

TWÓRZ ROZBUDOWANE APLIKACJE WEBOWE!

Apress®

# ASP.NET MVC 3 Framework Zaawansowane programowanie

Steven Sanderson, Adam Freeman

Helion 

Tytuł oryginału: Pro ASP.NET MVC 3 Framework

Tłumaczenie: Paweł Gonera

ISBN: 978-83-246-4822-1

Original edition copyright © 2011 by Adam Freeman and Steven Sanderson.  
All rights reserved.

Polish edition copyright © 2012 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](mailto: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/aspvm3.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
<http://helion.pl/user/opinie/aspvm3>  
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 autorach .....	5
	O recenzencie technicznym .....	7
<b>Część I</b>	<b>Wprowadzenie do ASP.NET MVC 3 .....</b>	<b>19</b>
<b>Rozdział 1.</b>	<b>Zagadnienia ogólne .....</b>	<b>21</b>
	Krótką historią programowania witryn WWW .....	21
	Tradycyjna technologia ASP.NET Web Forms .....	21
	Co poszło nie tak z ASP.NET Web Forms? .....	23
	Programowanie witryn WWW — stan obecny .....	23
	Standardy WWW oraz REST .....	24
	Programowanie zwinne i sterowane testami .....	24
	Ruby on Rails .....	25
	Sinatra .....	25
	Node.js .....	25
	Najważniejsze zalety ASP.NET MVC .....	26
	Architektura MVC .....	26
	Rozszerzalność .....	26
	Ścisła kontrola nad HTML i HTTP .....	27
	Łatwość testowania .....	27
	Zaawansowany system routingu .....	28
	Zbudowany na najlepszych częściach platformy ASP.NET .....	28
	Nowoczesne API .....	28
	ASP.NET MVC jest open source .....	29
	Kto powinien korzystać z ASP.NET MVC? .....	29
	Porównanie z ASP.NET Web Forms .....	29
	Migracja z Web Forms do MVC .....	30
	Porównanie z Ruby on Rails .....	30
	Porównanie z MonoRail .....	30
	Co nowego w ASP.NET MVC 3? .....	30
	Podsumowanie .....	31

<b>Rozdział 2. Przygotowania .....</b>	<b>33</b>
Przygotowanie stacji roboczej .....	33
Instalowanie Visual Studio 2010 .....	33
Instalowanie podstawowego oprogramowania .....	34
Instalowanie opcjonalnych komponentów .....	35
Przygotowanie serwera .....	37
Włączanie roli serwera WWW .....	37
Instalowanie dodatkowych komponentów .....	38
Konfigurowanie Web Deployment .....	39
Uzyskiwanie dalszych informacji .....	40
Podsumowanie .....	41
<b>Rozdział 3. Pierwsza aplikacja MVC .....</b>	<b>43</b>
Tworzenie nowego projektu ASP.NET MVC .....	43
Dodawanie pierwszego kontrolera .....	45
Przedstawiamy ścieżki .....	47
Generowanie stron WWW .....	47
Tworzenie i generowanie widoku .....	47
Dynamiczne dodawanie treści .....	50
Tworzenie prostej aplikacji wprowadzania danych .....	51
Przygotowanie sceny .....	51
Projektowanie modelu danych .....	52
Łączenie metod akcji .....	53
Budowanie formularza .....	55
Obsługa formularzy .....	57
Dodanie kontroli poprawności .....	60
Kończymy .....	63
Podsumowanie .....	65
<b>Rozdział 4. Wzorzec MVC .....</b>	<b>67</b>
Historia MVC .....	67
Wprowadzenie do wzorca MVC .....	67
Budowa modelu domeny .....	68
Implementacja MVC w ASP.NET .....	69
Porównanie MVC z innymi wzorcami .....	69
Przedstawiamy wzorzec Smart UI .....	69
Modelowanie domeny .....	72
Przykładowy model domeny .....	73
Wspólny język .....	73
Agregaty i uproszczenia .....	74
Definiowanie repozytoriów .....	75
Budowanie luźno połączonych komponentów .....	76
Wykorzystanie wstrzykiwania zależności .....	77
Przykład specyficzny dla MVC .....	78
Użycie kontenera wstrzykiwania zależności .....	79
Zaczynamy testy automatyczne .....	80
Zadania testów jednostkowych .....	80
Zadania testów integracyjnych .....	87
Podsumowanie .....	87
<b>Rozdział 5. Najważniejsze cechy języka .....</b>	<b>89</b>
Najważniejsze cechy C# .....	89
Użycie automatycznie implementowanych właściwości .....	89
Użycie inicjalizatorów obiektów i kolekcji .....	91

Użycie metod rozszerzających .....	93
Użycie wyrażenia lambda .....	97
Automatyczne wnioskowanie typów .....	98
Użycie typów anonimowych .....	99
Wykonywanie zapytań zintegrowanych z językiem .....	100
Przedstawiamy składnię silnika Razor .....	105
Tworzenie projektu .....	105
Prosty widok korzystający z silnika Razor .....	108
Podsumowanie .....	115
<b>Rozdział 6. Ważne narzędzia wspierające MVC .....</b>	<b>117</b>
Użycie Ninject .....	117
Tworzenie projektu .....	119
Zaczynamy korzystać z Ninject .....	119
Tworzenie łańcucha zależności .....	121
Definiowanie wartości właściwości i parametrów .....	122
Użycie samodzielnego łączenia .....	124
Dołączanie do typu pochodnego .....	124
Użycie łączenia warunkowego .....	125
Użycie Ninject w ASP.NET MVC .....	126
Testowanie jednostkowe w Visual Studio .....	128
Tworzenie projektu .....	128
Tworzenie testów jednostkowych .....	130
Uruchamianie testów (nieudane) .....	133
Implementacja funkcji .....	134
Użycie Moq .....	135
Dodawanie Moq do projektu Visual Studio .....	135
Tworzenie imitacji za pomocą Moq .....	135
Testy jednostkowe z użyciem Moq .....	137
Weryfikowanie przy użyciu Moq .....	139
Podsumowanie .....	139
<b>Rozdział 7. SportsStore — kompletna aplikacja .....</b>	<b>141</b>
Zaczynamy .....	142
Tworzenie rozwiązania i projektów w Visual Studio .....	142
Dodawanie referencji .....	143
Konfigurowanie kontenera DI .....	144
Uruchamiamy aplikację .....	145
Tworzenie modelu domeny .....	146
Tworzenie abstrakcyjnego repozytorium .....	147
Tworzenie imitacji repozytorium .....	148
Wyświetlanie listy produktów .....	148
Dodawanie kontrolera .....	148
Dodawanie widoku .....	149
Konfigurowanie domyślnej ścieżki .....	150
Uruchamianie aplikacji .....	151
Przygotowanie bazy danych .....	151
Tworzenie bazy danych .....	152
Definiowanie schematu bazy danych .....	153
Dodawanie danych do bazy .....	154
Tworzenie kontekstu Entity Framework .....	154
Tworzenie repozytorium produktów .....	156

