

KOMPENDIUM WIEDZY O PLATFORMIE ASP.NET MVC 4!

Apress®

ASP.NET MVC 4

Zaawansowane
programowanie

Adam Freeman

Helion



Tytuł oryginału: Pro ASP.NET MVC 4

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-7299-8

Original edition copyright © 2012 by Adam Freeman.
All rights reserved.

Polish edition copyright © 2013 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.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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/aspmv4.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/aspmv4>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

	O autorze	17
	O recenzencie technicznym	19
Część I	Wprowadzenie do ASP.NET MVC 4	21
Rozdział 1.	Zagadnienia ogólne	23
	Krótką historią programowania witryn WWW	23
	Tradycyjna technologia ASP.NET Web Forms	23
	Co poszło nie tak z ASP.NET Web Forms?	25
	Programowanie witryn WWW — stan obecny	25
	Standardy sieciowe oraz REST	26
	Programowanie zwinne i sterowane testami	26
	Ruby on Rails	27
	Sinatra	27
	Node.js	28
	Najważniejsze zalety ASP.NET MVC	28
	Architektura MVC	28
	Rozszerzalność	29
	Ścisła kontrola nad HTML i HTTP	29
	Łatwość testowania	30
	Zawansowany system routingu	30
	Zbudowany na najlepszych częściach platformy ASP.NET	30
	Nowoczesne API	31
	ASP.NET MVC jest open source	31
	Kto powinien korzystać z ASP.NET MVC?	31
	Porównanie z ASP.NET Web Forms	31
	Migracja z Web Forms do MVC	32
	Porównanie z Ruby on Rails	32
	Porównanie z MonoRail	32
	Co nowego w ASP.NET MVC 4?	33
	Podsumowanie	33

Rozdział 2. Pierwsza aplikacja MVC	35
Przygotowanie stacji roboczej	35
Tworzenie nowego projektu ASP.NET MVC	36
Dodawanie pierwszego kontrolera	38
Przedstawiamy ścieżki	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
Łączenie metod akcji	47
Budowanie formularza	49
Obsługa formularzy	51
Dodanie kontroli poprawności	54
Kończymy	60
Podsumowanie	61
Rozdział 3. Wzorzec MVC	63
Historia MVC	63
Wprowadzenie do wzorca MVC	64
Budowa modelu domeny	64
Implementacja MVC w ASP.NET	65
Porównanie MVC z innymi wzorcami	65
Przedstawiam wzorzec Smart UI	65
Modelowanie domeny	68
Przykładowy model domeny	69
Wspólny język	69
Agregaty i uproszczenia	70
Definiowanie repozytoriów	71
Budowanie luźno połączonych komponentów	73
Wykorzystanie wstrzykiwania zależności	73
Przykład specyficzny dla MVC	75
Użycie kontenera wstrzykiwania zależności	75
Zaczynamy testy automatyczne	76
Zadania testów jednostkowych	77
Zadania testów integracyjnych	84
Podsumowanie	84
Rozdział 4. Najważniejsze cechy języka	85
Utworzenie przykładowego projektu	85
Użycie automatycznie implementowanych właściwości	86
Użycie inicjalizatorów obiektów i kolekcji	89
Użycie metod rozszerzających	91
Stosowanie metod rozszerzających do interfejsów	93
Tworzenie filtrujących metod rozszerzających	95
Użycie wyrażeń lambda	97
Automatyczne wnioskowanie typów	100
Użycie typów anonimowych	100

Wykonywanie zapytań LINQ	102
Opóźnione zapytania LINQ	105
Użycie metod asynchronicznych	107
Użycie słów kluczowych async i await	108
Podsumowanie	109
Rozdział 5. Praca z silnikiem Razor	111
Tworzenie projektu	111
Definiowanie modelu	111
Definiowanie kontrolera	112
Tworzenie widoku	113
Korzystanie z obiektów modelu	114
Praca z układami	116
Tworzenie układu	116
Stosowanie układu	118
Użycie pliku ViewStart	119
Użycie układów współdzielonych	119
Użycie wyrażeń Razor	123
Wstawianie wartości danych	124
Przypisanie wartości atrybutu	125
Użycie konstrukcji warunkowych	127
Wyliczanie tablic i kolekcji	129
Praca z przestrzenią nazw	132
Podsumowanie	132
Rozdział 6. Ważne narzędzia wspierające MVC	133
Tworzenie przykładowego projektu	134
Utworzenie klas modelu	134
Dodanie kontrolera	135
Dodanie widoku	136
Użycie Ninject	136
Zrozumienie problemu	137
Zaczynamy korzystać z Ninject	139
Konfiguracja wstrzykiwania zależności na platformie MVC	141
Tworzenie łańcucha zależności	144
Definiowanie wartości właściwości i parametrów	146
Użycie łączenia warunkowego	148
Testy jednostkowe w Visual Studio	149
Tworzenie projektu testów jednostkowych	150
Tworzenie testów jednostkowych	151
Uruchamianie testów (nieudane)	154
Implementacja funkcji	155
Testowanie i poprawianie kodu	156
Użycie Moq	157
Zrozumienie problemu	158
Dodawanie Moq do projektu Visual Studio	159
Dodanie obiektu imitacyjnego do testu jednostkowego	160
Tworzenie obiektu imitacji	161
Tworzenie bardziej skomplikowanych obiektów Mock	163
Podsumowanie	166

Rozdział 7. SportsStore — kompletna aplikacja	167
Zaczynamy	168
Tworzenie rozwiązania i projektów w Visual Studio	168
Dodawanie referencji	169
Konfigurowanie kontenera DI	171
Uruchamiamy aplikację	172
Tworzenie modelu domeny	173
Tworzenie abstrakcyjnego repozytorium	174
Tworzenie imitacji repozytorium	174
Wyświetlanie listy produktów	175
Dodawanie kontrolera	176
Dodawanie widoku	177
Konfigurowanie domyślnej ścieżki	178
Uruchamianie aplikacji	179
Przygotowanie bazy danych	179
Tworzenie bazy danych	180
Definiowanie schematu bazy danych	181
Dodawanie danych do bazy	182
Tworzenie kontekstu Entity Framework	183
Tworzenie repozytorium produktów	184
Dodanie stronicowania	186
Wyświetlanie łączy stron	188
Ulepszanie adresów URL	195
Dodawanie stylu	196
Definiowanie wspólnej zawartości w pliku układu	197
Dodanie stylów CSS	197
Tworzenie widoku częściowego	199
Podsumowanie	200
Rozdział 8. SportsStore — nawigacja	201
Dodawanie kontrolki nawigacji	201
Filtrowanie listy produktów	201
Ulepszanie schematu URL	203
Budowanie menu nawigacji po kategoriach	207
Poprawianie licznika stron	214
Budowanie koszyka na zakupy	216
Definiowanie encji koszyka	217
Tworzenie przycisków koszyka	221
Implementowanie kontrolera koszyka	222
Wyświetlanie zawartości koszyka	224
Podsumowanie	227
Rozdział 9. SportsStore — ukończenie koszyka na zakupy	229
Użycie dołączania danych	229
Tworzenie własnego łącznika modelu	229
Kończenie budowania koszyka	234
Usuwanie produktów z koszyka	234
Dodawanie podsumowania koszyka	234

Składanie zamówień	237
Rozszerzanie modelu domeny	237
Dodawanie procesu zamawiania	238
Implementowanie mechanizmu przetwarzania zamówień	241
Rejestrowanie implementacji	243
Zakończenie pracy nad kontrolerem koszyka	244
Wyświetlanie informacji o błędach systemu kontroli poprawności	248
Wyświetlanie strony podsumowania	249
Podsumowanie	249
Rozdział 10. SportsStore — administracja	251
Dodajemy zarządzanie katalogiem	251
Tworzenie kontrolera CRUD	251
Tworzenie nowego pliku układu	253
Implementowanie widoku listy	255
Edycja produktów	260
Tworzenie nowych produktów	271
Usuwanie produktów	272
Podsumowanie	275
Rozdział 11. SportsStore — bezpieczeństwo i ostatnie usprawnienia	277
Zabezpieczanie funkcji administracyjnych	277
Realizacja uwierzytelniania z użyciem filtrów	278
Tworzenie dostawcy uwierzytelniania	280
Tworzenie kontrolera AccountController	281
Tworzenie widoku	283
Przesyłanie zdjęć	286
Rozszerzanie bazy danych	286
Rozszerzanie modelu domeny	287
Tworzenie interfejsu użytkownika do przesyłania plików	288
Zapisywanie zdjęć do bazy danych	289
Implementowanie metody akcji GetImage	290
Wyświetlanie zdjęć produktów	292
Podsumowanie	293
Część II ASP.NET MVC 4 — szczegółowy opis	295
Rozdział 12. Przegląd projektu MVC	297
Korzystanie z projektów MVC z Visual Studio	297
Przedstawienie konwencji MVC	301
Debugowanie aplikacji MVC	302
Tworzenie projektu	302
Uruchamianie debugera Visual Studio	304
Przerywanie pracy aplikacji przez debugger Visual Studio	306
Użycie opcji Edit and Continue	310
Podsumowanie	313
Rozdział 13. Routing URL	315
Tworzenie projektu routingu	316
Wprowadzenie do wzorców URL	318

Tworzenie i rejestrowanie prostej ścieżki	319
Użycie prostej ścieżki	324
Definiowanie wartości domyślnych	325
Użycie statycznych segmentów adresu URL	327
Definiowanie własnych zmiennych segmentów	331
Użycie własnych zmiennych jako parametrów metod akcji	334
Definiowanie opcjonalnych segmentów URL	335
Definiowanie ścieżek o zmiennej długości	337
Definiowanie priorytetów kontrolerów na podstawie przestrzeni nazw	339
Ograniczenia ścieżek	342
Ograniczanie ścieżki z użyciem wyrażeń regularnych	342
Ograniczanie ścieżki do zbioru wartości	343
Ograniczanie ścieżek z użyciem metod HTTP	343
Definiowanie własnych ograniczeń	344
Routing żądań dla plików dyskowych	346
Konfiguracja serwera aplikacji	347
Definiowanie ścieżek dla plików na dysku	349
Pomijanie systemu routingu	350
Podsumowanie	351
Rozdział 14. Zaawansowane funkcje routingu	353
Przygotowanie projektu	353
Generowanie wychodzących adresów URL w widokach	354
Użycie systemu routingu do wygenerowania wychodzącego adresu URL	354
Użycie innych kontrolerów	357
Przekazywanie dodatkowych parametrów	358
Definiowanie atrybutów HTML	360
Generowanie w pełni kwalifikowanych adresów URL w łączach	360
Generowanie adresów URL (nie łączy)	361
Generowanie wychodzących adresów URL w metodach akcji	362
Generowanie adresu URL na podstawie wybranej ścieżki	363
Dostosowanie systemu routingu	364
Tworzenie własnej implementacji RouteBase	364
Tworzenie własnego obiektu obsługi ścieżki	368
Korzystanie z obszarów	369
Tworzenie obszaru	369
Wypełnianie obszaru	371
Rozwiązywanie problemów z niejednoznacznością kontrolerów	373
Generowanie łącz do akcji z obszarów	374
Najlepsze praktyki schematu adresów URL	375
Twórz jasne i przyjazne dla człowieka adresy URL	375
GET oraz POST — wybierz właściwie	376
Podsumowanie	376
Rozdział 15. Kontrolery i akcje	377
Wprowadzenie do kontrolerów	377
Przygotowanie projektu	377
Tworzenie kontrolera z użyciem interfejsu IController	378
Tworzenie kontrolera przez dziedziczenie po klasie Controller	379

Odczytywanie danych wejściowych	380
Pobieranie danych z obiektów kontekstu	380
Użycie parametrów metod akcji	382
Tworzenie danych wyjściowych	383
Wyniki akcji	385
Zwracanie kodu HTML przez generowanie widoku	388
Przekazywanie danych z metody akcji do widoku	391
Wykonywanie przekierowań	394
Zwracanie błędów i kodów HTTP	399
Podsumowanie	401
Rozdział 16. Filtry	403
Użycie filtrów	403
Wprowadzenie do czterech podstawowych typów filtrów	404
Dołączanie filtrów do kontrolerów i metod akcji	405
Tworzenie projektu	406
Użycie filtrów autoryzacji	407
Użycie własnego filtra autoryzacji	409
Użycie wbudowanego filtra autoryzacji	409
Użycie filtrów wyjątków	410
Tworzenie filtra wyjątku	411
Użycie filtra wyjątków	412
Użycie widoku w celu reakcji na wyjątek	414
Użycie wbudowanego filtra wyjątków	417
Użycie filtrów akcji	419
Implementacja metody OnActionExecuting	420
Implementacja metody OnActionExecuted	421
Używanie filtra wyniku	423
Użycie wbudowanych klas filtrów akcji i wyniku	424
Użycie innych funkcji filtrów	425
Filtrowanie bez użycia atrybutów	426
Użycie filtrów globalnych	427
Określanie kolejności wykonywania filtrów	429
Użycie filtrów wbudowanych	431
Użycie filtra RequireHttps	432
Użycie filtra OutputCache	432
Podsumowanie	435
Rozdział 17. Rozszerzanie kontrolerów	437
Tworzenie projektu	437
Tworzenie własnej fabryki kontrolerów	439
Przygotowanie kontrolera zapasowego	441
Utworzenie klasy kontrolera	442
Implementacja innych metod interfejsu	442
Rejestrowanie własnej fabryki kontrolerów	442
Wykorzystanie wbudowanej fabryki kontrolerów	443
Nadawanie priorytetów przestrzeniom nazw	444
Dostosowywanie sposobu tworzenia kontrolerów w DefaultControllerFactory	445
Tworzenie własnego obiektu wywołującego akcje	447

Użycie wbudowanego obiektu wywołującego akcje	449
Użycie własnych nazw akcji	450
Selekcja metod akcji	451
Poprawianie wydajności z użyciem specjalizowanych kontrolerów	456
Użycie kontrolerów bezstanowych	456
Użycie kontrolerów asynchronicznych	458
Podsumowanie	463
Rozdział 18. Widoki	465
Tworzenie własnego silnika widoku	465
Tworzenie przykładowego projektu	467
Tworzenie własnej implementacji IView	468
Tworzenie implementacji IViewEngine	468
Rejestrowanie własnego silnika widoku	470
Testowanie silnika widoku	470
Korzystanie z silnika Razor	472
Tworzenie przykładowego projektu	472
Sposób generowania widoków przez Razor	473
Konfigurowanie wyszukiwania lokalizacji widoków	475
Dodawanie dynamicznych treści do widoku Razor	477
Zastosowanie sekcji	478
Użycie widoków częściowych	483
Użycie akcji podrzędnych	486
Podsumowanie	488
Rozdział 19. Metody pomocnicze	489
Tworzenie przykładowego projektu	489
Tworzenie własnej metody pomocniczej	491
Tworzenie wewnętrznej metody pomocniczej HTML	491
Tworzenie zewnętrznej metody pomocniczej HTML	492
Zarządzanie kodowaniem ciągów tekstowych w metodzie pomocniczej	496
Użycie wbudowanych metod pomocniczych	500
Przygotowania do obsługi formularzy	500
Określenie ścieżki używanej przez formularz	507
Użycie metod pomocniczych do wprowadzania danych	508
Tworzenie znaczników select	513
Podsumowanie	515
Rozdział 20. Szablonowe metody pomocnicze	517
Przegląd przykładowego projektu	517
Używanie szablonowych metod pomocniczych	519
Generowanie etykiety i wyświetlanie elementów	522
Użycie szablonowych metod pomocniczych dla całego modelu	525
Użycie metadanych modelu	527
Użycie metadanych do sterowania edycją i widocznością	528
Użycie metadanych dla etykiet	530
Użycie metadanych wartości danych	531
Użycie metadanych do wybierania szablonu wyświetlania	533
Dodawanie metadanych do klasy zaprzyjaźnionej	535
Korzystanie z parametrów typów złożonych	536

Dostosowywanie systemu szablonowych metod pomocniczych	538
Tworzenie własnego szablonu edytora	538
Tworzenie szablonu ogólnego	539
Wymiana szablonów wbudowanych	540
Podsumowanie	541
Rozdział 21. Metody pomocnicze URL i Ajax	543
Przegląd i przygotowanie projektu	543
Tworzenie podstawowych łączy i adresów URL	545
Nieprzeszkadzający Ajax	547
Tworzenie widoku formularza synchronicznego	547
Włączanie i wyłączanie nieprzeszkadzających wywołań Ajax	549
Użycie nieprzeszkadzających formularzy Ajax	550
Przygotowanie kontrolera	550
Tworzenie formularza Ajax	552
Sposób działania nieprzeszkadzających wywołań Ajax	553
Ustawianie opcji Ajax	554
Zapewnienie kontrolowanej degradacji	554
Informowanie użytkownika o realizowanym żądaniu Ajax	556
Wyświetlanie pytania przed wysłaniem żądania	557
Tworzenie łączy Ajax	558
Zapewnienie kontrolowanej degradacji dla łączy	559
Korzystanie z funkcji wywołania zwrotnego w Ajaksie	560
Wykorzystanie JSON	562
Dodanie obsługi JSON do kontrolera	562
Przetwarzanie JSON w przeglądarce	564
Przygotowanie danych do kodowania	565
Wykonywanie żądań Ajax w metodach akcji	567
Podsumowanie	570
Rozdział 22. Dołączanie modelu	571
Przygotowanie projektu	571
Użycie dołączania modelu	573
Użycie domyślnego łącznika modelu	575
Dołączanie typów prostych	575
Dołączanie typów złożonych	578
Dołączanie tablic i kolekcji	584
Jawne wywoływanie dołączania modelu	589
Obsługa błędów dołączania modelu	590
Dostosowanie systemu dołączania modelu	591
Tworzenie własnego dostawcy wartości	591
Tworzenie własnego łącznika modelu	594
Rejestracja własnego łącznika modelu	596
Podsumowanie	597
Rozdział 23. Kontrola poprawności modelu	599
Tworzenie projektu	599
Jawna kontrola poprawności modelu	602
Wyświetlenie użytkownikowi błędów podczas kontroli poprawności	603

