zakresu, 220
- widoki, 41, 71, 647–678
 - częściowe, 240, 662, 669, 691
 - edycji, 307
 - obsługa żądań POST, 310
 - hybrydowe, 704
 - komponenty, 679, 686
 - konfiguracja importu, 867
 - listy, 304
 - nadrzędne, 698
 - Razor, *Patrz* silnik widoku Razor
 - ścieżka dostępu, 525
- właściwości
 - interfejsu
 - IHostingEnvironment, 403
 - IModelBindingMessage
 - ↳ Provider, 842
 - klasy
 - ActionConstraintContext, 976
 - ActionDescriptor, 979
 - ActionExecutingContext, 591
 - ActionModel, 963
 - ActionSelectorCandidate, 979
 - ApplicationModel, 961
 - Claim, 933
 - ControllerContext, 516
 - ControllerModel, 962
 - FilterContext, 587
 - HtmlTargetElement, 723
 - HttpContext, 517
 - HttpRequest, 514
 - HttpResponse, 519
 - IdentityResult, 874
 - IdentityRole, 907
 - IdentityUser, 866
 - kontrolera, 513
 - ModelValidationContext, 852
 - MvcOptions, 422
 - ParameterModel, 963
 - PasswordOptions, 878
 - PropertyModel, 962
 - RazorPage<T>, 660
 - ResultExecutingContext, 595
 - TagHelperContext, 716
 - TagHelperOutput, 718, 728
 - TempData, 536
 - ViewComponentContext, 695
 - ViewContext, 650
 - ViewDataDictionary, 651
 - ViewLocationExpander
 - ↳ Context, 675
- silnika Razor, 661
- właściwość
 - @Model, 123
- klasy
 - AuthorizationFilterContext, 588
 - Layout, 126, 129
 - PostContent, 730
 - PreContent, 730
 - Repository, 577
 - TagHelperContext.Items, 736
- włączanie
 - atrybutów routingu, 463
 - atrybutu pomocniczego znacznika, 730
 - formatowania XML, 640
 - negocjacji treści, 644
 - obsługi
 - funkcji połączonych przeglądarek, 408
 - MVC, 387
 - sesji, 267
 - treści statycznej, 409
 - wyjątków, 406
 - szyfrowania SSL, 581
 - transformacji
 - oświadczenia, 937
 - uwierzytelniania, 951

WorkingWithVisualStudio
 atrybuty pomocniczych
 znaczników, 172
 dodanie
 akcji do kontrolera, 173
 odwołania, 177
 testy jednostkowe, 177
 uaktualnienie widoku, 174
 utworzenie
 formularza, 173
 kontrolera i widoku, 141
 modelu, 140
 wprowadzanie danych, 45
 wstawianie
 danych początkowych, 340,
 342
 wartości danych, 132
 wstrzykiwanie
 akcji, 575
 zależności, 220, 420, 545, 551,
 557
 dla konkretnego typu, 564
 metody rozszerzenia, 567
 pobieranie danych
 kontekstu, 732
 specjalizowane atrybuty,
 576
 użycie atrybutów, 575
 z filtrami, 603
 wybór
 arkusza stylów, 777
 przeglądarek WWW, 160
 widoków, 675
 źródła danych, 821
 wygaśnięcie buforowanej
 treści, 785
 wyjątki nieobsłużone, 153
 wykonywanie intencji, 520
 wyłączenie funkcji weryfikacji, 923
 wymagania polityki autoryzacji,
 942
 wynik działania komponentu
 widoku, 690
 wyniki akcji, 520, 544
 dla kodów stanu HTTP, 542
 dla plików, 540
 dla treści, 537
 wyrażenia
 lambda, 103, 107, 436
 metody i właściwości, 107

Razor, 131
 regularne
 ograniczanie trasy, 456
 wyrażenie
 @addTagHelper, 719
 @Json.Serialize, 671
 @Model, 124
 nameof, 116
 wyróżnianie
 bieżącej kategorii, 255
 pól z błędami, 61
 wysyłanie kodu 404, 543
 wyszukiwanie pliku widoku, 523
 wyświetlanie
 błędów kontroli
 poprawności, 843
 danych
 kontekstu, 515
 modelu, 552
 produktu, 308
 komunikatu potwierdzającego,
 312
 liczby stron, 260
 listy produktów, 210
 łączy stron, 227, 233
 odpowiedzi, 55
 strony podsumowania, 294
 użytkownikowi błędów, 838
 własnej zmiennej segmentu, 447
 zawartości
 koszyka, 270
 tablic i kolekcji, 136
 wywoływanie metody Configure(),
 401
 wyzerowanie bazy danych, 872
 wzorce
 dopasowania, 769, 771
 URL, 434
 konserwatywne, 435
 liberalne, 435
 wzorzec
 MVC, 69
 POST-przekierowanie-GET,
 535
 RESTful, 628
 Smart UI, 72
 URL
 z segmentem
 mieszanym, 441
 z segmentem
 statycznym, 440