Dodanie stronicowania .....	157
Wyświetlanie łączy stron .....	158
Ulepszanie adresów URL .....	165
Dodawanie stylu .....	165
Definiowanie wspólnej zawartości w pliku układu .....	166
Dodanie zasad CSS .....	167
Tworzenie widoku częściowego .....	167
Podsumowanie .....	169
<b>Rozdział 8. SportsStore — nawigacja i koszyk na zakupy .....</b>	<b>171</b>
Dodawanie kontrolki nawigacji .....	171
Filtrowanie listy produktów .....	171
Ulepszanie schematu URL .....	172
Budowanie menu nawigacji po kategoriach .....	175
Poprawianie licznika stron .....	182
Budowanie koszyka na zakupy .....	184
Definiowanie encji koszyka .....	185
Tworzenie przycisków koszyka .....	188
Implementowanie kontrolera koszyka .....	189
Wyświetlanie zawartości koszyka .....	190
Użycie dołączania danych .....	193
Tworzenie własnego łącznika modelu .....	193
Kończenie budowania koszyka .....	197
Usuwanie produktów z koszyka .....	197
Dodawanie podsumowania koszyka .....	198
Składanie zamówień .....	200
Rozszerzanie modelu domeny .....	200
Dodawanie procesu zamawiania .....	201
Implementowanie procesora zamówień .....	204
Rejestrowanie implementacji .....	206
Dokańczanie kontrolera koszyka .....	206
Wyświetlanie informacji o błędach systemu kontroli poprawności .....	210
Wyświetlanie strony podsumowania .....	210
Podsumowanie .....	212
<b>Rozdział 9. SportsStore — administracja .....</b>	<b>213</b>
Dodajemy zarządzanie katalogiem .....	213
Tworzenie kontrolera CRUD .....	214
Generowanie tabeli z produktami dostępnymi w repozytorium .....	215
Tworzenie nowego pliku układu .....	216
Implementowanie widoku listy .....	218
Edycja produktów .....	221
Tworzenie nowych produktów .....	232
Usuwanie produktów .....	232
Zabezpieczanie funkcji administracyjnych .....	235
Konfiguracja uwierzytelniania Forms .....	235
Realizacja uwierzytelniania z użyciem filtrów .....	236
Tworzenie dostawcy uwierzytelniania .....	238
Tworzenie kontrolera AccountController .....	239
Tworzenie widoku .....	240
Przesyłanie zdjęć .....	242
Rozszerzanie bazy danych .....	242
Rozszerzanie modelu domeny .....	244

Tworzenie interfejsu użytkownika do przesyłania plików .....	244
Zapisywanie zdjęć do bazy danych .....	245
Implementowanie metody akcji GetImage .....	246
Wyświetlanie zdjęć produktów .....	247
Podsumowanie .....	249
<b>Część II ASP.NET MVC 3 — szczegółowy opis .....</b>	<b>251</b>
<b>Rozdział 10. Przegląd projektu MVC .....</b>	<b>253</b>
Korzystanie z projektów MVC z Visual Studio .....	253
Użycie kontrolerów aplikacji internetowej i intranetowej .....	256
Przedstawiamy konwencje MVC .....	257
Debugowanie aplikacji MVC .....	258
Tworzenie projektu .....	258
Uruchamianie debugera Visual Studio .....	258
Przerywanie pracy aplikacji przez debugger Visual Studio .....	260
Użycie opcji Edit and Continue .....	263
Wstrzykiwanie zależności na poziomie projektu .....	266
Podsumowanie .....	268
<b>Rozdział 11. Adresy URL, routing i obszary .....</b>	<b>269</b>
Wprowadzenie do systemu routingu .....	269
Tworzenie projektu routingu .....	270
Wprowadzenie do wzorców URL .....	271
Tworzenie i rejestrowanie prostej ścieżki .....	272
Definiowanie wartości domyślnych .....	275
Użycie statycznych segmentów adresu URL .....	277
Definiowanie własnych zmiennych segmentów .....	280
Definiowanie opcjonalnych segmentów URL .....	282
Definiowanie ścieżek o zmiennej długości .....	282
Definiowanie priorytetów kontrolerów na podstawie przestrzeni nazw .....	284
Ograniczenia ścieżek .....	286
Routing żądań dla plików dyskowych .....	290
Pomijanie systemu routingu .....	292
Generowanie wychodzących adresów URL .....	293
Przygotowanie projektu .....	293
Generowanie wychodzących adresów URL w widokach .....	294
Generowanie wychodzących adresów URL w metodach akcji .....	299
Generowanie adresu URL na podstawie wybranej ścieżki .....	300
Dostosowanie systemu routingu .....	300
Tworzenie własnej implementacji RouteBase .....	301
Tworzenie własnego obiektu obsługi ścieżki .....	304
Korzystanie z obszarów .....	305
Tworzenie obszaru .....	306
Wypełnianie obszaru .....	307
Rozwiązywanie problemów z niejednoznacznością kontrolerów .....	308
Generowanie łączy do akcji z obszarów .....	310
Najlepsze praktyki schematu adresów URL .....	310
Twórz jasne i przyjazne dla człowieka adresy URL .....	310
GET oraz POST — wybierz właściwie .....	311
Podsumowanie .....	312

<b>Rozdział 12. Kontrolery i akcje .....</b>	<b>313</b>
Wprowadzenie do kontrolerów .....	313
Przygotowanie projektu .....	313
Tworzenie kontrolera z użyciem interfejsu IController .....	313
Tworzenie kontrolera przez dziedziczenie po klasie Controller .....	314
Odczytywanie danych wejściowych .....	316
Pobieranie danych z obiektów kontekstu .....	316
Użycie parametrów metod akcji .....	317
Tworzenie danych wyjściowych .....	319
Wyniki akcji .....	320
Zwracanie kodu HTML przez generowanie widoku .....	322
Przekazywanie danych z metody akcji do widoku .....	327
Wykonywanie przekierowań .....	331
Zwracanie danych tekstowych .....	333
Zwracanie danych XML .....	336
Zwracanie danych JSON .....	336
Zwracanie plików i danych binarnych .....	337
Zwracanie błędów i kodów HTTP .....	339
Tworzenie własnego wyniku akcji .....	341
Podsumowanie .....	343
<b>Rozdział 13. Filtry .....</b>	<b>345</b>
Użycie filtrów .....	345
Wprowadzenie do czterech podstawowych typów filtrów .....	346
Dołączanie filtrów do kontrolerów i metod akcji .....	347
Użycie filtrów autoryzacji .....	348
Użycie filtrów wyjątków .....	353
Użycie filtrów akcji i wyniku .....	356
Użycie innych funkcji filtrów .....	361
Użycie filtrów wbudowanych .....	366
Podsumowanie .....	369
<b>Rozdział 14. Rozszerzanie kontrolerów .....</b>	<b>371</b>
Komponenty potoku przetwarzania żądania .....	371
Tworzenie własnej fabryki kontrolerów .....	372
Definiowanie własnej fabryki kontrolerów .....	372
Rejestrowanie własnej fabryki kontrolerów .....	374
Wykorzystanie wbudowanej fabryki kontrolerów .....	374
Nadawanie priorytetów przestrzeniom nazw .....	374
Dostosowywanie sposobu tworzenia kontrolerów w DefaultControllerFactory .....	375
Tworzenie własnego obiektu wywołującego akcje .....	377
Użycie wbudowanego obiektu wywołującego akcje .....	378
Użycie własnych nazw akcji .....	379
Selekcja metod akcji .....	380
Użycie selektorów metod akcji do obsługi żądań REST .....	383
Poprawianie wydajności z użyciem specjalizowanych kontrolerów .....	385
Użycie kontrolerów bezstanowych .....	385
Użycie kontrolerów asynchronicznych .....	386
Podsumowanie .....	394