Wyświetlanie komunikatów kontroli poprawności	604
Wyświetlanie komunikatów kontroli poprawności poziomu właściwości	608
Użycie alternatywnych technik kontroli poprawności	608
Kontrola poprawności w łączniku modelu	609
Definiowanie zasad poprawności za pomocą metadanych	610
Definiowanie modeli samokontrolujących się	616
Użycie kontroli poprawności po stronie klienta	618
Aktywowanie i wyłączanie kontroli poprawności po stronie klienta	618
Użycie kontroli poprawności po stronie klienta	619
Jak działa kontrola poprawności po stronie klienta?	620
Wykonywanie zdalnej kontroli poprawności	621
Podsumowanie	624
Rozdział 24. Paczki i tryby wyświetlania	625
Domyślne biblioteki skryptów	625
Tworzenie przykładowej aplikacji	627
Zarządzanie skryptami i stylami	630
Profilowanie wczytywania skryptów i arkuszy stylów	630
Używanie paczek stylów i skryptów	632
Stosowanie paczek	635
Używanie sekcji script	637
Profilowanie wprowadzonych zmian	638
Przygotowanie aplikacji dla urządzeń mobilnych	640
Aplikacja standardowa	641
Użycie widoków i układów przeznaczonych dla urządzeń mobilnych	642
Tworzenie własnych trybów wyświetlania	643
Podsumowanie	646
Rozdział 25. Web API	647
Zrozumienie Web API	647
Tworzenie aplikacji Web API	648
Tworzenie modelu i repozytorium	648
Tworzenie kontrolera Home	650
Utworzenie widoku i CSS	651
Tworzenie kontrolera API	653
Testowanie kontrolera API	655
Jak działa kontroler API?	656
Jak wybierana jest akcja kontrolera API?	657
Mapowanie metod HTTP na metody akcji	657
Tworzenie kodu JavaScript wykorzystującego interfejs Web API	658
Tworzenie funkcji podstawowych	659
Dodanie obsługi edycji nowych rezerwacji	661
Dodanie obsługi usuwania rezerwacji	663
Dodanie obsługi tworzenia rezerwacji	664
Podsumowanie	665
Rozdział 26. Wdrażanie aplikacji	667
Przygotowanie aplikacji do dystrybucji	668
Wykrywanie błędów widoku	668
Wyłączanie trybu debugowania	670
Usunięcie nieużywanych ciągów tekstowych połączenia	670

Przygotowanie do użycia Windows Azure	671
Tworzenie witryny internetowej i bazy danych	672
Przygotowanie bazy danych do zdalnej administracji	674
Tworzenie schematu bazy danych	674
Wdrażanie aplikacji	676
Podsumowanie	680
Skorowidz	681

ROZDZIAŁ 13.



Routing URL

Przed wprowadzeniem platformy MVC założono w ASP.NET, że istnieje bezpośrednia relacja pomiędzy adresem URL żądania a plikiem na dysku serwera. Zadaniem serwera było odczytanie żądania wysłanego przez przeglądarkę i dostarczenie wyniku z odpowiedniego pliku.

Żądany URL	Odpowiadający mu plik
<code>http://witryna.pl/default.aspx</code>	<code>e:\webroot\default.aspx</code>
<code>http://witryna.pl/admin/login.aspx</code>	<code>e:\webroot\admin\login.aspx</code>
<code>http://witryna.pl/articles/AnnualReview</code>	Plik nie został znaleziony! Wygeneruj błąd 404.

Podejście to działa świetnie dla Web Forms, gdzie każda strona ASPX jest plikiem i zawiera odpowiedź na żądanie. Nie ma to sensu dla aplikacji MVC, w których żądania są przetwarzane przez metody akcji w klasach kontrolera i nie ma bezpośredniej korelacji z plikami na dysku.

Aby obsługiwać adresy URL MVC, platforma ASP.NET korzysta z *systemu routingu*. W tym rozdziale pokażę, jak skonfigurować i wykorzystywać routing w celu utworzenia zaawansowanego i elastycznego systemu obsługi adresów URL dla naszych projektów. Jak się przekonasz, system routingu oferuje możliwość tworzenia dowolnych wzorców URL i opisywania ich w jasny i spójny sposób. System routingu ma dwie funkcje:

- Analiza *przychodzącego* żądania URL i określenie kontrolera i akcji przeznaczonych dla tego żądania. Jak można się spodziewać, jest to oczekiwana akcja w przypadku otrzymania żądania klienta.
- Generowanie *wychodzących* adresów URL. Są to adresy URL pojawiające się w stronach HTML generowanych na podstawie naszych widoków, dzięki czemu po kliknięciu łącza przez użytkownika generowane są odpowiednie akcje (i stają się ponownie przychodzącymi żądaniem URL).

W pierwszej części tego rozdziału skupimy się na definiowaniu ścieżek i korzystaniu z nich do przetwarzania przychodzących adresów URL, dzięki którym użytkownik wywołuje nasze kontrolery i akcje. Następnie pokażę, w jaki sposób korzystać z tych samych ścieżek do wygenerowania wychodzących adresów URL, które musimy dołączać do widoków. Dowiesz się również, jak system routingu dostosować do własnych potrzeb i jak używać funkcji o nazwie *obszary*.

Tworzenie projektu routingu

Aby zademonstrować działanie systemu routingu, potrzebujemy projektu, w którym możemy dodawać ścieżki. Na potrzeby tego rozdziału stworzymy aplikację MVC z wykorzystaniem szablonu *Podstawowe* i nadajemy jej nazwę *UrlsAndRoutes*.

-
- **Wskazówka** W tym rozdziale znajdziesz wiele różnych testów jednostkowych. Jeżeli chcesz je wykorzystać, to podczas tworzenia nowego projektu musisz zaznaczyć pole wyboru *Utwórz projekt testu jednostki*, a następnie użyć menedżera pakietów NuGet w celu dodania biblioteki Moq do projektu.
-

W celu zademonstrowania funkcji routingu konieczne jest dodanie kilku prostych kontrolerów do utworzonej przed chwilą aplikacji. W rozdziale koncentrujemy się jedynie na sposobie interpretacji adresów URL w celu wywołania metod akcji. Jako modeli widoków będziemy więc używać ciągów tekstowych zdefiniowanych w *ViewBag*, które podają nazwę kontrolera i metody akcji. Jako pierwszy utwórz kontroler *HomeController* i umieść w nim kod przedstawiony na listingu 13.1.

Listing 13.1. Kod kontrolera HomeController

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Następnie utwórz kontroler *CustomerController* i umieść w nim kod przedstawiony na listingu 13.2.

Listing 13.2. Kod kontrolera CustomerController

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

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```



```

        public ActionResult List() {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}

```

Utwórz kolejny kontroler i nadaj mu nazwę `AdminController`, a następnie umieść w nim kod przedstawiony na listingu 13.3.

Listing 13.3. Kod kontrolera `AdminController`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class AdminController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Admin";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}

```

We wszystkich metodach akcji utworzonych kontrolerów został użyty widok `ActionName`, który pozwala na zdefiniowanie jednego widoku i jego użycie w całej aplikacji. Do katalogu `/Views/Shared` projektu dodaj więc widok o nazwie `ActionName.cshtml` i umieść w nim kod przedstawiony na listingu 13.4.

Listing 13.4. Kod widoku `ActionName.cshtml`

```

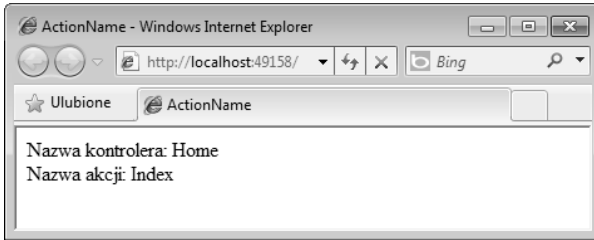
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>Nazwa kontrolera: @ViewBag.Controller</div>
    <div>Nazwa akcji: @ViewBag.Action</div>
</body>
</html>

```

Po uruchomieniu aplikacji otrzymasz komunikaty widoczne na rysunku 13.1.



Rysunek 13.1. Efekt uruchomienia przykładowej aplikacji

Wprowadzenie do wzorców URL

System routingu działa dzięki wykorzystaniu zbioru *ścieżek*. Ścieżki te są nazywane *schematem URL* dla aplikacji i definiują zbiór adresów URL, jakie aplikacja rozpoznaje i na jakie odpowiada.

Nie musimy ręcznie wpisywać wszystkich adresów URL, jakie chcemy obsługiwać. Zamiast tego każda ścieżka zawiera *wzorzec URL*, który jest porównywany z przychodzącym adresem URL. Jeżeli wzorzec pasuje do adresu, jest używany do przetworzenia tego adresu URL. Zacznijmy od przykładowego adresu URL aplikacji utworzonej w rozdziale:

`http://witryna.pl/Admin/Index`

Adresy URL mogą być podzielone na *segmenty*. Są to te części adresu URL, które są rozdzielane znakiem / z pominięciem nazwy hosta oraz ciągu zapytania. W przykładowym adresie URL występują dwa segmenty, jak pokazano na rysunku 13.2.

`http://witryna.pl/Admin/Index`

↑
↑
 Pierwszy segment Drugi segment

Rysunek 13.2. Segmenty przykładowego adresu URL

Pierwszy segment zawiera słowo *Admin*, a drugi słowo *Index*. Dla ludzkiego oka jest oczywiste, że pierwszy argument odnosi się do kontrolera, a drugi do akcji. Jasne jest, że musimy wyrazić tę relację w sposób zrozumiały dla systemu routingu. Wzorzec URL realizujący to zadanie wygląda następująco:

```
{controller}/{action}
```

W czasie przetwarzania przychodzącego adresu URL zadaniem systemu routingu jest dopasowanie adresu URL do wzorca oraz pobranie wartości do *zmiennych segmentu* zdefiniowanych we wzorcu. Zmienne segmentu są zapisywane z użyciem nawiasów klamrowych (znaków { oraz }). Przykładowy wzorzec zawiera dwie zmienne segmentu, o nazwach `controller` i `action`. Dlatego też wartością zmiennej segmentu `controller` jest `Admin`, natomiast wartością zmiennej segmentu `action` jest `Index`.

Mówimy o dopasowaniu wzorca, ponieważ aplikacja MVC zwykle zawiera kilka ścieżek, a system routingu będzie dopasowywał przychodzący adres URL do wzorca kolejnych ścieżek do momentu znalezienia dopasowania.

-
- **Uwaga** System routingu nie posiada żadnych informacji na temat kontrolerów i akcji. Po prostu pobiera wartości do zmiennych segmentów i przekazuje je do potoku żądania. W dalszej części potoku, gdy żądanie trafi do platformy MVC, są one związane ze zmiennymi kontrolera i akcji. Dzięki temu system routingu może być używany w Web Forms i Web API (interfejs Web API zostanie omówiony w rozdziale 25.).
-

Domyślnie wzorce URL są dopasowywane do dowolnego adresu URL mającego właściwą liczbę segmentów. Wzorzec {controller}/{action} jest dopasowywany do dowolnego adresu URL z dwoma segmentami, jak pokazano w tabeli 13.1.

Tabela 13.1. Dopasowanie adresów URL

Żądany URL	Zmienne segmentu
<code>http://witryna.pl/Admin/Index</code>	controller = Admin action = Index
<code>http://witryna.pl/Index/Admin</code>	controller = Index action = Admin
<code>http://witryna.pl/Apples/Oranges</code>	controller = Apples action = Oranges
<code>http://witryna.pl/Admin</code>	Brak dopasowania — za mało segmentów
<code>http://witryna.pl/Admin/Index/Apples</code>	Brak dopasowania — za dużo segmentów

W tabeli 13.1 przedstawione są dwie kluczowe cechy wzorców URL:

- Wzorce URL są *konserwatywne* i pasują wyłącznie do adresów, które mają taką samą liczbę segmentów jak wzorzec. Można to zauważyć w czwartym i piątym przykładzie z tabeli.
- Wzorce URL są *liberalne*. Jeżeli adres URL posiada prawidłową liczbę segmentów, zostanie pobrana wartość zmiennej segmentu, niezależnie od tego, jaka ta wartość jest.

Są to kluczowe zależności, które trzeba znać, aby zrozumieć sposób domyślnego działania wzorców URL. W dalszej części rozdziału wyjaśnimy, jak zmienić to domyślne działanie.

Jak wspomniałem, system routingu nie ma żadnych informacji na temat aplikacji MVC, dlatego wzorce URL będą dopasowywane nawet w przypadku, gdy nie ma kontrolera lub akcji pasującej do wartości pobranych z adresu URL. Jest to pokazane w drugim przykładzie z tabeli 13.1. Zamieniliśmy w nim segmenty *Admin* i *Index*, przez co również wartości pobrane z URL są zamienione, pomimo że w omawianym projekcie nie ma kontrolera *Index*.

Tworzenie i rejestrowanie prostej ścieżki

Po zapoznaniu się z wzorcami URL możemy użyć ich do zdefiniowania ścieżki. Ścieżki są definiowane w pliku *RouteConfig.cs*, który znajduje się w katalogu *App_Start* projektu. Początkowy kod wspomnianego pliku wygenerowany przez Visual Studio przedstawiono na listingu 13.5.

Listing 13.5. Domyślny kod w pliku *RouteConfig.cs*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
```

```

        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
}
}

```

Zdefiniowana w pliku *RouteConfig.cs* metoda statyczna `RegisterRoutes` jest wywoływana z pliku *Global.asax.cs*, który konfiguruje podstawowe komponenty platformy MVC podczas uruchamiania aplikacji. Domyślna zawartość pliku *Global.asax.cs* została przedstawiona na listingu 13.6, a wywołanie metody `RouteConfig.RegisterRoutes` z metody `Application_Start` oznaczono pogrubioną czcionką.

Listing 13.6. Domyślna zawartość pliku *Global.asax.cs*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}

```

Metoda `Application_Start` jest wywoływana przez platformę ASP.NET w trakcie pierwszego uruchomienia aplikacji MVC, co prowadzi do wywołania metody `RouteConfig.RegisterRoutes`. Parametrem metody jest wartość właściwości statycznej `RouteTable.Routes`, która jest egzemplarzem klasy `RouteCollection` (funkcje wymienionej klasy zostaną wkrótce przedstawione).

-
- **Wskazówka** Pozostałe wywołania w metodzie `Application_Start` zostaną omówione w innych rozdziałach. Wywołanie metody `AreaRegistration.RegisterAllAreas` przedstawiłem w rozdziale 14., metody `WebApiConfig.Register` w rozdziale 25., metody `FilterConfig.RegisterGlobalFilters` w rozdziale 16., natomiast metody `BundleConfig.RegisterBundles` w rozdziale 24.
-

Na listingu 13.7 pokazałem, w jaki sposób możemy utworzyć ścieżkę w metodzie `RegisterRoutes` z pliku *RouteConfig.cs* za pomocą przykładowego wzorca URL z poprzedniego punktu. (Pozostałe polecenia w metodzie zostały usunięte, aby umożliwić Ci skoncentrowanie się na przykładzie).

Listing 13.7. Rejestrowanie ścieżki

```

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

```

```

using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            Route myRoute = new Route("{controller}/{action}",
                new MvcRouteHandler());
            routes.Add("MyRoute", myRoute);

        }
    }
}

```

Tworzymy tu nowy obiekt ścieżki, przekazując do konstruktora wzorzec URL jako parametr. Przekazaliśmy do niego również obiekt `MvcRouteHandler`. Różne technologie ASP.NET zawierają różne klasy do obsługi routingu; w aplikacjach ASP.NET MVC będziemy używać właśnie tej klasy. Utworzoną ścieżkę dodajemy do obiektu `RouteCollection` za pomocą metody `Add` — przekazujemy nazwę, pod jaką powinna być zarejestrowana ścieżka, oraz samą ścieżkę.

-
- **Wskazówka** Nazywanie ścieżek jest opcjonalne i podnoszone są argumenty, że w ten sposób poświęca się czystą separację zadań, którą można uzyskać przy użyciu systemu routingu. Osobiście nie przywiązuję wielkiej wagi do kwestii nazywania ścieżek, ale na wszelki wypadek przedstawiam związane z tym problemy w punkcie „Generowanie adresu URL na podstawie wybranej ścieżki”, w dalszej części rozdziału.
-

Wygodniejszą metodą rejestrowania ścieżek jest użycie metody `MapRoute`, zdefiniowanej w klasie `RouteCollection`. Na listingu 13.8 przedstawione jest zastosowanie tej metody do zarejestrowania naszej ścieżki. Otrzymaony efekt jest dokładnie taki sam jak w poprzednim przykładzie, ale sama składnia jest bardziej przejrzysta.

Listing 13.8. Rejestrowanie ścieżki za pomocą metody `MapRoute`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}");

        }
    }
}

```

Podejście takie jest nieco bardziej zwarte, głównie dlatego, że nie trzeba tworzyć obiektu klasy `MvcRouteHandler`. Metoda `MapRoute` jest przeznaczona wyłącznie dla aplikacji MVC. Aplikacje ASP.NET Web Forms mogą korzystać z metody `MapPageRoute`, zdefiniowanej również w klasie `RouteCollection`.