X

XML, 640

Z

zabezpieczanie funkcji
 administracyjnych, 323
 zadania, 112
 niewłaściwe rozdzielanie, 26
 zależności, 561
 w ograniczeniu akcji, 981
 zamówienia, 297
 zaporą sieciową, 335
 zarządzanie
 arkuszami stylów CSS, 777
 katalogiem, 301
 pakietami, 142, 357
 pakietami .NET, 143
 plikami JavaScript, 768
 treścią, 768
 użytkownikami, 912
 zamówieniami, 297, 301
 zasięg atrybutu pomocniczego
 znacznika, 721
 zdalna kontrola poprawności, 856
 zewnętrzne pliki
 konfiguracyjne, 423
 liczenie produktów, 260
 zmiana
 adresu URL logowania, 900
 przeglądarki WWW, 38
 zmienne segmentu, 444
 opcjonalne, 449
 przechwytyjące, 451
 znacznik
 <a>, 500, 780
 <environment>, 767
 <image>, 781
 <input>, 747
 atrybut type, 749
 atrybutu type, 749
 konfigurowanie, 748
 <label>, 752
 <option>, 755
 <select>, 755
 <textarea>, 761
 znaczniki
 czasu, 159
 formularza, 744
 znak @, 123

zwracanie	Ż
błędów, 541	
fragmentów kodu HTML, 693	żądanie
kodów HTTP, 541	obiektu implementacji, 576
obiektu Task, 701	HTTP
widoku częściowego, 691	DELETE, 634
wyniku treści, 537	GET, 503, 631
Ż	PATCH, 633
źródło dołączania modelu, 819	POST, 310, 503, 632
	PUT, 633

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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 WYDAWNICZA

 **Helion SA**

ASP.NET Core MVC: rzecz dla najlepszych!

Framework ASP.NET Core MVC 2 powstał jako alternatywa ASP.NET Web Forms. Microsoft zbudował tę platformę całkowicie od podstaw. Dzięki zastosowaniu nowoczesnej architektury model – widok– kontroler programiści otrzymali narzędzie do szybszego tworzenia doskonalszego kodu. Łatwo się przekonać, że ten framework powstał wskutek kompletnej zmiany podejścia do technologii sieciowych: ASP.NET Core MVC 2 jest w pełni niezależny od platformy sprzętowej, a zastosowany wysoko produktywny model programowania zapewnia poprawną architekturę kodu, łatwe stosowanie testów jednostkowych oraz potężne możliwości rozbudowywania tworzonych systemów.

Ta książka jest kolejnym, zaktualizowanym i poprawionym wydaniem bardzo cenionego przez programistów podręcznika. Forma publikacji się nie zmieniła, jednak zawarte w niej informacje zostały gruntownie przejrane. Wyjaśniono najważniejsze koncepcje frameworka ASP.NET Core MVC 2. Omówiono budowę kompletnej i w pełni funkcjonalnej aplikacji ASP.NET Core MVC 2, którą można wykorzystać w charakterze szablonu we własnych projektach. Oprócz podstaw zaprezentowano tu także bardziej zaawansowane tematy, takie jak routing URL, kontrolery RESTful, stosowanie silnika Razor i wiele innych ważnych zagadnień.

W tej książce między innymi:

- solidne podstawy koncepcji MVC i ASP.NET Core MVC 2
- stosowanie najlepszych funkcji ASP.NET Core MVC 2 we własnych projektach
- praca z Visual Studio 2017, C# 7, Entity Framework 2, .NET Core 2 i Visual Studio Code
- modyfikacja klas C# i korzystanie z kontrolerów, akcji, filtrów
- konfiguracja ASP.NET Core Identity

Adam Freeman — jest wyjątkowo doświadczonym programistą i architektem. Doskonale rozumie wyzwania, związane z zapewnianiem bezpieczeństwa dużym systemom informatycznym. Pracował w wielu firmach, między innymi w Netscape i Sun Microsystems, ostatnio pełnił funkcję dyrektora naczelnego w międzynarodowym banku. Obecnie jest na emeryturze. Swoją czas dzieli między dwie pasje: pisanie i bieganie.

  helion.pl  HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	<i>Sprawdź nasze szkolenia!</i>  SZKOLENIA AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI <i>Sięgnij po więcej!</i>   ISBN 978-83-283-4600-0  9 788328 346000 Cena: 129,00 zł
INFORMATYKA W NAJLEPSZYM WYDANIU		

Apress