<b>Rozdział 15. Widoki</b> .....	<b>395</b>
Tworzenie własnego silnika widoku .....	395
Tworzenie własnej implementacji IView .....	397
Tworzenie implementacji IViewEngine .....	397
Rejestrowanie własnego silnika widoku .....	398
Korzystanie z silnika Razor .....	401
Sposób generowania widoków przez Razor .....	401
Wstrzykiwanie zależności w widokach Razor .....	402
Konfigurowanie wyszukiwania lokalizacji widoków .....	404
Dodawanie dynamicznych treści do widoku Razor .....	405
Użycie kodu wbudowanego .....	405
Użycie metod pomocniczych HTML .....	410
Tworzenie wewnętrznej metody pomocniczej HTML .....	411
Tworzenie zewnętrznej metody pomocniczej HTML .....	411
Użycie wbudowanych metod pomocniczych .....	413
Zastosowanie sekcji .....	425
Sprawdzanie istnienia sekcji .....	427
Generowanie sekcji opcjonalnych .....	428
Użycie widoków częściowych .....	428
Tworzenie widoku częściowego .....	428
Użycie silnie typowanych widoków częściowych .....	430
Użycie akcji podrzędnych .....	431
Tworzenie akcji podrzędnych .....	431
Wywoływanie akcji podrzędnych .....	432
Podsumowanie .....	433
<b>Rozdział 16. Szablony modelu</b> .....	<b>435</b>
Użycie metod pomocniczych widoku szablونowego .....	435
Nadawanie stylów wygenerowanym znacznikom HTML .....	438
Użycie metadanych modelu .....	440
Użycie metadanych wartości danych .....	443
Korzystanie z parametrów typów złożonych .....	447
Dostosowywanie systemu metod pomocniczych widoku szablونowego .....	448
Tworzenie własnego szablonu edytora .....	448
Tworzenie własnego szablonu wyświetlania .....	450
Tworzenie szablonu ogólnego .....	452
Wymiana szablonów wbudowanych .....	453
Użycie właściwości ViewData.TemplateInfo .....	454
Przekazywanie dodatkowych metadanych do szablonu .....	455
System dostawców metadanych .....	456
Tworzenie własnego dostawcy metadanych modelu .....	457
Dostosowywanie adnotacji danych dostawcy metadanych modelu .....	458
Podsumowanie .....	459
<b>Rozdział 17. Dołączanie modelu</b> .....	<b>461</b>
Użycie dołączania danych .....	461
Użycie domyślnego łącznika modelu .....	462
Dołączanie typów prostych .....	463
Dołączanie typów złożonych .....	464
Dołączanie tablic i kolekcji .....	467

Jawne wywoływanie dołączania modelu .....	469
Ograniczanie dołączania do wybranego źródła danych .....	470
Obsługa błędów dołączania modelu .....	471
Użycie dołączania modelu w celu pobierania przesłanych plików .....	472
Dostosowanie systemu dołączania danych .....	472
Tworzenie własnego dostawcy wartości .....	473
Tworzenie łącznika modelu obsługującego zależności .....	474
Tworzenie własnego łącznika modelu .....	475
Tworzenie dostawcy łączników modelu .....	477
Użycie atrybutu ModelBinder .....	478
Podsumowanie .....	478
<b>Rozdział 18. Kontrola poprawności modelu .....</b>	<b>479</b>
Tworzenie projektu .....	479
Jawna kontrola poprawności modelu .....	481
Wyświetlanie komunikatów kontroli poprawności .....	482
Wyświetlanie komunikatów kontroli poprawności poziomu właściwości .....	486
Użycie alternatywnych technik kontroli poprawności .....	487
Kontrola poprawności w łączniku modelu .....	488
Definiowanie zasad poprawności za pomocą metadanych .....	491
Definiowanie modeli samokontrolujących się .....	494
Tworzenie własnego dostawcy kontroli poprawności .....	495
Użycie kontroli poprawności po stronie klienta .....	499
Aktywowanie i wyłączanie kontroli poprawności po stronie klienta .....	499
Użycie kontroli poprawności po stronie klienta .....	502
Jak działa kontrola poprawności po stronie klienta? .....	504
Modyfikowanie kontroli poprawności na kliencie .....	505
Wykonywanie zdalnej kontroli poprawności .....	511
Podsumowanie .....	513
<b>Rozdział 19. Nieprzeszkadzający Ajax .....</b>	<b>515</b>
Użycie nieprzeszkadzających wywołań Ajax w MVC .....	515
Tworzenie projektu .....	515
Włączanie i wyłączanie nieprzeszkadzających wywołań Ajax .....	518
Użycie nieprzeszkadzających formularzy Ajax .....	518
Sposób działania nieprzeszkadzających wywołań Ajax .....	520
Ustawianie opcji Ajax .....	520
Zapewnienie kontrolowanej degradacji .....	520
Informowanie użytkownika o realizowanym żądaniu Ajax .....	522
Wyświetlanie pytania przed wysłaniem żądania .....	523
Tworzenie łączy Ajax .....	524
Zapewnienie kontrolowanej degradacji dla łączy .....	526
Korzystanie z funkcji wywołania zwrotnego w Ajaksie .....	527
Wykorzystanie JSON .....	529
Dodanie obsługi JSON do kontrolera .....	530
Przetwarzanie JSON w przeglądarce .....	531
Wykrywanie żądań Ajax w metodach akcji .....	532
Odbieranie danych JSON .....	533
Podsumowanie .....	535

<b>Rozdział 20. jQuery</b> .....	<b>537</b>
Tworzenie projektu .....	537
Odwoływanie się do jQuery .....	539
Tworzenie kodu jQuery .....	541
Utworzenie środowiska testowego dla jQuery .....	542
Podstawy jQuery .....	544
Poznajemy selektory jQuery .....	545
Użycie filtrów jQuery .....	546
Poznajemy metody jQuery .....	548
Czekając na DOM .....	549
Użycie metod CSS z jQuery .....	550
Manipulowanie modelem DOM .....	552
Użycie zdarzeń jQuery .....	555
Użycie efektów wizualnych jQuery .....	556
Użycie jQuery UI .....	558
Odwoływanie się do jQuery UI .....	558
Tworzenie lepszych przycisków .....	559
Użycie kontrolki Slider .....	560
Podsumowanie .....	562
<b>Część III Tworzenie udanych projektów ASP.NET MVC 3</b> .....	<b>565</b>
<b>Rozdział 21. Bezpieczeństwo i słabe punkty</b> .....	<b>567</b>
Każde dane mogą być sfalszowane .....	567
Falszowanie żądań HTTP .....	569
Skrypty międzywitrynowe i wstrzykiwanie HTML .....	570
Przedstawiamy ataki XSS .....	570
Kodowanie HTML w Razor .....	571
Kontrola żądań .....	572
Kodowanie ciągów znaków JavaScript a XSS .....	574
Przejęcie sesji .....	576
Obrona przez sprawdzanie adresu IP klienta .....	576
Obrona przez ustawienie w cookie znacznika HttpOnly .....	577
Międzywitrynowe fałszowanie żądań .....	577
Atak .....	578
Obrona .....	578
Ochrona przed atakami CSRF z użyciem metod zapobiegających fałszerstwom .....	579
Wstrzykiwanie SQL .....	580
Atak .....	580
Obrona z użyciem zapytań parametrycznych .....	581
Obrona z użyciem odwzorowania obiektowo-relacyjnego .....	581
Bezpieczne korzystanie z biblioteki MVC .....	581
Nie udostępniaj przypadkowo metod akcji .....	581
Nie używaj dołączania modelu w celu zmiany wrażliwych właściwości .....	582
Podsumowanie .....	582
<b>Rozdział 22. Uwierzytelnianie i autoryzacja</b> .....	<b>583</b>
Użycie uwierzytelniania Windows .....	583
Użycie uwierzytelniania Forms .....	585
Konfiguracja uwierzytelniania Forms .....	586
Uwierzytelnianie Forms bez użycia cookies .....	587