Test jednostkowy — testowanie przychodzących adresów URL

Zalecam, aby nawet w przypadku, gdy nie tworzymy testów jednostkowych dla reszty aplikacji, tworzyć testy jednostkowe dla ścieżek, dzięki czemu można się upewnić, że przetwarzanie przychodzących adresów URL działa w oczekiwany sposób. Schematy URL mogą być dosyć rozbudowane w dużych aplikacjach, więc łatwo jest utworzyć coś, co będzie dawało nieoczekiwane wyniki.

W poprzednich rozdziałach unikałem tworzenia metod pomocniczych, współdzielonych przez wiele testów, aby każdy test był niezależny. W tym rozdziale przyjmiemy inne podejście. Testowanie schematu routingu dla aplikacji będzie realizowane w najbardziej czytelny sposób, gdy połączymy kilka testów w jedną metodę. Najłatwiej możemy to zrealizować przy użyciu metod pomocniczych.

Aby testować ścieżki, musimy utworzyć imitację trzech klas: `HttpRequestBase`, `HttpContextBase` oraz `HttpResponseBase` (ostatnia z nich jest potrzebna do testowania wychodzących adresów URL, które przedstawię w następnym rozdziale). Klasy te pozwalają odtworzyć fragment infrastruktury MVC obsługującej system routingu. Do projektu testowego dodajemy nowy plik testów jednostkowych o nazwie `RouteTests.cs`. Poniżej zamieszczona jest metoda pomocnicza tworząca imitację obiektów `HttpContextBase`:

```
using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web;
using System.Web.Routing;
using Moq;
using System.Reflection;

namespace UrlsAndRoutes.Tests {
    [TestClass]
    public class RouteTests {

        private HttpContextBase CreateHttpContext(string targetUrl = null,
                                                string httpMethod = "GET") {

            // tworzenie imitacji żądania
            Mock<HttpRequestBase> mockRequest = new Mock<HttpRequestBase>();
            mockRequest.Setup(m => m.AppRelativeCurrentExecutionFilePath)
                .Returns(targetUrl);
            mockRequest.Setup(m => m.HttpMethod).Returns(httpMethod);

            // tworzenie imitacji odpowiedzi
            Mock<HttpResponseBase> mockResponse = new Mock<HttpResponseBase>();
            mockResponse.Setup(m => m.ApplyAppPathModifier(
                It.IsAny<string>())) .Returns<string>(s => s);

            // tworzenie imitacji kontekstu z użyciem żądania i odpowiedzi
            Mock<HttpContextBase> mockContext = new Mock<HttpContextBase>();
            mockContext.Setup(m => m.Request).Returns(mockRequest.Object);
            mockContext.Setup(m => m.Response).Returns(mockResponse.Object);

            // zwraca imitację kontekstu
            return mockContext.Object;
        }
    }
}
```

Konfiguracja jest prostsza, niż się wydaje. Udostępniamy URL do testowania poprzez właściwość `AppRelativeCurrentExecutionFilePath` klasy `HttpRequestBase`; udostępniamy także `HttpRequestBase` poprzez właściwość `Request` imitacji klasy `HttpContextBase`.

Nasza następna metoda pomocnicza pozwala testować ścieżkę:

```
...
private void TestRouteMatch(string url, string controller, string action,
    object routeProperties = null, string httpMethod = "GET") {
    // przygotowanie
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // działanie — przetwarzanie ścieżki
    RouteData result = routes.GetRouteData(CreateHttpContext(url, httpMethod));
    // asercje
    Assert.IsNotNull(result);
    Assert.IsTrue(TestIncomingRouteResult(result, controller,
        action, routeProperties));
}
...
```

Parametr tej metody pozwala nam określić adres URL do testowania, oczekiwane wartości dla zmiennych segmentów kontrolera i akcji oraz obiekt zawierający oczekiwane wartości dowolnych innych zdefiniowanych zmiennych. Sposób tworzenia takich zmiennych pokażę w dalszej części rozdziału. Zdefiniowaliśmy również parametr dla metody HTTP, którego użyjemy w punkcie „Ograniczanie ścieżek”.

Metoda `TestRouteMatch` bazuje na innej metodzie, `TestIncomingRouteResult`, która porównuje wyniki uzyskane z systemu routingu z oczekiwanymi wartościami zmiennych segmentów. Metody te korzystają z refleksji .NET, dzięki czemu możemy używać typów anonimowych do definiowania dodatkowych zmiennych segmentów. Jeżeli to, co tu napisałem, nie ma dla Ciebie sensu, nie przejmuj się, nie jest to wymagane do zrozumienia mechanizmów MVC, lecz jedynie ułatwia testowanie. Poniżej zamieszczona jest metoda `TestIncomingRouteResult`:

```
...
private bool TestIncomingRouteResult(RouteData routeResult, string controller,
    string action, object propertySet = null) {
    Func<object, object, bool> valCompare = (v1, v2) => {
        return StringComparer.InvariantCultureIgnoreCase.Compare(v1, v2) == 0;
    };
    bool result = valCompare(routeResult.Values["controller"], controller)
        && valCompare(routeResult.Values["action"], action);
    if (propertySet != null) {
        PropertyInfo[] propInfo = propertySet.GetType().GetProperties();
        foreach (PropertyInfo pi in propInfo) {
            if (!(routeResult.Values.ContainsKey(pi.Name)
                && valCompare(routeResult.Values[pi.Name],
                    pi.GetValue(propertySet, null)))) {
                result = false;
                break;
            }
        }
    }
    return result;
}
...
```

Potrzebujemy również sprawdzić nie działający adres URL. Jak pokażę, może to być ważna część definiowania schematu URL.

```
...
private void TestRouteFail(string url) {
    // przygotowanie
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // działanie — przetwarzanie ścieżki
    RouteData result = routes.GetRouteData(CreateHttpContext(url));
    // asercje
    Assert.IsTrue(result == null || result.Route == null);
}
...
```

Metody `TestRouteMatch` oraz `TestRouteFail` zawierają wywołania metody `Assert`, która zgłasza wyjątek, jeżeli asercja się nie powiedzie. Ponieważ wyjątki C# są propagowane w górę stosu, możemy utworzyć prostą metodę testową, która pozwoli sprawdzić zestaw adresów URL. Poniżej znajduje się metoda testująca ścieżkę zdefiniowaną na listingu 13.8.

```
...
[TestMethod]
public void TestIncomingRoutes() {

    // sprawdzenie, czy otrzymamy adres URL, jakiego oczekiwaliśmy
    TestRouteMatch("~/Admin/Index", "Admin", "Index");
    // sprawdzenie wartości uzyskanych z segmentów
    TestRouteMatch("~/One/Two", "One", "Two");

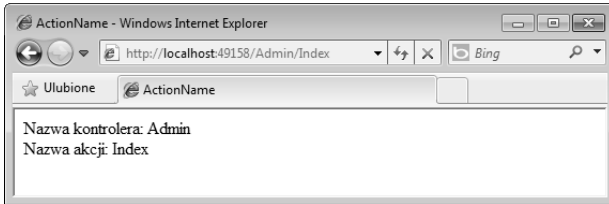
    // upewnienie się, że za mało lub za dużo segmentów spowoduje błąd dopasowania
    TestRouteFail("~/Admin/Index/Segment");
    TestRouteFail("~/Admin");
}
...
```

`Test ten` korzysta z metody `TestRouteMatch` do sprawdzenia oczekiwanego adresu URL, a także do sprawdzenia adresu w tym samym formacie, aby można było się upewnić, że wartości kontrolera i akcji są pozyskiwane w prawidłowych segmentach URL. Wykorzystaliśmy również metodę `TestRouteFail` w celu upewnienia się, że nasza aplikacja nie zaakceptuje adresów URL mających inną liczbę segmentów. Przy testowaniu musimy poprzedzić adres URL znakiem tyldy (-), ponieważ w taki sposób platforma ASP.NET prezentuje adresy URL systemowi routingu.

Zwróć uwagę, że nie musimy definiować ścieżek w metodach testowych. Ładujemy tu ścieżki bezpośrednio z metody `RegisterRoutes`, zdefiniowanej w klasie `RouteConfig`.

Użycie prostej ścieżki

Możemy zobaczyć efekt działania utworzonych ścieżek przez uruchomienie aplikacji. Gdy przeglądarka zażąda głównego adresu URL, aplikacja zwróci błąd. Jeżeli jednak podasz ścieżkę dopasowaną do wzorca `{controller}/{action}`, wówczas otrzymasz wynik pokazany na rysunku 13.3. Na wspomnianym rysunku pokazano efekt przejścia w aplikacji do adresu URL `/Admin/Index`.



Rysunek 13.3. Nawigacja za pomocą prostej ścieżki

Nasza prosta ścieżka nie informuje platformy MVC, w jaki sposób ma odpowiadać na żądania dotyczące głównego adresu URL, i obsługuje tylko jeden, konkretny wzorec URL. Tymczasowo wykonaliśmy więc krok wstecz względem funkcjonalności zdefiniowanej przez Visual Studio w pliku *RouteConfig.cs* podczas tworzenia projektu MVC. W dalszej części rozdziału pokażę, jak tworzyć bardziej złożone ścieżki i wzorce.

Definiowanie wartości domyślnych

Powodem pojawienia się błędu w przypadku domyślnego adresu URL dla aplikacji jest brak dopasowania do zdefiniowanej przez nas ścieżki. Domyślny adres URL jest przedstawiany systemowi routingu jako *~/*, więc nie ma w nim segmentów, które mogłyby być dopasowane do zmiennych *controller* oraz *action*.

Jak wcześniej wyjaśniłem, wzorce URL są konserwatywne, więc pasują wyłącznie do adresów URL o zdefiniowanej liczbie segmentów. Wspominałem również, że jest to domyślne zachowanie. Jednym ze sposobów zmiany tego zachowania jest użycie *wartości domyślnych*. Wartości domyślne są stosowane, gdy adres URL nie zawiera segmentu, który można dopasować do wartości. Na listingu 13.9 zamieściłem przykład ścieżki zawierającej wartość domyślną.

-
- **Uwaga** Od tej chwili, gdy pokazuję jakąkolwiek nową konfigurację routingu, wszelkie zmiany musisz wprowadzać w metodzie *RegisterRoutes* klasy *RouteConfig*.
-

Listing 13.9. Określanie wartości domyślnej w ścieżce

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { action = "Index" });
        }
    }
}
```

Wartości domyślne są dostarczane jako właściwości w typie anonimowym. Na listingu 13.9 zdefiniowaliśmy wartość domyślną *Index* dla zmiennej *action*. Ścieżka ta będzie dopasowywana do wszystkich dwusegmentowych adresów URL, tak jak poprzednio. Gdy zażądamy na przykład adresu URL *http://witryna.pl/Home/Index*, ścieżka pobierze *Home* jako wartość dla *controller* oraz *Index* jako wartość *action*.

Teraz mamy jednak przekazaną wartość domyślną dla segmentu `action`, więc ścieżka będzie dopasowywana również dla jednosegmentowych adresów URL. Przetwarzając adres URL, system routingu pobierze wartość zmiennej `controller` z jedyne go segmentu adresu URL oraz użyje wartości domyślnej dla zmiennej `action`. Zatem gdy zażądamy adresu URL `http://witryna.pl/Home`, zostanie wywołana metoda akcji `Index` z kontrolera `Home`.

Możemy pójść dalej i zdefiniować adresy URL niezawierające żadnych zmiennych segmentów, bazując przy identyfikowaniu kontrolera i akcji wyłącznie na wartościach domyślnych. Możemy w ten sposób zdefiniować domyślny URL, korzystając z wartości domyślnych dla obu zmiennych, jak pokazano na listingu 13.10.

Listing 13.10. Określanie domyślnych wartości dla kontrolera i akcji

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

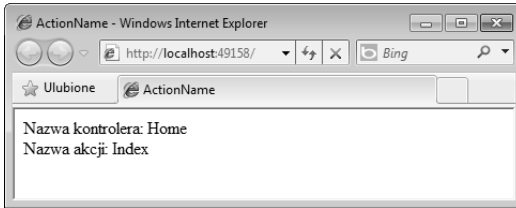
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

Definiując wartości domyślne dla zmiennych `controller` i `action`, utworzyliśmy ścieżkę, która pasuje do adresów URL mających zero, jeden lub dwa segmenty, co jest pokazane w tabeli 13.2.

Tabela 13.2. Dopasowanie adresów URL

Liczba segmentów	Przykład	Odwzorowany na
0	<code>witryna.pl</code>	<code>controller = Home</code> <code>action = Index</code>
1	<code>witryna.pl/Customer</code>	<code>controller = Customer</code> <code>action = Index</code>
2	<code>witryna.pl/Customer/List</code>	<code>controller = Customer</code> <code>action = List</code>
3	<code>witryna.pl/Customer/List/All</code>	Brak dopasowania — za dużo segmentów

Im mniej segmentów otrzymamy w przychodzącym adresie URL, tym bardziej polegamy na wartościach domyślnych aż do otrzymania adresu URL pozbawionego segmentów — w takim przypadku będą użyte jedynie wartości domyślne. Efekt zdefiniowania wartości domyślnych możesz zobaczyć po ponownym uruchomieniu aplikacji. Przeglądarka ponownie zażąda domyślnego adresu URL, ale tym razem nasza nowa ścieżka doda nasze domyślne wartości dla kontrolera i akcji, dzięki czemu przychodzący adres URL zostanie odwzorowany na akcję `Index` w kontrolerze `Home`, jak pokazano na rysunku 13.4.



Rysunek 13.4. Efekt użycia wartości domyślnych w celu rozszerzenia zasięgu ścieżki

Testy jednostkowe — wartości domyślne

Nie musimy wykonywać żadnych specjalnych akcji, jeżeli użyjemy naszych metod pomocniczych do definiowania ścieżek korzystających z wartości domyślnych. Poniżej zamieszczona jest uaktualniona metoda `TestIncomingRoutes` z pliku `RouteTests.cs` dla ścieżek z listingu 13.10:

```
...
[TestMethod]
public void TestIncomingRoutes () {

    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteFail("~/Customer/List/All");
}
...
```

Trzeba tylko pamiętać o podawaniu domyślnego adresu URL jako `~/`, ponieważ ASP.NET w taki sposób prezentuje adresy URL systemowi routingu. Jeżeli podamy pusty ciąg (`"`) lub `/`, system routingu zgłosi wyjątek i test się nie powiedzie.

Użycie statycznych segmentów adresu URL

Nie wszystkie segmenty we wzorcu URL muszą być zmiennymi. Można również tworzyć wzorce mające segmenty statyczne. Załóżmy, że chcemy dopasować poniższy adres URL w celu obsługi adresów URL poprzedzonych słowem *Public*:

```
http://witryna.pl/Public/Home/Index
```

Możemy zrobić to przez użycie wzorca zamieszczonego na listingu 13.11.

Listing 13.11. Wzorec URL z segmentem statycznym

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
```

```

        routes.MapRoute("MyRoute", "{controller}/{action}",
            new { controller = "Home", action = "Index" });

        routes.MapRoute("", "Public/{controller}/{action}",
            new { controller = "Home", action = "Index" });
    }
}

```

Wzorzec ten pasuje wyłącznie do adresów URL posiadających trzy segmenty, z których pierwszym *musi* być *Public*. Pozostałe dwa segmenty mogą zawierać dowolną wartość i będą używane dla zmiennych *controller* oraz *action*. Jeżeli dwa ostatnie segmenty zostaną pominięte, wtedy użyte będą wartości domyślne.

Możemy również tworzyć wzorce URL mające segmenty zawierające zarówno elementy statyczne, jak i zmienne (listing 13.12).

Listing 13.12. Wzorzec URL z segmentem mieszanym

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

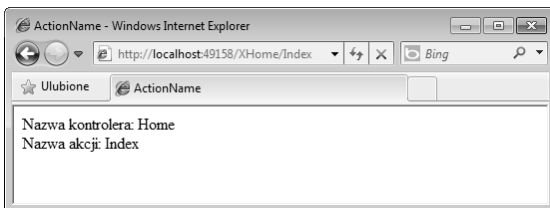
            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}

```

Wzorzec w tej ścieżce pasuje do dowolnego dwusegmentowego adresu URL, w którym pierwszy segment zaczyna się od litery *X*. Wartość zmiennej *controller* jest pobierana z pierwszego segmentu, poza początkową literą *X*. Wartość zmiennej *action* jest pobierana z drugiego segmentu. Efekt działania tego rodzaju ścieżki możesz zobaczyć po uruchomieniu aplikacji i przejściu do adresu URL */XHome/Index*, co zostało pokazane na rysunku 13.5.



Rysunek 13.5. Połączenie statycznych i zmiennych elementów w pojedynczym segmencie

Możemy połączyć statyczne segmenty URL oraz wartości domyślne w celu utworzenia aliasów dla wybranych adresów URL. Jest to przydatne, jeżeli opublikowaliśmy schemat URL w postaci kontraktu dla użytkownika.

Jeżeli zrefaktoryzujesz w takiej sytuacji aplikację, powinieneś zachować poprzedni format adresów URL, aby nadal działały adresy dodane przez użytkownika do ulubionych lub przygotowane przez niego makra. Wyobraźmy sobie, że mieliśmy kontroler o nazwie `Shop`, który został zastąpiony przez kontroler `Home`. Na listingu 13.13 pokazany jest sposób tworzenia ścieżek pozwalających na zachowanie starego schematu URL.

Listing 13.13. Łączenie statycznych segmentów URL oraz wartości domyślnych

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

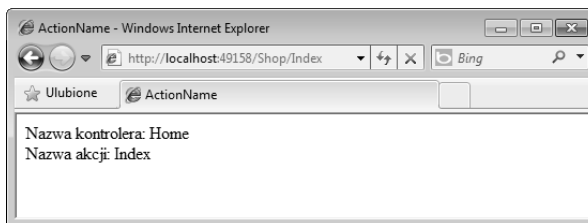
            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

Dodana przez nas ścieżka pasuje do wszystkich dwusegmentowych adresów URL, w których pierwszym segmentem jest `Shop`. Wartość zmiennej `action` jest pobierana z drugiego segmentu. Wzorzec URL nie zawiera zmiennej segmentu o nazwie `controller`, więc użyta jest podana przez nas wartość domyślna. Oznacza to, że żądanie wykonania akcji na kontrolerze `Shop` jest przekształcane w żądanie dla kontrolera `Home`. Efekt działania ścieżki można zobaczyć po uruchomieniu aplikacji i przejściu do adresu URL `/Shop/Index`. Jak pokazano na rysunku 13.6, dodana ścieżka spowodowała, że platforma MVC wywołuje metodę akcji `Index` kontrolera `Home`.



Rysunek 13.6. Utworzenie aliasu w celu zachowania schematu URL

Możemy również pójść o krok dalej i utworzyć aliasy dla metod akcji, które zostały zrefaktoryzowane i nie występują już w kontrolerze. W tym celu należy utworzyć statyczny URL i dostarczyć wartości dla `controller` oraz `action` w postaci wartości domyślnych, jak pokazano na listingu 13.14.

Listing 13.14. Aliasy dla kontrolera i akcji

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("ShopSchema2", "Shop/OldAction",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });

            routes.MapRoute("", "X{controller}/{action}");

            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });

            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}

```

Zwróć uwagę, że kolejny raz umieściliśmy naszą nową ścieżkę jako pierwszą. Jest ona bardziej szczegółowa niż wszystkie kolejne. Jeżeli żądanie otwarcia adresu */Shop/OldAction* byłoby przetworzone przez drugą z kolei ścieżkę, otrzymalibyśmy inny wynik, niż oczekiwaliśmy. Żądanie takie zostałoby obsługane przez zwrócenie informacji o błędzie 404, a nie przez przekształcenie pozwalające zachować istniejący schemat URL.

Kolejność ścieżek

Na listingu 13.12 zdefiniowaliśmy nową ścieżkę i umieściliśmy ją w metodzie `RegisterRoutes` przed wszystkimi innymi. Zrobiliśmy to, ponieważ ścieżki są stosowane w kolejności, w jakiej występują w obiekcie `RouteCollection`. Metoda `MapRoute` dodaje ścieżkę na koniec kolekcji, co oznacza, że ścieżki są zwykle przetwarzane w kolejności dodawania. Użyłem słowa „zwykle”, ponieważ istnieją metody pozwalające na wstawianie ścieżek w wybranym miejscu. Zazwyczaj nie korzystam z tych metod, gdyż uporządkowanie ścieżek w kolejności ich wykonywania pozwala łatwiej zrozumieć routing w aplikacji.

System routingu próbuje dopasować przychodzący adres URL do wzorca URL ścieżki zdefiniowanej jako pierwsza i jeżeli się to nie uda, przechodzi do następnej. Ścieżki są wypróbowywane po kolei, aż do wyczerpania ich zbioru. W konsekwencji musimy definiować *najbardziej szczegółowe ścieżki jako pierwsze*. Ścieżka dodana na listingu 13.12 jest bardziej szczegółowa niż następna. Załóżmy, że odwrócimy kolejność w poniższy sposób:

```

...
routes.MapRoute("MyRoute", "{controller}/{action}",
    new { controller = "Home", action = "Index" });

routes.MapRoute("", "X{controller}/{action}");

...

```

Teraz pierwszą użytą ścieżką będzie ta, która pasuje do *każdego* adresu URL posiadającego zero segmentów, jeden segment lub dwa segmenty. Bardziej szczegółowa ścieżka, znajdująca się na liście jako druga, nie będzie nigdy wykorzystana. Nowa ścieżka powoduje usunięcie początkowego X z adresu URL, co nie jest realizowane we wcześniejszej ścieżce. Z tego powodu poniższy adres URL:

```
http://witryna.pl/XHome/Index
```

zostanie skierowany do nieistniejącego kontrolera o nazwie XHome, wskutek czego nastąpi wygenerowanie użytkownikowi informacji o błędzie 404.

Jeżeli nie przeczytałeś tekstu z ramki na temat testowania jednostkowego przychodzących adresów URL, sugeruję powrót do niego. Jeśli testy jednostkowe mają obejmować tylko jedną część aplikacji MVC, powinien to być schemat adresów URL.

Test jednostkowy — testowanie segmentów statycznych

Kolejny raz użyjemy naszych metod pomocniczych do przetestowania ścieżek, których wzorec URL zawiera segmenty statyczne. Poniżej przedstawiono zmiany wprowadzone w metodzie `TestIncomingRoutes` w celu przetestowania ścieżki dodanej na listingu 13.14:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Shop/Index", "Home", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteFail("~/Customer/List/All");
}
...
```

Definiowanie własnych zmiennych segmentów

Zmienne segmentu `controller` i `action` mają specjalne znaczenie na platformie MVC i — co oczywiste — odpowiadają kontrolerowi i metodzie akcji, które będą użyte do obsługi danego żądania. Nie jesteśmy ograniczeni wyłącznie do zmiennych `controller` i `action`. Możemy również definiować własne zmienne w sposób pokazany na listingu 13.15. (Istniejące trasy z poprzednich sekcji zostały usunięte).

Listing 13.15. Definiowanie nowych zmiennych we wzorcu URL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
```

```

        new { controller = "Home", action = "Index", id = "DefaultId" });
    }
}

```

Wzorzec ścieżki URL definiuje typowe zmienne `controller` oraz `action`, jak również własną zmienną o nazwie `id`. Ścieżka ta pozwala dopasować adresy URL o długości od zera do trzech segmentów. Zawartość trzeciego segmentu jest przypisywana do zmiennej `id`, a jeżeli nie wystąpi trzeci segment, użyta zostanie wartość domyślna.

■ **Ostrzeżenie** Niektóre nazwy są zarezerwowane i nie są dostępne dla nazw zmiennych własnych segmentów. Nazwami tymi są `controller`, `action` oraz `area`. Znaczenie pierwszych dwóch jest oczywiste, a rolę trzeciej wyjaśnię w następnym rozdziale.

W metodzie akcji możemy odczytać każdą ze zmiennych segmentów, korzystając z właściwości `RouteData.Values`. Aby to zademonstrować, trzeba dodać do klasy `HomeController` metodę `CustomVariable` (listing 13.16).

Listing 13.16. Dostęp do własnej zmiennej segmentu w metodzie akcji

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = RouteData.Values["id"];
            return View();
        }
    }
}

```

Metoda ta pozyskuje wartość zmiennej z wzorca ścieżki i przekazuje ją do widoku poprzez `ViewBag`. Kliknij prawym przyciskiem myszy nową metodę akcji w edytorze kodu i wybierz opcję *Dodaj widok...* z menu kontekstowego. Widokowi nadaj nazwę `CustomVariable` i kliknij przycisk *Dodaj*. Visual Studio utworzy nowy plik widoku `CustomVariable.cshtml` w katalogu `/Views/Home`. Kod widoku zmodyfikuj tak, aby odpowiadał przedstawionemu na listingu 13.17.

Listing 13.17. Wyświetlanie wartości zmiennej własnego segmentu

```

@{
    Layout = null;
}

```



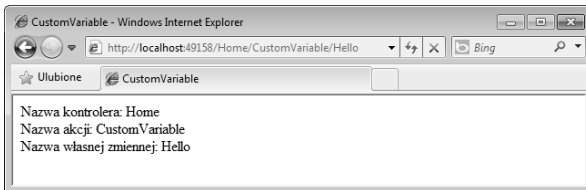
```

<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>CustomVariable</title>
</head>
<body>
  <div>Nazwa kontrolera: @ViewBag.Controller</div>
  <div>Nazwa akcji: @ViewBag.Action</div>
  <div>Nazwa własnej zmiennej: @ViewBag.CustomVariable</div>
</body>
</html>

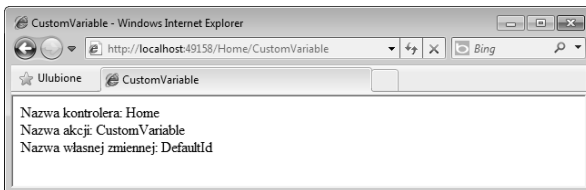
```

Aby zobaczyć efekt zdefiniowania zmiennej własnego segmentu, uruchom aplikację i przejdź do adresu URL `/Home/CustomVariable/Halo`. Zostanie wywołana akcja `CustomVariable` z kontrolera `Home`, a wartość zmiennej naszego segmentu będzie pobrana z `ViewBag` i wyświetlona na stronie, jak pokazano na rysunku 13.7.



Rysunek 13.7. Wyświetlanie wartości zmiennej własnego segmentu

Zmiennej segmentu przypisaliśmy wartość domyślną, co oznacza, że po przejściu do adresu URL `/Home/CustomVariable` otrzymasz wynik pokazany na rysunku 13.8.



Rysunek 13.8. Wyświetlanie wartości domyślnej zmiennej własnego segmentu

Test jednostkowy — testowanie zmiennych własnych segmentów

W naszych metodach pomocniczych testów dodaliśmy obsługę testowania zmiennych własnych segmentów. Metoda `TestRouteMatch` posiada opcjonalny parametr, który akceptuje typ anonimowy zawierający nazwy zmiennych, jakie chcemy testować, oraz oczekiwane wartości. Poniżej przedstawiono zmodyfikowaną wersję metody `TestIncomingRoutes` w celu przetestowania ścieżki zdefiniowanej na listingu 13.15.

...

```

[TestMethod]
public void TestIncomingRoutes() {

    TestRouteMatch("~/", "Home", "Index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer", "Customer", "index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer/List", "Customer", "List", new { id = "DefaultId" });
}

```

```

    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}
...

```

Użycie własnych zmiennych jako parametrów metod akcji

Użycie właściwości `RouteData.Values` jest jedynie pierwszym ze sposobów na dostęp do zmiennych własnych segmentów. Inny sposób jest znacznie elegantszy. Jeżeli zdefiniujemy parametr metody akcji o nazwie pasującej do zmiennej w wzorca URL, platforma MVC przekaże wartość pobraną z URL do tego parametru metody akcji. Na przykład własna zmienna zdefiniowana w ścieżce z listingu 13.15 ma nazwę `id`. Możemy zmodyfikować metodę akcji `CustomVariable` w taki sposób, aby posiadała analogiczny parametr, jak pokazano na listingu 13.18.

Listing 13.18. Mapowanie zmiennej własnego segmentu URL na parametr metody akcji

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }
    }
}

```

Gdy system routingu dopasuje URL do ścieżki zdefiniowanej na listingu 13.15, wartość trzeciego segmentu w adresie URL zostanie przypisana do zmiennej `id`. Platforma MVC porówna listę zmiennych segmentów z listą parametrów metody akcji i jeżeli zostaną znalezione pasujące nazwy, wartości z adresu URL będą przekazane do metody.

Parametr `id` zdefiniowaliśmy jako `string`, ale platforma MVC będzie próbowała skonwertować wartość z URL na dowolny zdefiniowany przez nas typ. Jeżeli zadeklarujemy parametr `id` jako `int` lub `DateTime`, otrzymamy wartość z URL w postaci obiektu właściwego typu. Jest to elegancka i przydatna funkcja, która pozwala uniknąć samodzielnej realizacji konwersji.

-
- **Uwaga** Przy konwersji wartości znajdujących się w adresie URL na typy .NET platforma MVC korzysta z systemu dołączania obiektu, który jest w stanie obsłużyć sytuacje znacznie bardziej skomplikowane niż pokazane w tym przykładzie. Dołączanie modelu przedstawię w rozdziale 22.
-

Definiowanie opcjonalnych segmentów URL

Opcjonalny segment URL to taki, który nie musi być specyfikowany przez użytkownika, ale dla którego nie są podane wartości domyślne. Na listingu 13.19 pokazany jest przykład. Opcjonalność zmiennej segmentu zaznaczyliśmy przez ustawienie domyślnej wartości parametru `UrlParameter.Optional`, co zostało oznaczone czcionką pogrubioną.

Listing 13.19. Określanie opcjonalnego segmentu URL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional });
        }
    }
}
```

Ścieżka ta będzie dopasowana do adresów URL niezależnie od tego, czy zostanie podany segment `id`. W tabeli 13.3 pokazane jest działanie tego mechanizmu dla różnych adresów URL.

Tabela 13.3. Dopasowanie adresów URL z opcjonalną zmienną segmentu

Liczba segmentów	Przykładowy URL	Odwzorowany na
0	<i>witryna.pl</i>	controller = Home action = Index
1	<i>witryna.pl/Customer</i>	controller = Customer action = Index
2	<i>witryna.pl/Customer/List</i>	controller = Customer action = List
3	<i>witryna.pl/Customer/List/All</i>	controller = Customer action = List id = All
4	<i>witryna.pl/Customer/List/All/Delete</i>	Brak dopasowania — za dużo segmentów

Jak można zauważyć w tabeli, zmienna `id` jest dodawana do zbioru zmiennych tylko w przypadku, gdy odpowiedni segment znajduje się w przychodzącym adresie URL. Funkcja ta jest przydatna, gdy musimy wiedzieć, czy użytkownik podał wartość. Jeżeli dla opcjonalnej zmiennej segmentu nie zostanie podana żadna wartość, wówczas wartością odpowiadającego jej parametru będzie `null`. Na listingu 13.20 pokazano uaktualnioną wersję kontrolera, który odpowiada, gdy nie zostanie podana żadna wartość dla zmiennej `id` segmentu.

Listing 13.20. Sprawdzenie, czy opcjonalnej zmiennej segmentu została przypisana wartość

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

```

using System.Web;
using System.Web.Mvc;

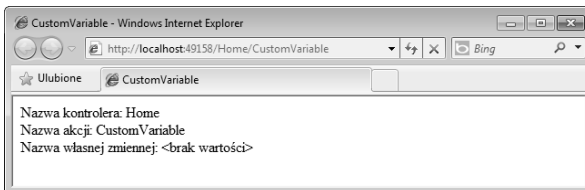
namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }

        public ActionResult CustomVariable(string id) {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id == null ? "<brak wartości>" : id;
            return View();
        }
    }
}

```

Po uruchomieniu aplikacji i przejściu do adresu URL `/Home/CustomVariable` (który nie posiada zdefiniowanej wartości domyślnej dla zmiennej `id` segmentu) otrzymasz wynik pokazany na rysunku 13.9.



Rysunek 13.9. Aplikacja wykryła, że adres URL nie zawiera wartości dla opcjonalnej zmiennej segmentu

Użycie opcjonalnych segmentów URL w celu wymuszenia separacji zadań

Niektórzy programiści bardzo mocno koncentrują się na separacji zadań na platformie MVC i nie lubią umieszczania wartości domyślnych zmiennych segmentu w ścieżkach aplikacji. Jeżeli jest to problemem również dla Ciebie, możesz użyć funkcji parametrów opcjonalnych w C# wraz z opcjonalną zmienną segmentu w trasie w celu zdefiniowania wartości domyślnych dla parametrów metod akcji. Jak pokazano na listingu 13.21, należy zmodyfikować metodę akcji `CustomVariable` i zdefiniować wartość domyślną dla parametru `id`, która będzie używana, jeśli adres URL nie będzie zawierał wartości dla wspomnianego parametru.

Listing 13.21. Definiowanie wartości domyślnej dla parametru metody akcji

```

...
public ViewResult CustomVariable(string id = "DefaultId") {
    ViewBag.Controller = "Home";
    ViewBag.Action = "CustomVariable";
    ViewBag.CustomVariable = id;
    return View();
}
...

```

W ten sposób zapewnimy wartość dla parametru `id` (albo pochodzącą z adresu URL, albo domyślną), więc możemy usunąć kod odpowiedzialny za obsługę wartości `null`. Ta metoda akcji, w połączeniu ze ścieżką zdefiniowaną na listingu 13.19, ma taką samą funkcjonalność jak ścieżka zdefiniowana na listingu 13.15:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "DefaultId" });
...
```

Różnica polega na tym, że wartość domyślna dla zmiennej `id` segmentu jest zdefiniowana w kodzie kontrolera, a nie w definicji ścieżki.

Definiowanie ścieżek o zmiennej długości

Innym sposobem na zmianę domyślnego konserwatywnego zachowania ścieżek URL jest akceptowanie zmiennej liczby segmentów URL. Pozwala to na obsługiwanie adresów URL o dowolnej długości przy zastosowaniu jednej definicji ścieżki. Obsługę zmiennej liczby segmentów realizuje się przez wyznaczenie jednej ze zmiennych segmentów jako zmiennej przechwytywającej, co jest realizowane przez poprzedzenie jej znakiem gwiazdki (*), jak pokazano na listingu 13.22.