Członkostwo, role i profile .....	588
Konfigurowanie i wykorzystanie członkostwa .....	589
Konfigurowanie i wykorzystanie ról .....	597
Konfigurowanie i wykorzystanie profili .....	600
Dlaczego nie należy korzystać z uwierzytelniania bazującego na adresach URL .....	604
Ograniczanie dostępu z użyciem adresów IP oraz domen .....	604
Podsumowanie .....	606
<b>Rozdział 23. Instalacja .....</b>	<b>607</b>
Przygotowanie aplikacji do dystrybucji .....	607
Wykrywanie błędów przed instalacją .....	607
Kontrolowanie dynamicznej kompilacji stron .....	608
Przygotowanie do instalacji binarnej .....	609
Przygotowanie pliku Web.config do przekształcania .....	609
Przygotowanie projektu do instalacji bazy danych .....	619
Podstawy działania serwera IIS .....	621
Witryny Web .....	621
Katalogi wirtualne .....	621
Pule aplikacji .....	621
Wiązanie witryn z nazwami hostów, adresami IP oraz portami .....	622
Przygotowanie serwera do instalacji .....	622
Instalowanie aplikacji .....	624
Instalowanie aplikacji przez kopiowanie plików .....	624
Użycie pakietu instalacyjnego .....	625
Publikowanie jednym kliknięciem .....	628
Podsumowanie .....	629
<b>Skorowidz .....</b>	<b>631</b>



# Kontrola poprawności modelu

W poprzednim rozdziale przedstawiliśmy, w jaki sposób na podstawie żądań HTTP biblioteka MVC tworzy obiekty modelu w procesie dołączania modelu. Zakładaliśmy, że dane wprowadzone przez użytkownika były prawidłowe. W rzeczywistości użytkownicy często wprowadzają dane, z których nie możemy skorzystać, i tym zajmujemy się teraz — *kontrolą poprawności danych modelu*.

Kontrola poprawności modelu jest procesem, dzięki któremu upewniamy się, że otrzymane dane nadają się do użycia w modelu, a jeżeli nie, dostarczamy użytkownikom informacje pomagające rozwiązać problem.

Pierwsza część procesu — sprawdzanie otrzymanych danych — jest jednym ze sposobów zapewnienia integralności danych w modelu domeny. Przez odrzucenie danych, które nie mają sensu w kontekście naszej domeny, zapobiegamy powstawaniu dziwnych i niechcianych stanów aplikacji. Druga część — pomoc użytkownikowi w rozwiązaniu problemu — jest równie ważna. Jeżeli nie zapewnimy użytkownikowi informacji i narzędzi potrzebnych do interakcji z aplikacją w oczekiwany przez nas sposób, szybko spowodujemy jego frustrację. W przypadku aplikacji dostępnych publicznie często powoduje to wycofywanie się użytkowników, natomiast w przypadku aplikacji korporacyjnych może to skutkować spowolnieniem ich działań. Każda z tych sytuacji jest niepożądana.

Na szczęście biblioteka MVC zapewnia rozbudowaną obsługę kontroli poprawności modelu. Pokażemy teraz, w jaki sposób korzystać z podstawowych funkcji, a następnie przejdziemy do bardziej zaawansowanych technik sterowania procesem kontroli poprawności.

## Tworzenie projektu

Potrzebna jest nam prosta aplikacja MVC, w której będziemy mogli stosować różne techniki kontroli poprawności modelu. Na jej potrzeby utworzyliśmy klasę modelu widoku o nazwie `Appointment`, zamieszczoną na listingu 18.1.

### *Listing 18.1. Klasa modelu widoku*

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

namespace MvcApp.Models {
    public class Appointment {

        public string ClientName { get; set; }
        [DataType(DataType.Date)]
        public DateTime Date {get; set;}
    }
}
```

```

        public bool TermsAccepted { get; set; }
    }
}

```

Na listingu 18.2 zamieszczony jest widok generujący edytory dla klasy Appointment, zapisany w pliku *MakeBooking.cshtml*.

**Listing 18.2.** Widok edytora

```

@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)</p>
    <p>Data wizyty: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki</p>

    <input type="submit" value="Wyślij rezerwację" />
}

```

Na listingu 18.3 zamieszczona jest klasa kontrolera, AppointmentController, posiadająca metody akcji operujące na obiektach Appointment.

**Listing 18.3.** Klasa AppointmentController

```

public class AppointmentController : Controller {
    private IAppointmentRepository repository;

    public AppointmentController(IAppointmentRepository repo) {
        repository = repo;
    }

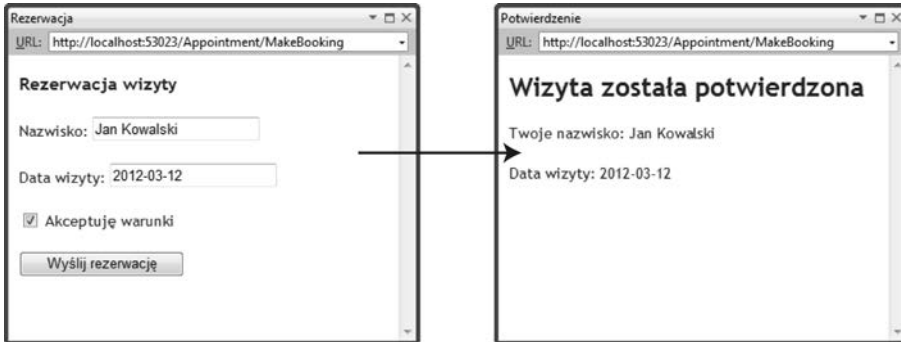
    public ActionResult MakeBooking() {
        return View(new Appointment { Date = DateTime.Now });
    }

    [HttpPost]
    public ActionResult MakeBooking(Appointment appt) {

        repository.SaveAppointment(appt);
        return View("Completed", appt);
    }
}

```

Kontroler realizuje znany już nam wzorzec. Metoda akcji MakeBooking generuje widok *MakeBooking.cshtml*. Widok ten zawiera formularz, którego dane są przesyłane do drugiej wersji metody MakeBooking, oczekującej parametru Appointment. Łącznik modelu tworzy obiekt Appointment na podstawie wartości elementów formularza HTML, a następnie przekazuje go do metody akcji, zapisującej nowy obiekt w repozytorium, które zostanie dostarczone poprzez wstrzykiwanie zależności (utworzyliśmy tylko szkielet repozytorium — dane wizyty są ignorowane, ponieważ chcemy się tu skupić wyłącznie na kontroli poprawności). Po przekazaniu wizyty do repozytorium kontroler generuje widok *Completed.cshtml*, który pozwala przekazać informację zwrotną dla użytkownika. Na rysunku 18.1 przedstawiona jest ta sekwencja widoków.



**Rysunek 18.1.** Kolejność widoków w przykładowej aplikacji

Obecnie nasza aplikacja będzie akceptowała dowolne przesłane przez użytkownika dane, ale aby zachować integralność aplikacji i modelu domeny, wymagamy spełnienia trzech warunków przed zaakceptowaniem przesłanych przez użytkownika danych wizyty:

- użytkownik musi podać nazwisko,
- użytkownik musi podać przyszłą datę (w formacie mm-dd-rrrr),
- użytkownik musi zaznaczyć pole wyboru informujące o zaakceptowaniu zasad.

Kontrola poprawności modelu jest procesem umożliwiającym wymuszenie tych wymagań. W kolejnych punktach pokażemy różne techniki pozwalające weryfikować dane przesłane przez użytkownika oraz przekazanie użytkownikowi informacji, dlaczego nie możemy wykorzystać dostarczonych danych.

## Jawna kontrola poprawności modelu

Najbardziej bezpośrednim sposobem kontroli poprawności modelu jest wykonanie tej operacji w metodzie akcji. Realizacja tego zadania jest zamieszczona na listingu 18.4.

**Listing 18.4.** Jawna kontrola poprawności modelu