Listing 13.22. Wyznaczanie zmiennej przechwytywającej

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional });
        }
    }
}
```

Testy jednostkowe — opcjonalne segmenty URL

Jedynym problemem, na który musimy zwrócić uwagę przy testowaniu opcjonalnych segmentów URL jest to, czy zmienna segmentu nie zostanie dodana do kolekcji `RouteData.Values`, gdy wartość nie zostanie znaleziona w adresie URL. Oznacza to, że nie powinniśmy dołączać zmiennej w typie anonimowym, o ile nie testujemy adresu URL zawierającego opcjonalny segment. Poniżej przedstawiono zmiany, jakie trzeba wprowadzić w metodzie `TestIncomingRoutes`, aby przetestować ścieżkę zdefiniowaną na listingu 13.19.

```
...

[TestMethod]
public void TestIncomingRoutes() {

    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}

...
```

Rozszerzyliśmy ścieżkę z poprzedniego przykładu o dodatkową zmienną segmentu przechwytyjącego, o nazwie `catchall`. Ścieżka ta pasuje teraz do *dowolnego* adresu URL, niezależnie od liczby segmentów lub wartości któregośkolwiek z nich. Pierwsze trzy segmenty są wykorzystywane do skonfigurowania wartości zmiennych `controller`, `action` oraz `id`. Jeżeli adres URL będzie zawierał kolejne segmenty, zostaną one przypisane do zmiennej `catchall` w sposób pokazany w tabeli 13.4.

Tabela 13.4. Dopasowanie adresów URL ze zmienną przechwytyjącą segmentu

Liczba segmentów	Przykładowy URL	Odwzorowany na
0	<i>witryna.pl</i>	<code>controller = Home</code> <code>action = Index</code>
1	<i>witryna.pl/Customer</i>	<code>controller = Customer</code> <code>action = Index</code>
2	<i>witryna.pl/Customer/List</i>	<code>controller = Customer</code> <code>action = List</code>
3	<i>witryna.pl/Customer/List/All</i>	<code>controller = Customer</code> <code>action = List</code> <code>id = All</code>
4	<i>witryna.pl/Customer/List/All/Delete</i>	<code>controller = Customer</code> <code>action = List</code> <code>id = All</code> <code>catchall = Delete</code>
5	<i>witryna.pl/Customer/List/All/Delete/Perm</i>	<code>controller = Customer</code> <code>action = List</code> <code>id = All</code> <code>catchall = Delete/Perm</code>

Nie istnieje górna granica liczby segmentów możliwych do dopasowania przez wzorec URL. Zwróć uwagę, że segmenty w zmiennej przechwytyjącej są prezentowane w postaci *segment/segment/segment*. To my jesteśmy odpowiedzialni za przetworzenie tego ciągu znaków i jego podział na pojedyncze segmenty.

Test jednostkowy — testowanie zmiennych segmentów przechwytyjących

Zmienną przechwytyjącą możemy potraktować jak każdą inną zmienną. Jedyna różnica polega na tym, że musimy oczekiwać otrzymania wartości wielu segmentów połączonych w jedną wartość, na przykład *segment/segment/segment*. Zwróć uwagę, że nie otrzymamy początkowego ani końcowego znaku `/`. Poniżej przedstawiono zmiany, jakie trzeba wprowadzić w metodzie `TestIncomingRoutes`, aby przetestować ścieżkę zdefiniowaną na listingu 13.22 oraz adresy URL z tabeli 13.4:

...

`[TestMethod]`

```
public void TestIncomingRoutes() {

    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteMatch("~/Customer/List/All/Delete", "Customer", "List",
        new { id = "All", catchall = "Delete" });
    TestRouteMatch("~/Customer/List/All/Delete/Perm", "Customer", "List",
        new { id = "All", catchall = "Delete/Perm" });
}
...
```

Definiowanie priorytetów kontrolerów na podstawie przestrzeni nazw

Gdy przychodzący adres URL zostanie dopasowany do ścieżki, platforma MVC odczytuje nazwę zmiennej `controller` i szuka klasy o odpowiedniej nazwie. Jeżeli na przykład wartością zmiennej `controller` jest `Home`, platforma MVC poszukuje klasy o nazwie `HomeController`. Jest to *niekwalifikowana* nazwa klasy, co oznacza, że w przypadku znalezienia co najmniej dwóch klas o nazwie `HomeController` w różnych przestrzeniach nazw platforma nie będzie „wiedziała”, którą z nich należy wybrać.

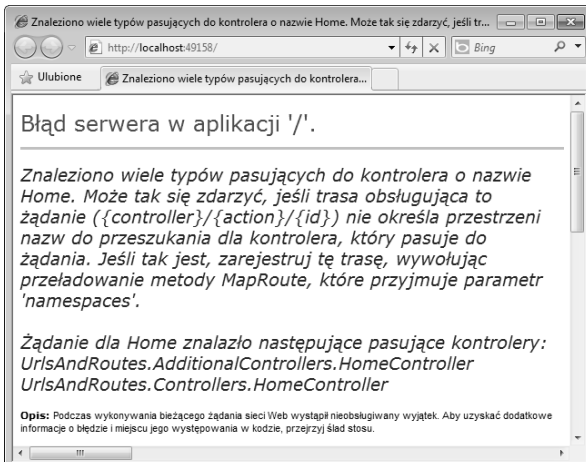
Aby zademonstrować ten problem, utwórz nowy podkatalog w katalogu głównym projektu i nadaj mu nazwę `AdditionalControllers`. Następnie umieść w nim nowy kontroler `HomeController`, którego kod przedstawiono na listingu 13.23.

Listing 13.23. Dodanie drugiego kontrolera `HomeController` do omawianego projektu

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

namespace UrlsAndRoutes.AdditionalControllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Additional Controllers - Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Po uruchomieniu aplikacji zobaczysz błąd pokazany na rysunku 13.10.



Rysunek 13.10. Błąd występujący w przypadku niejednoznaczności klas kontrolerów

Platforma MVC szukała klasy `HomeController` i znalazła dwie: pierwszą w katalogu `AdditionalControllers` i drugą w katalogu `Controllers`. Jeżeli wczytasz się w tekst komunikatu pokazanego na rysunku 13.10, to dowiesz się, które klasy zostały znalezione przez platformę MVC.

Problem ten pojawia się częściej, niż można się tego spodziewać, szczególnie jeżeli pracujemy nad dużym projektem MVC, który korzysta z bibliotek kontrolerów pochodzących od różnych zespołów lub zewnętrznych dostawców. Naturalne jest nazywanie kontrolera związanego z kontami użytkowników ciągiem `AccountController`, a jest to tylko jeden z przypadków, gdy napotkamy konflikty nazw.

Aby rozwiązać ten problem, możemy określić przestrzenie nazw, które powinny mieć wyższy priorytet przy wyborze nazwy klasy kontrolera (listing 13.24).

Listing 13.24. Określanie kolejności wykorzystania przestrzeni nazw

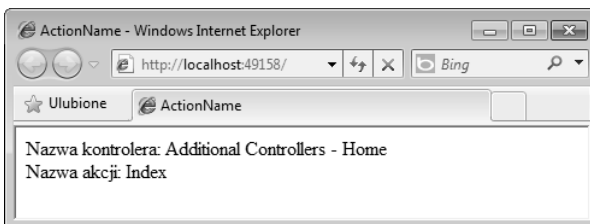
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "UrlsAndRoutes.AdditionalControllers" });
        }
    }
}
```

Przestrzenie nazw zapisujemy jako tablicę ciągów znakowych. W kodzie zamieszczonym na powyższym listingu informujemy platformę MVC, aby przeszukiwała przestrzeń nazw `UrlsAndRoutes.AdditionalControllers` jako pierwszą.

Jeżeli w podanej przestrzeni nazw nie zostanie znaleziony odpowiedni kontroler, platforma MVC wróci do standardowego działania i przeszuka wszystkie dostępne przestrzenie nazw. Po uruchomieniu aplikacji na tym etapie otrzymasz wynik pokazany na rysunku 13.11. Na wymienionym rysunku pokazano, że żądanie skierowane do adresu głównego aplikacji, które jest przetwarzane przez metodę akcji `Index` kontrolera `Home`, zostało przekazane kontrolerowi zdefiniowanemu w przestrzeni nazw `AdditionalControllers`.



Rysunek 13.11. Nadanie priorytetu kontrolerom we wskazanej przestrzeni nazw

Przestrzenie nazw dodawane do ścieżki mają identyczny priorytet. Platforma MVC nie sprawdza pierwszej przestrzeni nazw, potem przechodzi do następnej itd. Dodajmy do ścieżki na przykład obie nasze przestrzenie nazw w poniższy sposób:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
    new { controller = "Home", action = "Index",
```



```

        id = UrlParameter.Optional
    },
    new[] { "UrlsAndRoutes.AdditionalControllers", "UrlsAndRoutes.Controllers" });
...

```

Ponownie zobaczymy informacje o błędzie pokazane na rysunku 13.10, ponieważ platforma MVC próbuje znaleźć klasę kontrolera we *wszystkich* przestrzeniach nazw dodanych do ścieżki. Jeżeli chcemy zwiększyć priorytet kontrolera z jednej przestrzeni nazw, a wszystkie inne kontrolery wyszukiwać w innej przestrzeni, musimy utworzyć wiele ścieżek, jak pokazano na listingu 13.25.

Listing 13.25. Użycie wielu ścieżek do sterowania przeszukiwaniem przestrzeni nazw

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            routes.MapRoute("AddControllerRoute", "Home/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "UrlsAndRoutes.AdditionalControllers" });

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}

```

Pierwsza ścieżka zostanie użyta, kiedy użytkownik wyraźnie wskaże adres URL, którego pierwszym segmentem jest Home. W takim przypadku żądanie będzie skierowane do kontrolera HomeController w katalogu *AdditionalControllers*. Wszystkie pozostałe żądania, łącznie z tymi, w których nie zdefiniowano pierwszego segmentu, zostaną obsłużone przez kontrolery znajdujące się w katalogu *Controllers*.

Możemy również zmusić platformę MVC, aby szukała *wyłącznie* w podanych przez nas przestrzeniach nazw. Jeżeli nie zostanie znaleziony odpowiedni kontroler, biblioteka nie będzie szukała go w innych przestrzeniach. Na listingu 13.26 przedstawiony jest sposób użycia tej funkcji.

Listing 13.26. Wylączenie domyślnego przeszukiwania przestrzeni nazw

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

            Route myRoute = routes.MapRoute("AddControllerRoute",
                "Home/{action}/{id}/{*catchall}",

```

```

        new { controller = "Home", action = "Index",
              id = UrlParameter.Optional },
        new[] { "UrlsAndRoutes.AdditionalControllers" });

    myRoute.DataTokens["UseNamespaceFallback"] = false;
}
}
}

```

Metoda `MapRoute` zwraca obiekt `Route`. Do tej pory nie korzystaliśmy z tej metody, ponieważ nie potrzebowaliśmy wprowadzać żadnych korekt do tworzonych ścieżek. Aby wyłączyć przeszukiwanie kontrolerów w innych przestrzeniach nazw, musimy pobrać obiekt `Route` i przypisać kluczowi `UseNamespaceFallback` w kolekcji `DataTokens` wartość `false`.

Ustawienie to zostanie przekazane do komponentu odpowiedzialnego za wyszukiwanie kontrolerów, nazywanego *fabryką kontrolerów*, który przedstawię szczegółowo w rozdziale 17. Efektem wprowadzonej zmiany jest to, że żądania, które nie mogą być obsłużone przez kontroler `Home` z katalogu `AdditionalControllers`, zakończą się niepowodzeniem.

Ograniczenia ścieżek

Na początku rozdziału napisałem, że wzorce URL są konserwatywne przy dopasowywaniu segmentów i liberalne przy dopasowywaniu zawartości tych segmentów. W kilku poprzednich punktach przedstawiłem różne techniki kontrolowania poziomu konserwatyzmu — tworzenia ścieżek pasujących do większej lub mniejszej liczby segmentów przez użycie wartości domyślnych, zmiennych opcjonalnych itd.

Teraz czas zająć się sposobami kontrolowania liberalizmu przy dostosowywaniu zawartości segmentów URL — możliwościami ograniczenia zbioru adresów URL — do której będzie pasowała ścieżka. Ponieważ mamy kontrolę nad oboma tymi aspektami ścieżki, możemy tworzyć schematy URL, które działają z laserową precyzją.

Ograniczanie ścieżki z użyciem wyrażeń regularnych

Pierwszą techniką, jaką się zajmiemy, jest ograniczanie ścieżek z użyciem wyrażeń regularnych. Na listingu 13.27 pokazany jest przykład.

Listing 13.27. Użycie wyrażeń regularnych do ograniczania ścieżki

```

...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*"},
        new[] { "UrlsAndRoutes.Controllers"});
}
...

```

Ograniczenia definiujemy przez przekazanie ich jako parametru do metody `MapRoute`. Podobnie jak w przypadku wartości domyślnych, ograniczenia są zapisywane w postaci typu anonimowego, którego właściwości odpowiadają nazwom zmiennych segmentów, które chcemy ograniczyć.

W zamieszczonym przykładzie użyliśmy stałej z wyrażeniem regularnym pasującym do adresu URL tylko wtedy, gdy wartość zmiennej kontrolera zaczyna się od litery `H`.

- **Uwaga** Wartości domyślne są używane przed sprawdzeniem ograniczeń. Jeżeli zatem otworzymy URL `/`, zostanie zastosowana domyślna wartość dla zmiennej `controller`, czyli w tym przypadku `Home`. Następnie są sprawdzane ograniczenia, a ponieważ wartość zmiennej `controller` zaczyna się od `H`, domyślny URL będzie pasował do tej ścieżki.

Ograniczanie ścieżki do zbioru wartości

Wyrażenia regularne możemy wykorzystać do definiowania ścieżki pasującej wyłącznie do specyficznych wartości segmentu. W tym celu zastosujemy znak `|`, jak pokazano na listingu 13.28.

Listing 13.28. Ograniczanie ścieżki do zbioru wartości zmiennej segmentu

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*", action = "^Index$|^About$"},
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

Ograniczenie to powoduje, że ścieżka pasuje wyłącznie do adresów URL, których wartość segmentu `action` jest `Index` lub `About`. Ograniczenia są stosowane jednocześnie, więc ograniczenia nałożone na wartości zmiennej `action` są łączone z tymi, które są nałożone na zmienną `controller`. Oznacza to, że ścieżka z listingu 13.28 będzie pasowała wyłącznie do adresów URL, których zmienna `controller` zaczyna się od litery `H`, a zmienna `action` ma wartość `Index` lub `About`. Teraz wiesz już, co miałem na myśli, pisząc o bardzo precyzyjnych ścieżkach.

Ograniczanie ścieżek z użyciem metod HTTP

Możliwe jest ograniczanie ścieżek w taki sposób, aby dopasowywały wyłącznie adresy URL w momencie, gdy żądanie korzysta z wybranej metody HTTP, jak pokazano na listingu 13.29.

Listing 13.29. Ograniczanie ścieżek bazujące na metodzie HTTP

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*", action = "Index|About",
            HttpMethod = new HttpMethodConstraint("GET") },
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

Format definiowania metody HTTP jest trochę dziwny. Nazwa nadana właściwości nie ma znaczenia — wystarczy, że będzie egzemplarzem klasy `HttpMethodConstraint`. W powyższym listingu nazwaliśmy ją `httpMethod`, aby pomóc odróżnić ją od wcześniej zdefiniowanych ograniczeń wartościowych.

- **Uwaga** Możliwość ograniczania ścieżek za pomocą metod HTTP nie jest związana z możliwością ograniczania metod akcji za pomocą takich atrybutów jak `HttpGet` czy `HttpPost`. Ograniczenia ścieżek są przetwarzane znacznie wcześniej w potoku obsługi żądania i wyznaczają nazwę kontrolera i akcji wymaganej do przetworzenia żądania. Atrybuty metod akcji są używane do wybrania wersji metody akcji stosowanej do obsługi żądania przez kontroler. Więcej informacji na temat obsługi różnych rodzajów metod HTTP (w tym również rzadziej stosowanych, takich jak PUT i DELETE) przedstawiamy w rozdziale 14.

Do konstruktora klasy `HttpMethodConstraint` przekazujemy nazwy metod HTTP, które chcemy obsługiwać. Na wcześniejszym listingu ograniczyliśmy ścieżkę wyłącznie do żądań GET, ale możemy łatwo dodać obsługę innych metod:

```
...
httpMethod = new HttpMethodConstraint("GET", "POST" ),
...
```

Testy jednostkowe — ograniczenia ścieżek

Przy testowaniu ograniczeń ścieżek ważne jest, aby sprawdzić zarówno niepasujące adresy URL, jak i adresy, które próbujemy wykluczyć, co możemy zrealizować przez wykorzystanie metod pomocniczych wprowadzonych na początku tego rozdziału. Poniżej przedstawiono zmodyfikowaną metodę testową `TestIncomingRoutes`, której użyjemy do przetestowania ścieżek zdefiniowanych na listingu 13.29:

```
...
[TestMethod]
public void TestIncomingRoutes () {

    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Home", "Home", "Index");
    TestRouteMatch("~/Home/Index", "Home", "Index");

    TestRouteMatch("~/Home/About", "Home", "About");
    TestRouteMatch("~/Home/About/MyId", "Home", "About", new { id = "MyId" });
    TestRouteMatch("~/Home/About/MyId/More/Segments", "Home", "About",
        new {
            id = "MyId",
            catchall = "More/Segments"
        });

    TestRouteFail("~/Home/OtherAction");
    TestRouteFail("~/Account/Index");
    TestRouteFail("~/Account/About");
}
...
```

Definiowanie własnych ograniczeń

Jeżeli standardowe ograniczenia nie są wystarczające do naszych potrzeb, możemy zdefiniować własne ograniczenia przez zaimplementowanie interfejsu `IRouteConstraint`. Aby zademonstrować tę funkcję, do projektu dodajemy katalog *Infrastructure*, w którym następnie tworzymy nowy plik klasy o nazwie *UserAgentConstraint.cs* i umieszczamy w niej kod przedstawiony na listingu 13.30.

Listing 13.30. Tworzenie własnego ograniczenia ścieżki

```

using System.Web;
using System.Web.Routing;

namespace UrlsAndRoutes.Infrastructure {

    public class UserAgentConstraint : IRouteConstraint {
        private string requiredUserAgent;

        public UserAgentConstraint(string agentParam) {
            requiredUserAgent = agentParam;
        }

        public bool Match(HttpContextBase httpContext, Route route,
            string parameterName, RouteValueDictionary values,
            RouteDirection routeDirection) {

            return httpContext.Request.UserAgent != null &&
                httpContext.Request.UserAgent.Contains(requiredUserAgent);
        }
    }
}

```

Interfejs `IRouteConstraint` definiuje metodę `Match`, której implementacja wskazuje systemowi routingu, czy ograniczenie jest spełnione. Parametry metody `Match` zapewniają dostęp do żądania od klienta, do kontrolowanej ścieżki, do zmiennych segmentów pobranych z adresu URL oraz do informacji, czy żądanie dotyczy przychodzącego, czy wychodzącego adresu URL. W naszym przykładzie sprawdzamy, czy wartość właściwości `UserAgent` w żądaniu klienta jest taka sama jak wartość przekazana do konstruktora. Na listingu 13.31 pokazane jest nasze ograniczenie użyte w ścieżce.

Listing 13.31. Użycie niestandardowego ograniczenia w ścieżce

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;

namespace UrlsAndRoutes {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {

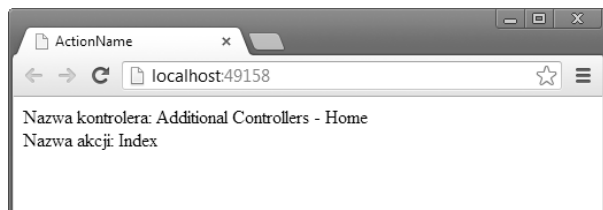
            routes.MapRoute("ChromeRoute", "{*catchall}",
                new { controller = "Home", action = "Index" },
                new {
                    customConstraint = new UserAgentConstraint("Chrome")
                },
                new[] { "UrlsAndRoutes.AdditionalControllers" });

            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional },
                new[] { "UrlsAndRoutes.Controllers" });
        }
    }
}

```

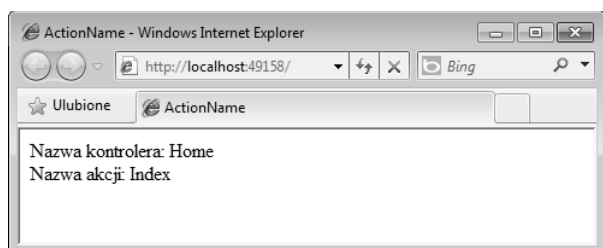
Na listingu mamy zdefiniowane ograniczenie ścieżki pozwalające na dopasowanie wyłącznie żądań wykonywanych z przeglądarek, których nagłówek `user-agent` zawiera Chrome. Jeżeli ścieżka zostanie dopasowana, wówczas żądanie będzie skierowane do metody akcji `Index` w kontrolerze `HomeController` zdefiniowanym w katalogu `AdditionalControllers` bez względu na strukturę i treść żadanego adresu URL. Nasz wzorec URL składa się ze zmiennej `catchall` segmentu, co oznacza, że wartości zmiennych `controller` i `action` segmentu zawsze będą wartościami domyślnymi, a nie pobranymi z adresu URL.

Druga ścieżka spowoduje dopasowanie wszystkich żądań i kontrolerów docelowych w katalogu `Controllers`. Efektem zastosowania omówionych ścieżek jest to, że jedna z przeglądarek zawsze będzie przechodziła do tego samego miejsca w aplikacji, co możesz zobaczyć na rysunku 13.12. Na wymienionym rysunku pokazano efekt uruchomienia aplikacji w przeglądarce Google Chrome.



Rysunek 13.12. Aplikacja uruchomiona w przeglądarce Google Chrome

Z kolei na rysunku 13.13 pokazano wynik uruchomienia tej samej aplikacji w przeglądarce Internet Explorer.



Rysunek 13.13. Aplikacja uruchomiona w przeglądarce Internet Explorer

-
- **Uwaga** Chcę postawić sprawę jasno — nie sugeruję, abyś ograniczał aplikację do obsługi przeglądarki tylko jednego typu. Użyłem nagłówka `user-agent` wyłącznie w celu zademonstrowania własnych ograniczeń ścieżki, ponieważ wierzę w równe szanse wszystkich przeglądarek. Naprawdę nie noszę witryn, które wymuszają na użytkownikach wybór przeglądarki.
-

Routing żądań dla plików dyskowych

Nie wszystkie żądania do aplikacji MVC odnoszą się do kontrolerów i akcji. Nadal musimy udostępniać takie dane, jak zdjęcia, statyczne pliki HTML, biblioteki JavaScript itd. Na przykład w naszej aplikacji MVC utworzymy w katalogu `Content` plik o nazwie `StaticContent.html`. Zawartość tego pliku znajduje się na listingu 13.32.

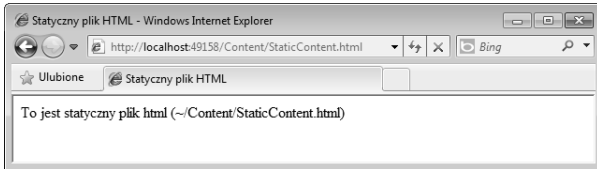
Listing 13.32. Plik *StaticContent.html*

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Statyczny plik HTML</title></head>
  <body>
    To jest statyczny plik html (~Content/StaticContent.html)
  </body>
</html>

```

System routingu zawiera zintegrowaną obsługę tego typu treści. Jeżeli uruchomisz aplikację i użyjesz adresu URL */Content/StaticContent.html*, zobaczysz w przeglądarce zawartość tego prostego pliku HTML (rysunek 13.14).



Rysunek 13.14. Żądanie przesłania pliku statycznego

Domyślnie system routingu sprawdza, czy adres URL pasuje do pliku na dysku, *zanim* zacznie przetwarzać ścieżki aplikacji. Dlatego też nie trzeba definiować ścieżki, aby otrzymać efekt pokazany na rysunku 13.14.

Jeżeli zostanie znalezione dopasowanie, plik z dysku jest udostępniany przeglądarce, a ścieżki nie są przetwarzane. Możemy odwrócić ten sposób działania, aby nasze ścieżki były przetwarzane przed sprawdzaniem plików na dysku — zmienić właściwość *RouteExistingFiles* w obiekcie *RouteCollection* na *true* (listing 13.33).

Listing 13.33. Aktywowanie przetwarzania ścieżek przed kontrolą plików

```

public static void RegisterRoutes(RouteCollection routes) {

    routes.RouteExistingFiles = true;

    routes.MapRoute("ChromeRoute", "{*catchall}",
        new { controller = "Home", action = "Index" },
        new {
            customConstraint = new UserAgentConstraint("Chrome")
        },
        new[] { "UrlsAndRoutes.AdditionalControllers" });

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index",
            id = UrlParameter.Optional },
        new[] { "UrlsAndRoutes.Controllers" });
}

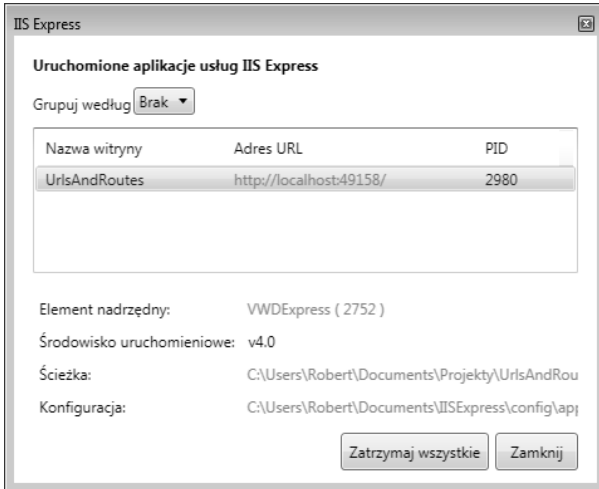
```

Zgodnie z konwencją instrukcja ta powinna znajdować się blisko początku metody *RegisterRoutes*, choć będzie działała nawet wtedy, gdy zostanie ustawiona po zdefiniowaniu ścieżek.

Konfiguracja serwera aplikacji

Visual Studio 2012 używa IIS Express jako serwera aplikacji dla projektów MVC. Nie tylko powinniśmy ustawić wartość *true* właściwości *RouteExistingFiles* w metodzie *RegisterRoutes*, ale również poinformować serwer IIS Express, aby nie przechwytywał żądań do plików na dysku, zanim nie zostaną one przekazane systemowi routingu MVC.

Przede wszystkim uruchom IIS Express. Najłatwiejszym sposobem jest uruchomienie aplikacji MVC w Visual Studio, co spowoduje wyświetlenie ikony IIS Express na pasku zadań. Kliknij tę ikonę prawym przyciskiem myszy i z menu kontekstowego wybierz opcję *Pokaż wszystkie aplikacje*. Kliknij *UrlsAndRoutes* w kolumnie *Nazwa witryny*, aby w ten sposób wyświetlić informacje konfiguracyjne, jak pokazano na rysunku 13.15.

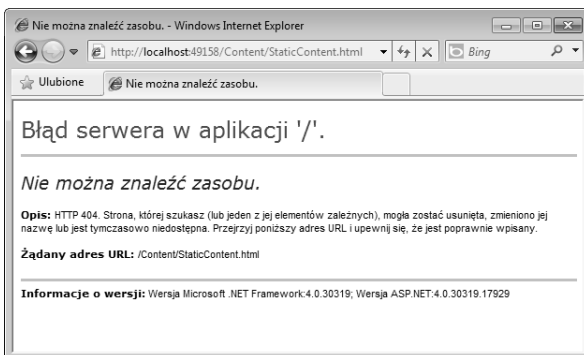


Rysunek 13.15. Informacje konfiguracyjne serwera IIS Express

Kliknij łącze *Konfiguracja* znajdujące się na dole okna, co spowoduje wyświetlenie w Visual Studio pliku konfiguracyjnego IIS Express. Teraz naciśnij klawisze *Ctrl+F* i wyszukaj `UrlRoutingModule-4.0`. Odpowiedni wpis znajduje się w sekcji `modules` pliku konfiguracyjnego. Naszym celem jest ustawienie atrybutowi `preCondition` pustego ciągu tekstowego, np.:

```
<add name="UrlRoutingModule-4.0" type="System.Web.Routing.UrlRoutingModule"
preCondition="" />
```

Teraz ponownie uruchom aplikację w Visual Studio, aby zmodyfikowane ustawienia zostały uwzględnione, a następnie przejdź do strony pod adresem `/Content/StaticContent.html`. Zamiast zobaczyć zawartość pliku, w oknie przeglądarki internetowej zostanie wyświetlony komunikat błędu widoczny na rysunku 13.16. Błąd wynika z tego, że żądanie pliku HTML zostało przekazane do systemu routingu MVC, ale ścieżka dopasowująca adres URL przekierowuje żądanie do nieistniejącego kontrolera `Content`.



Rysunek 13.16. Żądanie pliku statycznego obsługowane przez system routingu

- **Wskazówka** Innym rozwiązaniem jest użycie serwera Visual Studio, który możesz aktywować w sekcji *Sieć Web* konfiguracji projektu wyświetlanej po wybraniu opcji *Właściwości UrlsAndRoutes...* w menu *Projekt* w Visual Studio. Serwer Visual Studio jest całkiem prostą, a nie okrojoną wersją IIS tak jak IIS Express, więc nie przechwytyje żądań w ten sam sposób.

Definiowanie ścieżek dla plików na dysku

Gdy właściwość `RouteExistingFiles` została przypisana wartość `true`, możemy zdefiniować ścieżki odpowiadające adresom URL dla plików na dysku, takie jak na listingu 13.34.

Listing 13.34. Ścieżka, której wzorec URL odpowiada plikowi na dysku

```
...
public static void RegisterRoutes(RouteCollection routes) {

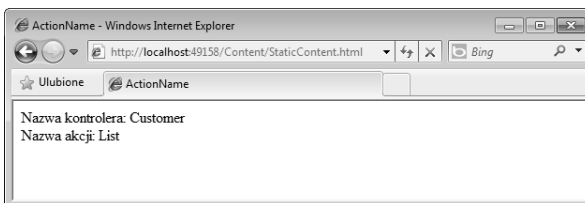
    routes.RouteExistingFiles = true;

    routes.MapRoute("DiskFile", "Content/StaticContent.html",
        new {
            controller = "Customer", action = "List",
        });

    routes.MapRoute("ChromeRoute", "{*catchall}",
        new { controller = "Home", action = "Index" },
        new {
            customConstraint = new UserAgentConstraint("Chrome")
        },
        new[] { "UrlsAndRoutes.AdditionalControllers" });

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

Powyższa ścieżka powoduje mapowanie żądań adresu URL `/Content/StaticContent.html` na akcję `List` kontrolera `Customer`. Działające mapowanie pokazano na rysunku 13.17. Pokazany na rysunku efekt otrzymasz po uruchomieniu aplikacji i przejściu do adresu URL `/Content/StaticContent.html`.



Rysunek 13.17. Przechwytywanie żądania pliku dyskowego z użyciem ścieżki

Routing żądań przeznaczonych dla plików dyskowych wymaga dokładnego przemyślenia, ponieważ wzorce URL będą dopasowywać adresy tego typu równie chętnie, jak wszystkie inne. Na przykład żądanie dla `/Content/StaticContent.html` może być dopasowane do wzorca URL takiego jak `{controller}/{action}`. Jeżeli nie będziesz ostrożny, może się to skończyć bardzo dziwnymi wynikami i obniżoną wydajnością. Dlatego wykorzystanie tej opcji powinno być ostatecznością.

Pomijanie systemu routingu

Użycie właściwości `RouteExistingFiles`, przedstawionej w poprzednim punkcie, powoduje, że system routingu zaczyna obsługiwać więcej żądań. Żądania, które normalnie pomijały system routingu, są teraz dopasowywane do zdefiniowanych ścieżek.

Przeciwnieństwem tej funkcji jest możliwość ograniczenia liczby adresów URL dopasowanych do naszych ścieżek. Realizujemy to przez użycie metody `IgnoreRoute` z klasy `RouteCollection`, która jest pokazana na listingu 13.35.

Listing 13.35. Użycie metody `IgnoreRoute`

```
...
public static void RegisterRoutes(RouteCollection routes) {

    routes.RouteExistingFiles = true;

    routes.IgnoreRoute("Content/{filename}.html");

    routes.MapRoute("DiskFile", "Content/StaticContent.html",
        new {
            controller = "Customer", action = "List",
        });

    routes.MapRoute("ChromeRoute", "{*catchall}",
        new { controller = "Home", action = "Index" },
        new {
            customConstraint = new UserAgentConstraint("Chrome")
        },
        new[] { "UrlsAndRoutes.AdditionalControllers" });

    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new[] { "UrlsAndRoutes.Controllers" });
}
...
```

Do dopasowania zakresów adresów URL możemy użyć zmiennych segmentu, takich jak `{filename}`. W tym przypadku wzorzec URL będzie dopasowywał dowolne trójsegmentowe adresy URL, których pierwszym segmentem jest `Content`; drugi segment ma rozszerzenie `.html`.

Metoda `IgnoreRoute` tworzy wpis w `RouteCollection`, w której obiekt zarządzania ścieżki jest typu `StopRoutingHandler` zamiast `MvcRouteHandler`. System routingu ma wbudowany kod rozpoznający ten obiekt obsługi. Jeżeli wzorzec URL przekazany do metody `IgnoreRoute` zostanie dopasowany, to nie będą analizowane kolejne ścieżki, tak jak w przypadku dopasowania ścieżki standardowej. Z tego powodu ważne jest również miejsce, w którym umieszczone jest wywołanie metody `IgnoreRoute`. Jeżeli uruchomisz aplikację i ponownie przejdziesz do adresu URL `/Content/StaticContent.html`, będziesz mógł zobaczyć zawartość wskazanego pliku HTML. Wynika to z faktu przetworzenia obiektu `StopRoutingHandler`, zanim jakkolwiek ścieżka będzie mogła dopasować adres URL.

Podsumowanie

W tym rozdziale przedstawiłem szczegółowo system routingu. Pokazałem, jak są dopasowywane i obsługiwane przychodzące żądania URL, jak dostosować ścieżki do własnych potrzeb przez zmianę sposobu dopasowywania segmentów URL i przez używanie wartości domyślnych oraz segmentów opcjonalnych. Zdemontowałem także sposób ograniczania ścieżek w wyniku zmniejszania zakresu dopasowywanych żądań i pokazałem, jak obsługiwać żądania treści statycznej.