```
[HttpPost]
public ActionResult MakeBooking(Appointment appt) {

    if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date) {
        ModelState.AddModelError("Date", "Proszę podać przyszłą datę.");
    }

    if (!appt.TermsAccepted) {
        ModelState.AddModelError("TermsAccepted", "Zaakceptowanie zasad jest wymagane.");
    }

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}
```

Sprawdzamy tu wartości przypisane przez łącznik modelu do właściwości obiektu parametru i rejestrujemy wszystkie znalezione błędy we właściwości `ModelState`, którą kontroler dziedziczy po swojej klasie bazowej. Dla przykładu przeanalizujemy, w jaki sposób sprawdzamy właściwość `ClientName`:

```
if (string.IsNullOrEmpty(appt.ClientName)) {
    ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
}
```

Chcemy, aby użytkownik podał wartość tej właściwości, więc do jej sprawdzenia wykorzystujemy statyczną metodę `string.IsNullOrEmpty`. Jeżeli nie otrzymamy wartości, wywołujemy metodę `ModelState.AddModelError`, podając nazwę właściwości, której dotyczy błąd (`ClientName`), oraz komunikat, jaki powinien być wyświetlony użytkownikowi, aby mu pomóc przy rozwiązaniu problemu (*Proszę podać swoje nazwisko.*).

Za pomocą właściwości `ModelState.IsValidField` jesteśmy w stanie sprawdzić, czy łącznik modelu był w stanie przypisać wartość do właściwości. Wykonaliśmy to dla właściwości `Date`, aby upewnić się, czy łącznik modelu skonwertował przekazaną przez użytkownika wartość; jeżeli nie, nie ma sensu wykonywać dodatkowych sprawdzeń i raportować kolejnych błędów.

Po sprawdzeniu wszystkich właściwości w obiekcie modelu odczytujemy właściwość `ModelState.IsValid` w celu sprawdzenia, czy wystąpiły błędy. Właściwość ta ma wartość `false`, jeżeli zarejestrowane zostały jakiegokolwiek błędy lub jeżeli łącznik modelu napotkał problemy:

```
if (ModelState.IsValid) {
    repository.SaveAppointment(appt);
    return View("Completed", appt);
} else {
    return View();
}
```

Jeżeli nie wystąpiły błędy, zapisujemy wizytę za pomocą repozytorium i generujemy widok *Completed*. Jeżeli pojawiły się problemy, wywołujemy po prostu metodę `View` bez parametrów. Powoduje to ponowne wygenerowanie bieżącego widoku, dzięki czemu użytkownik zobaczy informacje o błędach i będzie mógł skorygować wprowadzane dane.

Metody pomocnicze widoku szablonowego używane do wygenerowania edytorów dla właściwości modelu obsługują błędy kontroli poprawności. Jeżeli dla właściwości zostanie zareportowany błąd, metoda pomocnicza doda do pola tekstowego klasę CSS o nazwie `input-validation-error`. Plik `~/Content/Site.css` zawiera następującą domyślną definicję tego stylu:

```
.input-validation-error {
    border: 1px solid #ff0000;
    background-color: #ffebee;
}
```

Powoduje ona ustawienie czerwonej ramki oraz różowego tła w elemencie, w którym wystąpił błąd. Na rysunku 18.2 przedstawiony jest ten efekt dla wszystkich trzech właściwości.

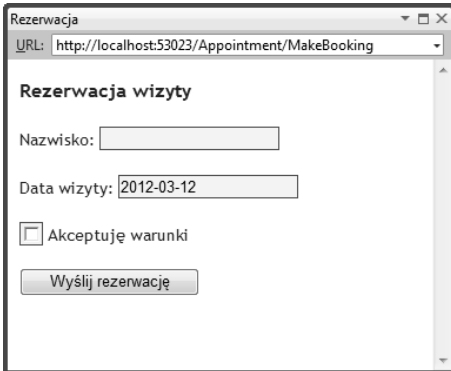
## Wyświetlanie komunikatów kontroli poprawności

Style CSS dołączane przez metody widoku szablonowego do elementów tekstowych informują o wystąpieniu problemu w polu, ale nie pozwalają przekazać mu danych o samym problemie. Na szczęście dostępne są inne metody pomocnicze HTML wspierające nas w tym zadaniu. Na listingu 18.5 pokazane jest użycie jednej z tych metod w widoku `MakeBooking`.

### Listing 18.5. Użycie metody pomocniczej podsumowania

```
@model MvcApp.Models.Appointment
@{
    ViewBag.Title = "Rezerwacja";
```





Rysunek 18.2. Błędy powodują wyróżnienie elementów

## Nadawanie stylu polom wyboru

Niektóre przeglądarki, w tym Chrome i Firefox, ignorują style nadawane polom tekstowym, co prowadzi do powstania niespójnego wyglądu strony. Rozwiązaniem jest zastąpienie szablonu edytora Boolean własnym, umieszczonym w `~/Views/Shared/EditorTemplates/Boolean.cshtml`, w którym pole wyboru jest osadzone w elemencie `div`. Poniżej jest zamieszczony używany przez nas szablon, ale możesz go dostosować do swoich aplikacji:

```
@model bool?
```

```
@if (ViewData.ModelMetadata.IsNullableValueType) {
    @Html.DropDownListFor(m => m, new SelectList(new [] { "Nie ustawiono", "Prawda", "Fałsz" },
    ViewData.ModelMetadata.PropertyName, ViewData.Model))
} else {
    ModelState state = ViewData.ModelState[ViewData.ModelMetadata.PropertyName];
    bool value = Model ?? false;

    if (state != null && state.Errors.Count > 0) {
        <div class="input-validation-error" style="float:left">
            @Html.CheckBox("", value)
        </div>
    } else {
        @Html.CheckBox("", value)
    }
}
```

W szablonie tym umieściliśmy pole wyboru w elemencie `div`, do którego dołączany jest styl `input-validation-error`, jeżeli zostały zarejestrowane błędy modelu skojarzone z właściwością obsługiwaną przez szablon. Więcej informacji na temat wymiany szablonów edytorów znajduje się w rozdziale 16.

```
}
```

```
<h4>Rezerwacja wizyty</h4>
```

```
@using (Html.BeginForm()) {
```

```
    @Html.ValidationSummary()
```

```
    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)</p>
```

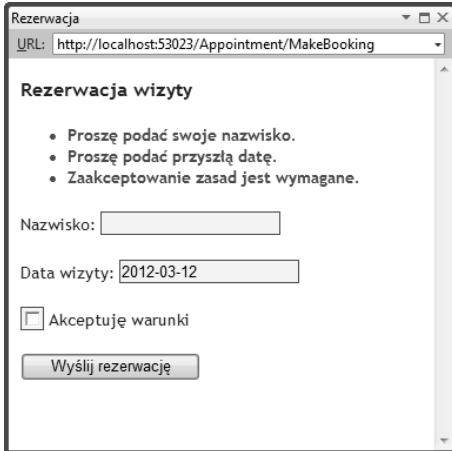
```

<p>Data wizyty: @Html.EditorFor(m => m.Date)</p>
<p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki</p>

<input type="submit" value="Wyślij rezerwację" />
}

```

Metoda pomocnicza `Html.ValidationSummary` pozwala wyświetlić podsumowanie błędów kontroli poprawności zarejestrowanych dla strony. Jeżeli nie ma błędów, metoda pomocnicza nie generuje żadnego kodu HTML. Na rysunku 18.3 przedstawione jest działanie pola podsumowania.



**Rysunek 18.3.** Wyświetlanie podsumowania kontroli poprawności

Pole podsumowania wyświetla komunikaty zarejestrowane przez metodę akcji w obiekcie `ModelState`. Na listingu 18.6 zamieszczony jest kod HTML wygenerowany przez tę metodę.

**Listing 18.6.** Kod HTML wygenerowany przez metodę pomocniczą `ValidationSummary`

```

<div class="validation-summary-errors" data-valmsg-summary="true">
  <ul>
    <li>Proszę podać swoje nazwisko.</li>
    <li>Proszę podać przyszłą datę.</li>
    <li>Zaakceptowanie zasad jest wymagane.</li>
  </ul>
</div>

```

Informacja o błędach są wyrażane jako lista umieszczona w elemencie `div`, do którego jest dołączona klasa CSS `validation-summary-errors`. Styl jest zdefiniowany w pliku `~/Content/Site.css` i w razie potrzeby może być zmieniony. Domyślny styl powoduje użycie pogrubionego, czerwonego tekstu:

```

.validation-summary-errors {
  font-weight: bold;
  color: #ff0000;
}

```

Metoda `ValidationSummary` posiada kilka przeciążonych wersji; najprzydatniejsze są zamieszczone w tabeli 18.1.

Niektóre z przeciążonych wersji metody `ValidationSummary` pozwalają nam na wskazanie, że powinny być wyświetlone wyłącznie *komunikaty o błędach na poziomie modelu*. Błędy, jakie rejestrowaliśmy do tej pory w `ModelState`, były *błędami na poziomie właściwości*, co oznacza, że wystąpił problem z wartością podaną w danej właściwości i jej zmiana może rozwiązać problem.

**Tabela 18.1.** Najprzydatniejsze przeciążone wersje metody `ValidationSummary`

Przeciążona metoda	Opis
<code>Html.ValidationSummary()</code>	Generuje podsumowanie dla wszystkich błędów.
<code>Html.ValidationSummary(bool)</code>	Jeżeli parametr <code>bool</code> ma wartość <code>true</code> , to wyświetlane są tylko komunikaty o błędach poziomu modelu (patrz wyjaśnienie pod tabelą). Jeżeli parametr ma wartość <code>false</code> , wyświetlane są komunikaty o wszystkich błędach.
<code>Html.ValidationSummary(string)</code>	Wyświetla komunikat (przekazany w parametrze typu <code>string</code> ) przed podsumowaniem wszystkich błędów.
<code>Html.ValidationSummary(bool, string)</code>	Wyświetla komunikat przed błędami. Jeżeli parametr <code>bool</code> ma wartość <code>true</code> , wyświetlane są wyłącznie komunikaty o błędach poziomu modelu.

Z kolei błędy na poziomie modelu mogą być wykorzystywane, jeżeli powstaje problem z interakcją pomiędzy dwoma właściwościami (lub większą ich liczbą). Wyobraźmy sobie, że klient Jan nie może składać rezerwacji w poniedziałki. Na listingu 18.7 pokazany jest sposób wymuszenia tej zasady i raportowania problemów za pomocą błędów kontroli poprawności na poziomie modelu.

**Listing 18.7.** Błąd kontroli poprawności na poziomie modelu

```
public IActionResult MakeBooking(Appointment appt) {
    if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date) {
        ModelState.AddModelError("Date", "Proszę podać przyszłą datę.");
    }

    if (!appt.TermsAccepted) {
        ModelState.AddModelError("TermsAccepted", "Zaakceptowanie zasad jest wymagane.");
    }

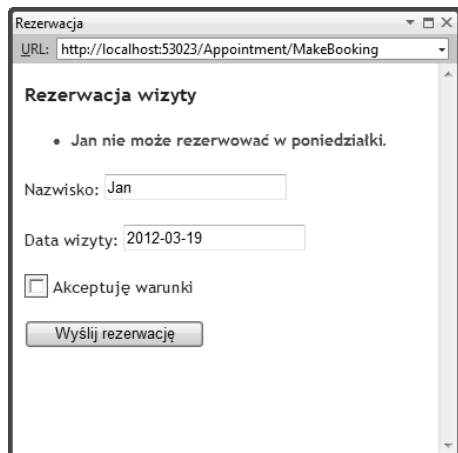
    if (ModelState.IsValidField("ClientName") && ModelState.IsValidField("Date")
        && appt.ClientName == "Jan" && appt.Date.DayOfWeek == DayOfWeek.Monday) {
        ModelState.AddModelError("", "Jan nie może rezerwować w poniedziałki.");
    }

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}
```

Zanim sprawdzimy, czy Jan próbuje zarezerwować wizytę na poniedziałek, korzystamy z metody `ModelState.IsValidField` w celu upewnienia się, że mamy prawidłowe wartości pól `ClientName` oraz `Date`. Oznacza to, że nie możemy wygenerować błędu poziomu modelu do momentu spełnienia warunków dla właściwości. Błędy poziomu modelu rejestrujemy, podając pusty ciąg znaków ("") jako pierwszy parametr metody `ModelState.AddModelError`:

```
ModelState.AddModelError("", "Jan nie może rezerwować w poniedziałki.");
```

Użycie metody `ValidationSummary` z parametrem `bool` pozwala nam wyświetlić informacje wyłącznie o błędach na poziomie modelu, co jest pokazane na rysunku 18.4.



**Rysunek 18.4.** Wyświetlanie wyłącznie informacji o błędach na poziomie modelu

Jak można zauważyć na rysunku, mamy tu dwa błędy kontroli poprawności. Pierwszym jest błąd na poziomie modelu, który wystąpił, gdy Jan próbował zarezerwować wizytę na poniedziałek. Drugi wynika z braku zaznaczenia pola akceptacji zasad. Ponieważ w podsumowaniu wyświetlamy wyłącznie komunikaty o błędach poziomu modelu, użytkownik nie zobaczy żadnej informacji o braku zaznaczenia opcji.

## Wyświetlanie komunikatów kontroli poprawności poziomu właściwości

Powodem ograniczania komunikatów o błędach wyświetlanych w podsumowaniu do błędów poziomu modelu jest możliwość wyświetlania komunikatów o błędach poziomu właściwości obok pól. W takim przypadku nie chcemy powielać komunikatów z poziomu właściwości. Na listingu 18.8 zamieszczony jest zaktualizowany widok `MakeBooking`, w którym komunikaty o błędach poziomu modelu są wyświetlane w podsumowaniu, a o błędach na poziomie właściwości — obok odpowiedniego pola.

Metoda pomocnicza `Html.ValidationMessageFor` wyświetla komunikaty o błędach dla poszczególnych właściwości modelu. Na rysunku 18.5 przedstawiony jest wynik działania tej metody.

**Listing 18.8.** Użycie komunikatów o błędach poziomu właściwości

```
@model MvcApp.Models.Appointment
@{
    ViewBag.Title = "Rezerwacja";
}
<h4>Rezerwacja wizyty</h4>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <p>
        Nazwisko: @Html.EditorFor(m => m.ClientName)
```

```

    @Html.ValidationMessageFor(m => m.ClientName)
</p>
<p>
    Data wizyty: @Html.EditorFor(m => m.Date)
    @Html.ValidationMessageFor(m => m.Date)
</p>
<p>
    @Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki
    @Html.ValidationMessageFor(m => m.TermsAccepted)
</p>