W następnym rozdziale pokażę, jak generować wychodzące żądania URL ze ścieżek w widokach oraz jak korzystać z funkcji obszarów na platformie MVC. Wspomniana funkcja opiera się na systemie routingu i można ją wykorzystać do zarządzania ogromnymi i skomplikowanymi aplikacjami opartymi na technologii ASP.NET MVC.

Skorowidz

A

Active Server Pages, 24

adres

URL, 30, 38, 40, 144, 188, 203, 223, 316, 396, 545, 546, 573, 655

domyślny, 325, 326, 327

generowanie, 361, 362, 363

generowany na podstawie schematu, 354

ograniczenia, 342

przychodzący, 315, 318, 322, 326, 330, 335, 339, 345, 365

przyjazny dla użytkownika, 375

segment, 318, 331, 337, 343, 356

segment opcjonalny, 335, 336, 337, 338

segment statyczny, 327, 331

segment zmienny, 359

składany, 195

skojarzony z metodą akcji, 65

w pełni kwalifikowany, 360

wychodzący, 206, 315, 345, 353, 356, 357, 358, 367

żądania, 315

względny, 361

agregat, 70, 71

Ajax, 24, 25, 26, 28, 543, 547, 626

formularz nieprzeszkadzający, 550

łącza, 558

opcje, 554

wykrywanie żądań, 567

wywołanie

nieprzeszkadzające, 547, 549, 550, 553, 554, 555

zwrotne, 560

akcja, 448

kontrolera API, 657

metoda, *Patrz:* metoda akcji

nazwa, 450, 451, 455

podrzędna, 477, 486, 487, 496

potomna, 207

testowanie, 386

użytkownika, 63

wynik, 379, 385, 387, 405, 424

buforowanie, 433

wywołanie, 447, 448, 449

aktywator kontrolerów, *Patrz:* kontroler aktywator

Amazon Kindle Fire, 643

antywzorzec, 66

API, 31, 298, 318, 494, 633, 647, 648, 658

format danych, 656

aplikacja, 167, 168

dla urządzeń mobilnych, 640, 641

domena, 46

dystrybucja, 668

kompilowanie, 48

sieciowa, 625

wdrażanie, 667, 676

architektura

model-widok, 66

model-widok-kontroler, *Patrz:* MVC

model-widok-prezenter, *Patrz:* MVP

model-widok-widok-model, *Patrz:* MVVM

MVC, *Patrz:* MVC

obsługująca intensywne testowanie, 133

trójwarstwowa, 67

arkusz stylów, *Patrz:* CSS

ASP, *Patrz:* Active Server Pages

ASPX, 37, 465, 472

asynchroniczność, 28
atak, 497
AutoFac, 27
Azure, *Patrz:* Windows Azure

B

baza

danych, 50

- administracja zdalna, 674
- konfiguracja połączenia, 670
- kontrola wersji, 32
- migracja, 32
- na podstawie klas modelu, 183
- nierelacyjna, 28
- obiektowa, 71
- relacyjna, 70, 71, 179
- SQL Server, 35, 179
- zapisywanie zdjęć, 289

dokumentów, 70, 71

BDD, *Patrz:* programowanie sterowane zachowaniami

bezpieczeństwo, 496, 497, 555, 564

bezstanowość, 23, 32, 63

biblioteka

GUI, 67

imitacyjna, 161

JavaScript, *Patrz:* JavaScript

jQuery, *Patrz:* jQuery

jQuery Mobile, *Patrz:* jQuery Mobile

jQuery UI, *Patrz:* jQuery UI

Knockout, *Patrz:* Knockout

kolekcji generycznych, 91

LINQ, 102

Modernizr, *Patrz:* Modernizr

Moq, *Patrz:* Moq

narzędziowa, 169

ORM, 65

skryptów, 625

błąd, 399, 603, 608

usuwanie, 35

C

CDN, 29

CGI, *Patrz:* Common Gateway Interface

chmura, 667

Chrome, 604

Common Gateway Interface, 24

cookie, 223, 280

CouchDB, 28

CRUD, 167, 251

Cruise Control, 27

CSS, 25, 26, 29, 58, 196, 211, 262, 299, 632

członkostwo, 30

D

DAL, 67

DDD, 68, 70

debugowanie, 38, 169, 172, 302, 304, 306, 307, 309,

310, 566, 626, 670

widoków Razor, 309

delegat, 97, 100

DI, 74, 75, 78, 136, 171, 241

domena

klasa, *Patrz:* klasa domeny

model, *Patrz:* model domeny

modelowanie, 68

typ, 64

dostawca wartości, 382

E

Edit and Continue, *Patrz:* Visual Studio Edit and Continue

edytor kodu, 35, 307

e-mail, 241

encja, 64, 69, 169, 174

główna, 70

koszyka, 217

Entity Framework, 106, 179, 265, 535

kontekstu, 183

etykieta, 530

F

Fiddler, 566, 631

filtr, 278, 379, 403, 405, 427, 429

Action, 405

akcji, 420, 430, 433

Authorization, 405

Authorize, 278, 279

autoryzacji, 407, 408, 409, 410, 430, 433

własny, 409

bez atrybutów, 427

Exception, 405

globalny, 429, 430, 433

kolejność wykonywania, 430, 433
 kontrolera, 279
 OutputCache, 433, 434
 RequireHttps, 433
 Result, 405
 stos, 432
 uwierzytelniania, 278
 wbudowany, 433
 wyjątków, 411, 413, 415, 433
 wbudowany, 418, 419
 wyniku, 424, 430
 filtrowanie według kategorii, 203, 205
 Firefox, 38, 84, 604
 formularz, 51, 380, 433, 500, 502
 Ajax, 552
 nieprzeszkadzający, 550
 obsługujący, 628
 HTML, 222, 625
 znacznik, 50
 synchroniczny, 547, 548, 552
 tworzenie, 504
 ukryte pole, 528
 uwierzytelnianie, 277, 280
 funkcja
 code-first, 183
 SQL Server, 180
 uwierzytelniania formularzy, 277

G

getter, 89
 Google Web Accelerator, 376
 GridView, 194
 GUI, *Patrz:* interfejs użytkownika graficzny
 Guthrie Scott, 28

H

hasło, 277, 278, 299
 HTML, 23, 25, 26, 29, 31, 47, 63, 188, 477, 489, 492
 generowanie kodu, 388
 HTML5, 26
 HTTP, 23, 25, 31, 221, 343
 kody, 399, 400

I

IDC, *Patrz:* Microsoft Internet Database Connector
 IDE, 133, 149
 IIS, 667
 IntelliSense, 381, 392, 634

intencja
 definiowanie, 385
 wykonywanie, 385
 interfejs, 93
 .NET, 29
 C#, 137
 dostawcy uwierzytelniania, 280
 edycji, 261
 IActionFilter, 405, 426, 427
 IActionInvoker, 466
 IAuthorizationFilter, 407, 409, 427
 IController, 378, 441, 447, 654
 IControllerFactory, 301
 IDiscountHelper, 162, 163
 IExceptionFilter, 418
 IExecutionFilter, 427
 IHandler, 368
 implementacja, 133
 IOrderProcessor, 241, 243
 IProductRepository, 174
 IQueryable, 106
 IResultFilter, 405, 426, 427
 IRouteConstraint, 344
 IRouteHandler, 368
 IViewEngine, 465
 Ninject.IKernel, 140
 repozytorium, 649
 użytkownika, 23, 55, 64, 65, 68, 169
 automatyzacja, 84
 do przesyłania plików, 288
 graficzny, 23
 prototypowanie, 66
 Web API, *Patrz:* API
 internacjonalizacja, 30
 Internet Explorer, 38, 84, 144, 172, 630, 631
 inversion of control, *Patrz:* IoC
 IoC, 74, *Patrz też:* DI

J

JavaScript, 25, 26, 28, 169, 269, 299, 300, 346, 625
 V8, 28
 wyłączona obsługa, 555, 559
 jQuery, 26, 28, 29, 169, 299, 543, 547, 550, 625, 626,
 637, 658
 jQuery Mobile, 26, 169, 625, 640
 jQuery UI, 26, 169, 625
 jQuery Validation, 621, 625, 626
 JSON, 26, 562, 647
 wyświetlenie danych, 566

K

- kalendarz, 29
- klasa, 38, 46, 88, 93, 137
 - abstrakcyjna, 29
 - ActionFilterAttribute, 405, 426
 - ActionMethodSelectorAttribute, 453
 - AjaxOptions, 553, 560
 - ApiController, 657
 - Assert, 152, 153
 - AuthorizeAttribute, 410, 411
 - ControllerActionInvoker, 466
 - ControllerContext, 230, 411
 - CSS, 197
 - DefaultControllerFactory, 443
 - domeny, 111
 - ExceptionContext, 412
 - fabryki kontrolerów, 443
 - FilterAttribute, 413
 - FormsAuthentication, 280
 - HandleErrorAttribute, 419
 - HttpStatusCodeResult, 400
 - imitacja, 322
 - implementacja, 146
 - It, 162
 - kontrolerów, 299
 - modelu, 68, 134, 177
 - częściowa, 535
 - domeny, 299
 - generowana automatycznie, 535
 - widoku, 299
 - nazwa niekwalifikowana, 339
 - NinjectControllerFactory, 171
 - osłonowa, 91
 - RangeExceptionAttribute, 413
 - RouteBase, *Patrz:* RouteBase
 - RouteCollection, 320
 - System.Mvc.Web.Html.InputExtensions, 604
 - System.Web.Http.ApiController, 654
 - System.Web.Mvc.Controller, 379, 654
 - System.Web.Security.FormsAuthentication, 280
 - testowanie, 158
 - ViewContext, 493
 - ViewEngineResult, 467
 - zaprzyjaźniona, 535, 536
- klucz główny, 182
- Knockout, 169, 625, 626
- kod
 - HTML, 388
 - HTTP, 399, 400
 - obsługi trwałości, 67
 - wbudowany, 477
- kolekcja, 90, 129, 130, 251, 586, 587, 588
 - filtrowanie, 95
 - generyczna, 91, 95
 - obiektów, 134
- kompilacja, 173
- kompilator, 35, 109
- komunikat, 267, 268, 557
 - błędu, 417
 - kontroli poprawności, 604, 608
 - o błędach, 399
- konstruktor, 245
 - konfiguracja parametrów, 76
 - oczekujący, 176
- kontener
 - DI, 75, 76, 133, 136, 171, 241
 - IoC, *Patrz:* kontener DI
- kontrola
 - poprawności, 54, 57, 68, 248, 269, 595, 599, 604, 608, 616
 - jawna, 602
 - jQuery, 621
 - nieprzeszkadzająca, 618
 - po stronie klienta, 269, 271, 284, 618, 619, 620, 621
 - w łączniku modelu, 609
 - zasady poprawności, 610, 612, 613, 614
 - zdalna, 621
 - wersji bazy danych, 32
- kontroler, 27, 29, 38, 40, 43, 64, 68, 85, 135, 142, 169, 299, 301, 315, 368, 371, 377, 405, 406, 429, 650
- aktywator, 445, 446
- API, 647, 648, 650, 655, 656
 - akcja, 657
 - tworzenie, 653
- asynchroniczny, 28, 458, 459, 461
 - tworzenie, 461
- bezstanowy, 456, 457
- CRUD, 251
- fabryka, 437, 441, 446, 447, 466
 - domyślna, 441
 - tworzenie, 439, 442, 445
 - wbudowana, 443, 457
- koszyka na zakupy, 222
- nawigacji, 207
- niejednoznaczność, 373
- poprawiający wydajność, 456
- priorytet, 339

rozszerzanie, 437
 testowanie, 386
 tworzenie, 378, 379, 380, 445
 zapasowy, 441
 kontrolka, 23, 47, 625
 GridView, 194
 nawigacji, 201
 serwerowa, 31
 UI, 32
 użytkownika
 dla urządzeń mobilnych, 625
 wprowadzania danych
 silnie typowana, 512
 kontynuacja, 108
 konwencja
 dla klas kontrolerów, 301
 dla układów, 302
 dla widoków, 301
 koszyk na zakupy, 216, 222, 224, 232, 234, 244
 kreator projektów testów, 30

L

Language Integrated Query, *Patrz:* LINQ
 LINQ, 31, 83, 102, 106, 162, 179
 Parallel, 106
 to Entities, 106
 to Objects, 106
 to XML, 106
 lista, 47
 kategorii, 208
 posortowana alfabetycznie, 209
 rozwijana, 57
 LocalDB, 180
 logika
 aplikacji, 377
 biznesowa, 66, 123, 169

Ł

łańcuch zależności, 76
 łącznik modelu, *Patrz:* model łącznik

M

Mac OS X, 32
 mapowanie obiektowo-relacyjne, 27, 179
 mechanizm
 dołączania modelu, *Patrz:* model dołączanie

rozwiązywania zależności, *Patrz:* zależność
 stanu sesji, 223
 ViewBag, 212
 wstrzykiwania zależności, *Patrz:* DI
 menu
 kaskadowe, 29
 nawigacji, 207
 metadane, 527, 530, 531, 533, 535, 538, 610, 614
 metoda, 448
 .ToString, 178
 @RenderSection, 478
 Action, 208
 ActionLink, 47, 211, 484
 AddBindings, 171
 Ajax.BeginForm, 554
 akcji, 38, 41, 43, 65, 113, 120, 123, 178, 229, 260,
 282, 300, 316, 362, 368, 379, 388, 390, 405, 424,
 455, 573, *Patrz:* metoda publiczna
 alias, 329
 niejednoznaczność, 455
 parametry, 380, 382, 383
 przekazywanie danych do widoku, 391
 selektor, 453
 tworzenie, 48
 wywoływana z widoku, 486
 AreaRegistration.RegisterAllAreas, 320, 371
 AreEqual, 152
 AreNotEqual, 152
 AsQueryable, 175
 Assert.AreEqual, 152
 asynchroniczna, 107, 108, 109, 461
 Authenticate, 280
 AuthorizeCore, 408, 410
 BeginForm, 50, 51, 221, 554, *Patrz też:* metoda
 Html.BeginForm
 BeginRouteForm, 507, 508
 Bind, 140, 142
 BundleConfig.RegisterBundles, 320
 C#, 573
 CanChangeLoginName, 78
 CRUD, 252
 DefaultControllerFactory, 447
 DisplayForModel, 536
 EditorForModel, 261, 530, 536, *Patrz też:* metoda
 Html.EditorForModel
 Execute, 378, 379
 ExecuteResult, 385
 FilterConfig.RegisterGlobalFilters, 320
 formatująca, 178

metoda

GetControllerSessionBehavior, 442
 GetImage, 290
 GetModelStateValue, 604
 GetRouteData, 366
 GetService, 142
 GetServices, 142
 HandleNonHttpRequest, 433
 Html.ActionLink, 354, 362
 Html.BeginForm, 504, 506, 507
 Html.Editor, 519
 Html.EditorFor, 240, 519
 Html.EditorForModel, 241, 269, 536
 Html.EndForm, 504
 Html.HiddenFor, 234
 Html.HttpMethodOverride, 664
 Html.TextBoxFor, 240
 Html.ValidationSummary, 605, 606
 HTML, 188, 189, 477
 HTTP, 343, 344
 IsAny, 162
 IsInRange, 165
 It.Is, 165
 Label, 530
 LabelFor, 530
 LabelForModel, 530
 List, 301
 łańcuch wywołań, 147
 łączenie warunkowe, 148
 MapPageRoute, 321
 MapRoute, 321
 ModelState.AddModelError, 602
 Ninject, 244
 obsługi zdarzeń, 65
 OnAuthorization, 409
 OnException, 411
 opóźniona, 105, 106
 pomocnicza, 47, 200, 221, 234, 322, 489, 492, 496, 530
 generująca adresy URL, 545, 546
 kodowanie ciągów tekstowych, 496
 kodowanie treści, 498
 obsługująca formularze, 500
 obsługująca kontrolę poprawności, 604
 rusztowanie, *Patrz:* rusztowanie
 silnie typowana, 512
 szablonowa, 240, 517, 519, 522, 525, 527, 533, 538, 539, 603
 tworząca znacznik select, 513

tworzenie, 491
 wbudowana, 500
 wewnętrzna, 491, 492, 495
 widoku szablonowego, 240
 własna, 495
 wprowadzanie danych, 508, 509
 zewnętrzna, 493
 POST, 221
 publiczna, *Patrz:* metoda akcji
 Redirect, 395, 396
 RedirectToAction, 362
 RegisterGlobalFilters, 429
 ReleaseController, 442
 RenderAction, 208
 RenderPartial, 200, 208
 Response.Write, 383
 Returns, 162
 RouteConfig.RegisterRoutes, 320
 RouteLink, 211, 213
 rozszerzająca, 91, 92, 93, 103, 104
 filtrująca, 95
 LINQ, 104, 105
 SetAuthCookie, 280
 Setup, 162
 statyczna, 152, 153, 162, 280
 string.IsNullOrEmpty, 602
 synchroniczna, 461
 TestIncomingRouteResult, 323
 TextBoxFor, 50, *Patrz też:* metoda
 Html.TextBoxFor
 To, 140, 142
 ToConstant, 175
 Url.Action, 362
 UrlHelper.GenerateURL, 357
 View, 41, 391, 603
 WebApiConfig.Register, 320
 widoku, 120
 WithConstructorArgument, 244
 WithPropertyValue, 146
 metodologia zwinna, *Patrz:* programowanie zwinne
 Microsoft Internet Database Connector, 24
 Microsoft's Entity Framework, 32
 model, 46, 64, 113, 371
 dołączanie, 53, 57, 68, 229, 380, 571, 573, 579, 591, 609
 kolekcji, 584, 586
 obsługa błędów, 590
 selektywne właściwości, 583
 tablic, 584
 wywołanie jawne, 589