<input type="submit" value="Wyślij rezerwację" />
}

```

The screenshot shows a web browser window titled "Rezerwacja" with the URL "http://localhost:53023/Appointment/MakeBooking". The page content is as follows:

**Rezerwacja wizyty**

Nazwisko:  Proszę podać swoje nazwisko.

Data wizyty:  Proszę podać przyszłą datę.

Akceptuję warunki Zaakceptowanie zasad jest wymagane.

**Rysunek 18.5.** Użycie komunikatów kontroli poprawności dla pojedynczych właściwości

Oczywiście nie ma sensu wyświetlanie informacji o błędach z poziomu modelu obok określonej właściwości, więc umieścimy je w osobnej sekcji, wygenerowanej za pomocą metody pomocniczej `Html.ValidationSummary`, jak pokazano na rysunku 18.6.

The screenshot shows the same web browser window as in Figure 18.5. The page content is as follows:

**Rezerwacja wizyty**

- Jan nie może rezerwować w poniedziałki.

Nazwisko:

Data wizyty:

Akceptuję warunki Zaakceptowanie zasad jest wymagane.

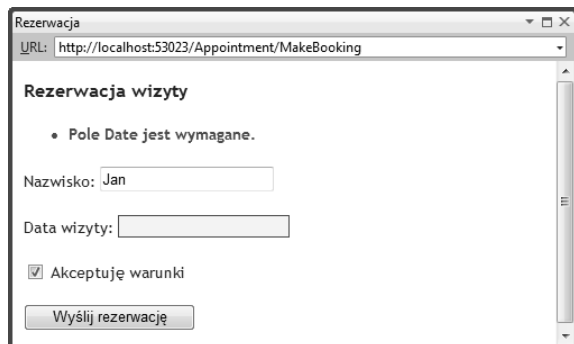
**Rysunek 18.6.** Wyświetlanie komunikatów o błędach modelu oraz właściwości

## Użycie alternatywnych technik kontroli poprawności

Wykonywanie kontroli poprawności w metodzie akcji jest tylko jedną z technik dostępnych w bibliotece MVC. W kolejnych punktach przedstawimy inne podejścia do tego zagadnienia.

## Kontrola poprawności w łączniku modelu

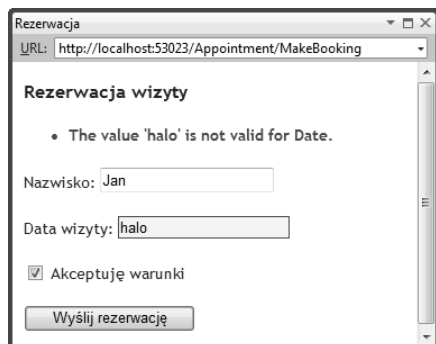
Domyślny łącznik modelu wykonuje kontrolę poprawności w ramach procesu dołączania. Na rysunku 18.7 pokazane jest, co się stanie, gdy wyczyszcimy pole Date i wyślemy dane formularza.



**Rysunek 18.7.** Komunikat kontroli poprawności z łącznika modelu

Zauważ, że dla pola Date wyświetlany jest komunikat o błędzie (wróciliśmy tu do wyświetlania wszystkich komunikatów o błędach w podsumowaniu). Komunikat ten został dodany przez łącznik modelu, ponieważ nie był on w stanie utworzyć obiektu DateTime z pustego pola formularza.

Łącznik modelu realizuje podstawową kontrolę poprawności dla każdej właściwości w obiekcie modelu. Jeżeli wartość nie zostanie dostarczona, będzie wygenerowany komunikat pokazany na rysunku 18.7. Jeżeli podamy wartość, której nie da się przekształcić na typ właściwości modelu, wyświetlony zostanie inny komunikat (rysunek 18.8).



**Rysunek 18.8.** Błąd kontroli poprawności formatu wyświetlany przez łącznik modelu

Wbudowana klasa łącznika modelu, `DefaultModelBinder`, posiada kilka użytecznych metod, które możemy nadpisać w celu dodania kontroli poprawności do łącznika. Metody te są opisane w tabeli 18.2.

Możemy nadpisać metody zamieszczone w tabeli 18.2, dodając w ten sposób logikę kontroli poprawności do łącznika. Przykład jest zamieszczony na listingu 18.9. Ilustruje on jedynie możliwość użycia tego rozwiązania, ale zalecamy korzystanie z dostawców kontroli poprawności przedstawionych w dalszej części rozdziału.

Kontrola poprawności w łączniku modelu wygląda na bardziej skomplikowaną, niż jest w rzeczywistości. Logika kontroli poprawności jest taka sama jak w metodzie akcji. Kontrolę poprawności na poziomie właściwości realizujemy w metodzie `SetProperty`. Metoda ta jest wywoływana dla każdej właściwości modelu. Z punktu widzenia procesu dołączania obiektu modelu nie został w pełni utworzony, więc przy kontroli poprawności korzystamy z parametrów metody. Odczytujemy nazwę właściwości z parametru `PropertyDescriptor`, przypisaną wartość z parametru `object`, a dostęp do `ModelState` mamy poprzez parametr `BindingContext`.

**Tabela 18.2.** Metody klasy *DefaultModelBinder* pozwalające na dodanie kontroli poprawności do procesu dołączania modelu

Metoda	Opis	Domyślna implementacja
OnModelUpdated	Wywoływana w momencie, gdy łącznik próbuje przypisać wartości do wszystkich właściwości obiektu modelu.	Stosuje zasady poprawności zdefiniowane przez metadane modelu oraz rejestruje wszystkie błędy w ModelState. Użycie metadanych do kontroli poprawności zostanie opisane w dalszej części rozdziału.
SetProperty	Wywoływana, gdy łącznik chce przypisać wartość do określonej właściwości.	Jeżeli właściwość nie może zawierać wartości null, a nie ma wartości do przypisania, w ModelState rejestrowany jest błąd <i>The &lt;nazwa&gt; field is required</i> (rysunek 18.7). Jeżeli istnieje wartość, która nie może być skonwertowana, to rejestrowany jest błąd <i>The value &lt;wartość&gt; is not valid for &lt;nazwa&gt;</i> (rysunek 18.8).

**Listing 18.9.** Dziedziczenie po klasie *DefaultModelBinder* w celu dodania kontroli poprawności do procesu dołączania

```
using System;
using System.ComponentModel;
using System.Web.Mvc;
using MvcApp.Models;
namespace MvcApp.Infrastructure {
    public class ValidatingModelBinder : DefaultModelBinder {

        protected override void SetProperty(ControllerContext controllerContext,
            ModelBindingContext bindingContext, PropertyDescriptor propertyDescriptor,
            object value) {

            // upewnij się, że zostanie wywołana implementacja bazowa
            base.SetProperty(controllerContext, bindingContext, propertyDescriptor, value);

            // wykonanie naszej kontroli poprawności na poziomie właściwości
            switch (propertyDescriptor.Name) {
                case "ClientName":
                    if (string.IsNullOrEmpty((string)value)) {
                        bindingContext.ModelState.AddModelError("ClientName",
                            "Proszę podać nazwisko.");
                    }
                    break;
                case "Date":
                    if (bindingContext.ModelState.IsValidField("Date") &&
                        DateTime.Now > ((DateTime)value)) {
                        bindingContext.ModelState.AddModelError("Date",
                            "Proszę podać przyszłą datę.");
                    }
                    break;
                case "TermsAccepted":
                    if (!((bool)value)) {
                        bindingContext.ModelState.AddModelError("TermsAccepted",
                            "Zaakceptowanie warunków jest obowiązkowe.");
                    }
                    break;
            }
        }

        protected override void OnModelUpdated(ControllerContext controllerContext,
            ModelBindingContext bindingContext) {
```