- domeny, 46, 55, 64, 66, 67, 69, 111, 123, 168, 173, 230, 269, 299, 377
 - implementacja, 65
 - trwały, 64
 - klasa, *Patrz:* klasa modelu
 - kojarzenie z bazą danych, 184
 - kontrola poprawności, *Patrz:* kontrola poprawności
 - łącznik, 229, 232, 265, 282, 382, 502, 574, 581, 594, 595, 596, 609
 - domyślny, 575
 - metadane, 68, 262, 527
 - obiekt, 115
 - samokontrolujący, 616
 - typ, 115
 - widoku, 64, 68, 299, 309, 316, 392
 - właściwość, 50
 - wyświetlanie danych, 531
 - ModelState, 383
 - Modernizr, 169, 626
 - Mongo, 28
 - MonoRail, 32
 - Moq, 27, 133, 157, 158, 159, 163, 280
 - MSTest, 30
 - MVC, 27, 28, 29, 30, 31, 38, 46, 65, 68, 133, 159, 167, 229, 297, 625
 - aplikacja, 35, 299
 - wprowadzania danych, 45
 - bezpieczeństwo, 278
 - elementy projektu, 299, 300
 - konwencja, 301
 - migracja z Web Forms, 32
 - odmiany, 67
 - programowanie po stronie klienta, 628
 - wzorzec projektowy, *Patrz:* wzorzec MVC
 - MVC 4, 298
 - MVP, 67
 - MVVM, 68
- N**
- narzędzie
 - F12, 630, 631
 - imitujące, 79, 133
 - National Center for Supercomputing Applications, 24
 - nawigacja, 207
 - nazwa kwalifikowana, 55
 - NCSA, 24
 - NHibernate, 27, 32
 - Ninject, 27, 76, 133, 136, 138, 139, 141, 144, 148, 241
 - Node.js, 28
 - notacja kropki, 103
 - NuGet, 138, 139, 159, 640
 - NUnit, 27, 30, 133, 149
- O**
- obiekt, 70
 - AuthorizationContext, 409
 - C#, 229
 - ControllerContext, 380
 - cykl życia, 223
 - czas życia, 76
 - dołączanie, 334
 - dynamiczny, 44
 - filtrowanie kolekcji, 95
 - HttpContextBase, 409
 - imitacyjny, 160, 175
 - tworzenie, 161
 - inicjalizator, 90, 93, 100
 - kernel Ninject, 140
 - kolekcja, *Patrz:* kolekcja obiektów kontekstu, 380, 381
 - ModelState, 383
 - modelu domeny, 123, 230
 - modelu widoku, 392
 - obsługi ścieżki, 368
 - RedirectToRouteResult, 362
 - stanu sesji, 223
 - tworzenie, 89
 - ViewBag, 43, 213, 394
 - wywołujący akcje, 437
 - zarządzanie czasem życia, 76
 - obsługa stanu, 32
 - obsługa łańcucha zależności, 76
 - obszar, 300, 369, 371
 - tworzenie, 369
 - odwrócenie kontroli, 74, *Patrz też:* DI
 - Open Source Initiative, *Patrz:* OSI
 - Opera, 84
 - Opera Mobile, 643
 - Opera Mobile Emulator, 641, 643
 - operacja, 69
 - optymalizacja, 630, 631, 640
 - ORM, *Patrz:* mapowanie obiektowo-relacyjne
 - OSI, 31

P

paczka, 625, 632, 635, 670
 arkuszy stylów, 633
 skryptów, 633
 ścieżka dostępu, 633
 platforma ORM, 179
 plik
 .cshtml, 42
 IntelliSense, 634
 konfiguracyjny, 300
 skryptów, 632
 ViewStart, 119
 widoku, 122
 zminimalizowany, 626, 627, 634
 poczta elektroniczna, 241
 pole
 prywatne, 88
 tekstowe, 47, 57, 604
 wyboru, 47, 604
 powiązanie luźne, 73
 prefiks, 580, 581
 prezydent, 67, 68
 profil, 30
 profilowanie, 638
 programowanie
 równoległe, 107, 108
 sieciowe, 26
 sterowane domeną, *Patrz:* DDD
 sterowane testami, 26, 76, 79, 83
 sterowane zachowaniami, 26
 zwinne, 26
 protokół
 HTTP, *Patrz:* HTTP
 HTTPS, 433
 przeglądarka
 automatyzacja, 84
 internetowa, 38, 144, 300, 643, 655
 symulowanie, 84
 przekierowanie, 395, 396, 397, 398
 trwałe, 395, 397
 tymczasowe, 395, 397
 przerwanie, 306
 przestrzeń nazw, 132, 339, 341, 536
 domyślna, 371
 MyProject, 445
 priorytet, 340, 444
 System.ComponentModel.DataAnnotations, 262
 System.Web.Mvc, 378

przycisk, 221, 222
 puła wątków roboczych, 458, 459
 punkt
 przerwania, 306, 307, 309
 zatrzymania, 306
 Python, 32

R

Rails, *Patrz:* Ruby on Rails
 Razor, 33, 37, 54, 65, 111, 120, 126, 177, 389, 465,
 472, 473, 475, 478, 484
 blok kodu, 116
 instrukcja, 114
 konstrukcje warunkowe, 127, 129
 składnia, 111
 wyrażenia, 123
 refaktoring, 199, 329
 kontrolera, 142
 relacja, 69
 repozytorium, 71, 179, 649
 imitacja, 175, 179
 magazynów dokumentów, 179
 obiektów, 179
 tworzenie, 184
 wzorzec, 174
 Representational State Transfer, *Patrz:* REST
 REST, 26, 30, 657
 Rhino Mock, 27, 133
 rola, 30
 RouteBase, 364
 routing, 27, 29, 30, 40, 195, 206, 299, 315, 318, 330,
 356, 369, 396, 409, 544
 dostosowanie, 364
 konfiguracja, 40, 175, 300, 355, 546
 pomijanie, 350
 tworzenie projektu, 316
 żądań dla plików dyskowych, 346
 Ruby on Rails, 27, 28, 29, 32, 301
 rusztowanie, 525, 526, 536
 wyłączanie właściwości, 530

S

Safari, 84
 Sanderson Steve, 626
 Sanderson Steven, 169
 schemat URL
 w postaci kontraktu dla użytkownika, 328
 dla aplikacji, 318

sekcja, 477, 478, 479, 481
 opcjonalna, 481
 Selenium RC, 84
 serwer
 aplikacji, 347
 Content Delivery Network, *Patrz:* CDN
 Internet Information Services, *Patrz:* IIS
 SMTP, 243
 Visual Studio, 349
 sesja, 223
 stan, 456, 457
 wyłączony, 458
 setter, 89
 silnik
 ASPX, *Patrz:* ASPX
 Razor, *Patrz:* Razor
 widoku, 37, 379, 465, 470
 tworzenie, 467, 468, 470
 Silverlight, 26, 68, 674
 Sinatra, 27, 28
 składnia <% %>, 111
 skrypt, 630, 632, 637
 słowo kluczowe
 async, 108, 109
 await, 108, 109
 foreach, 492
 model, 392
 return, 108
 select, 103
 this, 92
 using, 104, 152
 var, 100, 101
 yield, 95
 zarezerwowane, 360
 Smalltalk, 28, 63
 SMTP, 241
 SOAP, 26
 SQL, 102
 skrypt, 182
 SQL Server, 180
 standard sieciowy, 26
 stoper, 423
 strona wzorcowa, 30
 stronicowanie, 186, 187
 Subsonic, 27, 32
 system wyników akcji, 379, 385
 System.Linq, 104
 System.Web.Routing, 30
 szablon, *Patrz:* metoda pomocnicza szablonowa
 kolejność wyszukiwania, 539

ogólny, 539
 wbudowany, 540

Ś

ścieżka, 40, 195, 196, 205, 206, 301, 315, 339, 507
 definiowanie, 319
 domyślna, 40, 178
 dopasowanie, 356
 generowanie adresu URL, 363
 kolejność, 330
 nazwa, 321, 332, 363, 370
 ograniczenia, 342, 344
 metody HTTP, 343, 344
 własne, 344
 wyrażenia regularne, 342
 zbiór wartości, 343
 pliku na dysku, 349
 prosta, 324, 325
 rejestrowanie, 321
 własny obiekt obsługi, 368
 wzorzec, *Patrz:* wzorzec ścieżki
 zbiór, 318
 śledzenie, 139

T

tabela, 32, 675
 tablica, 90, 129, 130, 131, 584, 587
 Request.Form, 53
 Request.QueryString, 53
 TDD, 167, *Patrz:* programowanie sterowane testami
 TeamCity, 27
 test, 26
 testowanie, 30, 154
 automatyczne, 25, 27, 30, 32, 76
 integracyjne, 76, 84
 jednostkowe, 25, 26, 27, 30, 67, 76, 77, 79, 83, 84,
 133, 149, 151, 156, 158, 159, 160, 163, 167, 168,
 169, 203, 344
 generowanie metod testowych, 151
 nazewnictwo, 152
 routingu, 316, 322, 327
 tworzenie projektu, 150
 trwałość, 71
 tryb wyświetlania, 625
 typ
 anonimowy, 101
 złożony, 536
 typowanie niejawne, *Patrz:* wnioskowanie typów

U

UI, *Patrz:* interfejs użytkownika
 układ, 117, 119, 300, 302, 417, 478, 481, 549
 stosowanie, 118
 tworzenie, 116, 253
 wersja mobilna, 642
 współdzielony, 119
 Unity, 76, 133
 URI, 26
 URL, 27
 urządzenie przenośne, 298
 usługa RESTful, 657, 664
 ustawienia regionalne, 178, 577
 uwierzytelnianie, 30, 284, 298, 299, 403, 411, 640
 formularzy, *Patrz:* formularz uwierzytelnianie
 systemu Windows, 277
 z użyciem filtrów, 278
 użytkownik, 277, 278, 280, 299
 dane, 612

V

ViewBag, 212, 266, 282, 316, 332, 393, 394, 458, 489
 ViewData, 458
 ViewState, 25, 31
 Visual Studio, 30, 35, 36, 46, 85, 111, 133, 138, 149,
 151, 180, 297, 299, 536, 626
 błędy, 115
 debuger, 172, 304
 Edit and Continue, 310, 312
 IntelliSense, 50, 65
 okno, 308
 Call Stack, 308, 309
 Locals, 308, 309
 serwer, 349

W

warstwa dostępu do danych, *Patrz:* DAL
 WatiN, 84
 wątek
 roboczy, 458
 tła, 459
 Web API, *Patrz:* API
 Web Forms, 23, 24, 25, 26, 27, 28, 29, 31, 65, 67, 167,
 194, 300, 315
 migracja do MVC, 32

widok, 27, 29, 33, 41, 43, 53, 64, 68, 114, 117, 123,
 136, 169, 177, 300, 301, 309, 362, 363, 368, 371,
 377, 390, 415, 465, 473, 477, 625, 635
 aktualizacja, 63
 bezstanowość, 63
 beztypowy, 392
 C#, 42
 częściowy, 199, 209, 300, 301, 477, 483, 484, 487, 496
 tworzenie, 483
 typ, 485
 generowanie, 389
 z użyciem ścieżki, 391
 kompilacja, 668
 niezależny, 116
 przekazywanie danych z metody akcji, 391
 rusztowania, 256, 257, 260
 silnik, *Patrz:* silnik widoku
 słabo typowany, *Patrz:* widok beztypowy
 szablon wbudowany, 534
 ściśle określonego typu, 48
 treści dynamiczne, 477
 treści statyczne, 477
 tworzenie, 113
 wersja mobilna, 642
 wykrywanie błędów, 668
 wyszukiwanie, 475
 Windows Azure, 667, 671
 Windows Presentation Foundation, *Patrz:* WPF
 właściwość
 AppRelativeCurrentExecutionFilePath, 322
 automatyczna, *Patrz:* właściwość automatycznie
 implementowana
 automatycznie implementowana, 88
 Canceled, 423
 class, 360
 controller, 178
 Controller.ActionInvoker, 448
 Controller.ControllerContext, 380
 Controller.ViewBag, 393
 Date, 393
 dynamiczna, 491
 EmailSettings.WriteAsFile, 243
 Exception, 423
 ExceptionHandled, 423
 Greeting, 44
 HttpContext, 230, 380
 IDENTITY, 182
 Layout, 119
 Message, 393

- ModelState.IsValid, 266
 - Request, 380
 - Response, 380
 - RouteData, 380
 - RouteData.Values, 334
 - RouteExistingFiles, 349
 - RouteTable.Routes, 320
 - Server, 380
 - Server.MachineName, 384
 - statyczna, 320
 - ukrywanie, 529
 - ułatwiająca, 380
 - ViewBag.ProductCount, 128
 - ViewContext, 493
 - ViewResult.ViewData.Model, 392
 - wartość, 146
 - WillAttend, 46, 55
 - wyłączanie, 530
 - wnioskowanie typów, 100
 - WPF, 68
 - wstrzykiwanie, 176
 - implementacji, 148
 - konstruktora, 144
 - wartości do konstruktora, 147
 - za pomocą konstruktora, 74
 - za pomocą settera, 74
 - zależności, 141, 144, 176, 442
 - wyjątek, 306, 309, 415
 - nieobsłużony, 411, 430
 - obsłużony, 309
 - wyrażenie
 - @Model, 124, 136
 - @using, 132
 - @ViewBag, 124
 - lambda, 50, 83, 97, 98, 99, 100, 108, 162, 512
 - Razor, *Patrz:* Razor wyrażenia
 - regularne, 445
 - wywołanie
 - postback, 31
 - zwrotne, 28
 - wzorzec
 - architektury oprogramowania, 65
 - DI, *Patrz:* DI
 - konwencja przed konfiguracją, *Patrz:* zasada konwencja przed konfiguracją
 - MVC, 63, 64
 - POST-Redirect-GET, 395
 - repozytorium, *Patrz:* repozytorium wzorzec
 - Smart UI, 64, 65, 66
 - ścieżki, 332
 - URL, 27, 318, 330
 - konserwatywny, 319
 - liberalny, 319
- X**
- XML, 26
 - xUnit, 27, 30
- Y**
- Yahoo! UI Library, 29
- Z**
- zachowanie oczekiwane, 26
 - zadanie
 - przekrojowe, 403
 - separacja, 63, 73, 336
 - zależność, 139, 141, 142, 144, 170, 171, 446
 - łańcuch, 144
 - wstrzykiwanie, *Patrz:* wstrzykiwanie zależności
 - zapytanie, 380
 - LINQ, 102
 - opóźnione, 105
 - składnia, 103
 - zasada
 - czerwone-zielone-refaktoryzacja, 79
 - konwencja przed konfiguracją, 301, 444
 - zbiór
 - ścieżek, *Patrz:* ścieżka zbiór
 - zasobów, *Patrz:* URI
 - zdarzenie
 - klienckie, 23
 - obsługa, 65
 - zdjęcia, 286, 290, 346
 - wyświetlanie, 292
 - zintegrowane środowisko programistyczne, *Patrz:* IDE
 - zmienna przechwytyjąca, 337, 338
 - znacznik, 65
 - formularza HTML, 50
 - HTML, 25, 504, 533
 - łącza (<a>), 360
 - select, 513

Ż

żądanie, 23, 28, 318, 383, 441

Ajax, 647

GET, 52, 221, 376, 395, 433, 564, 657

HTTP, 229, 451, 573

klienta, 64

POST, 52, 244, 265, 273, 376, 395, 433, 564

przechwytywanie, 446

przychodzące, 206, 315, 377

sieciowe profilowanie, 631

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 MVC jest rewelacyjną alternatywą dla ASP.NET Web Forms. Pozwala na połączenie efektywności ze schludnością architektury model-widok-kontroler (MVC). Nowa wersja platformy – ASP.NET MVC 4 – to kolejny milowy krok w rozwoju platformy ASP.NET stworzonej przez giganta z Redmond. Dzięki ASP.NET MVC 4 błyskawicznie stworzysz łatwe w utrzymaniu i rozwojowe aplikacje internetowe.

Lektura tej książki pozwoli Ci zapoznać się z technikami programowania opartego na testach (TDD). Przekonasz się, jak działa wzorzec MVC w praktyce. Ponadto dogłębnie poznasz całą platformę ASP.NET MVC oraz różnice, wady i zalety ASP.NET MVC względem klasycznego ASP.NET. Przekonasz się, jak zastosowanie filtrów może ułatwić Ci życie oraz jak niezwykle istotną kwestią jest zapewnienie bezpieczeństwa Twojej aplikacji. Książka ta jest doskonałym i kompletnym źródłem wiedzy na temat ASP.NET MVC. Obowiązkowa lektura dla każdego programisty tej platformy!

Sięgnij po książkę i sprawdź:

- jak wykorzystanie architektury MVC może ułatwić Ci pracę
- jaki wpływ na jakość Twojego kodu ma architektura MVC
- jak zapewnić bezpieczeństwo Twojej aplikacji
- w jaki sposób implementowana jest architektura MVC

Apress®

nr katalogowy: 14573

Księgarnia internetowa:
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900

helion.pl
księgarnia
internetowa

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najczęściej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowosci>

Helion

Helion SA
ul. Koszuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>



cena 99,00 zł

ISBN 978-83-246-7299-8



9 788324 672998

Informatyka w najlepszym wydaniu