```

// upewnij się, że zostanie wywołana implementacja bazowa
base.OnModelUpdated(controllerContext, bindingContext);

// pobierz model
Appointment model = bindingContext.Model as Appointment;
// wykonaj kontrolę poprawności na poziomie modelu
if (model != null &&
    bindingContext.ModelState.IsValidField("ClientName") &&
    bindingContext.ModelState.IsValidField("Date") &&
    model.ClientName == "Jan" &&
    model.Date.DayOfWeek == DayOfWeek.Monday) {
    bindingContext.ModelState.AddModelError("",
        "Jan nie może rezerwować w poniedziałki.");
}
}
}
}

```

Kontrolę poprawności na poziomie modelu realizujemy w metodzie `OnModelUpdate`. W tym momencie procesu dołączania wartości są przypisane do obiektu modelu, więc możemy odczytywać właściwości w celu wykonania kontroli.

- 
- **Ostrzeżenie** Przy korzystaniu z łącznika do realizacji kontroli poprawności jest ważne, aby wywołać bazowe implementacje metod `SetProperty` oraz `OnModelUpdated`. W przeciwnym razie stracimy część ważnych funkcji, takich jak kontrola poprawności modelu z użyciem metadanych (przedstawiona w dalszej części rozdziału).
- 

Kolejnym krokiem jest zarejestrowanie naszego łącznika w metodzie `Application_Start` znajdującej się w pliku `Global.asax`:

```

protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    ModelBinders.Binders.Add(typeof(Appointment), new ValidatingModelBinder());

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}

```

Ponieważ przenieśliśmy logikę kontroli poprawności klasy modelu `Appointment` do łącznika, możemy uprościć metodę akcji do postaci pokazanej na listingu 18.10.

#### **Listing 18.10.** Uproszczona metoda akcji

```

[HttpPost]
public ActionResult MakeBooking(Appointment appt) {

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}

```



Wszystkie błędy zarejestrowane w łączniku będą dostępne w metodzie akcji, w kolekcji ModelState. Dzięki temu możemy sprawdzić właściwość ModelState.IsValid, która będzie miała wartość false w przypadku, gdy łącznik rejestruje błędy kontroli poprawności. Jeżeli nie występowały błędy, zapisujemy obiekt Appointment za pomocą repozytorium i generujemy widok Completed. Jeżeli wystąpiły błędy walidacji, generujemy bieżący widok i wyświetlamy komunikaty o błędach.

## Definiowanie zasad poprawności za pomocą metadanych

Nie musimy definiować logiki kontroli poprawności w sposób programowy. Biblioteka MVC pozwala na użycie metadanych do definiowania zasad poprawności modelu. Zaletą użycia metadanych jest wymuszenie zasad kontroli poprawności w każdym procesie dołączania danych do klasy modelu — w przeciwieństwie do wykorzystywania pojedynczych metod akcji. Atrybuty kontroli poprawności są wykrywane i wymuszane przez wbudowaną klasę łącznika modelu, DefaultModelBinder. Atrybuty te są stosowane w klasie modelu w sposób pokazany na listingu 18.11.

### Listing 18.11. Użycie atrybutów do definiowania zasad poprawności

```
using System;
using System.ComponentModel.DataAnnotations;
using MvcApp.Infrastructure;

namespace MvcApp.Models {

    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [Required(ErrorMessage="Proszę podać datę")]
        public DateTime Date { get; set; }
        [Range(typeof(bool), "true", "true", ErrorMessage="Zaakceptowanie warunków jest
        ↪obowiązkowe")]
        public bool TermsAccepted { get; set; }
    }
}
```

Użyliśmy tu dwóch atrybutów kontroli poprawności — Required oraz Range. Atrybut Required powoduje powstanie błędu kontroli poprawności, jeżeli użytkownik nie poda wartości dla właściwości. Atrybut Range pozwala określić akceptowalny zakres wartości. W tabeli 18.3 zamieszczone są wbudowane atrybuty kontroli poprawności.

Wszystkie atrybuty kontroli poprawności pozwalają nam określić własny komunikat o błędzie przez ustawienie właściwości ErrorMessage, tak jak w poniższym przykładzie:

```
[Required(ErrorMessage="Proszę podać datę.")]
```

Jeżeli nie podamy własnego komunikatu o błędzie, to będzie użyty domyślny, jak pokazano na rysunkach 18.7 i 18.8. Atrybuty kontroli poprawności są dosyć proste i pozwalają wyłącznie na kontrolę poprawności na poziomie właściwości. Mimo to mamy sporo narzędzi, aby zapewnić spójne działanie tego mechanizmu. Weźmy pod uwagę atrybut zastosowany dla właściwości TermsAccepted.

```
[Range(typeof(bool), "true", "true", ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
public bool TermsAccepted { get; set; }
```

Chcemy upewnić się, że użytkownik zaznaczy pole wyboru zaakceptowania warunków. Nie możemy użyć atrybutu Required, ponieważ metody pomocnicze widoku szablonowego generują ukryte pole HTML w celu upewnienia się, że otrzymamy wartość nawet w przypadku, gdy pole nie jest zaznaczone. Aby ominąć ten problem,

**Tabela 18.3.** Wbudowane atrybuty kontroli poprawności

Atrybut	Przykład	Opis
Compare	[Compare("MyOtherProperty")]	Dwie właściwości muszą mieć taką samą wartość. Jest to przydatne, jeżeli prosimy użytkownika o dwukrotne podanie tej samej danej, na przykład adresu e-mail lub hasła.
Range	[Range(10, 20)]	Wartość numeryczna (lub właściwość typu implementującego IComparable) musi mieć wartość pomiędzy podanym minimum i maksimum. Aby zdefiniować granicę tylko z jednej strony, należy użyć stałych MinValue lub MaxValue — na przykład [Range(int.MinValue, 50)].
RegularExpression	[RegularExpression("wzorzec")]	Wartość znakowa musi pasować do zdefiniowanego wyrażenia regularnego. Zwróć uwagę, że wzorzec musi pasować do całego wyrażenia podanego przez użytkownika, a nie tylko do jego fragmentu. Domyślnie dopasowania rozpoznają wielkie i małe litery, ale można to zmienić, dodając modyfikator (?i) — czyli [RegularExpression("(?i)wzorzec")].
Required	[Required]	Wartość musi być niepusta lub być ciągiem znaków zawierającym tylko spacje. Jeżeli chcesz traktować białe znaki jako prawidłową zawartość, użyj [Required(AllowEmptyStrings = true)].
StringLength	[StringLength(10)]	Wartość znakowa musi być nie dłuższa niż podana wartość maksymalna. Możemy również określić minimalną długość: [StringLength(10, MinimumLength=2)].

użyliśmy atrybutu Range, który pozwala określić typ oraz górną i dolną granicę w postaci wartości znakowych. Przez ustawienie obu granic na true utworzyliśmy odpowiednik atrybutu Required dla właściwości bool edytowanej za pomocą pola wyboru.

- **Wskazówka** Atrybut `DataType` nie może być używany do kontroli poprawności danych użytkownika — stanowi wyłącznie podpowiedź przy emitowaniu wartości z zastosowaniem metod pomocniczych widoku szablonowego (opisanych w rozdziale 16.). Dlatego nie należy oczekiwać, że atrybut `DataType(DataType.EmailAddress)` pozwoli wymusić odpowiedni format.

## Tworzenie własnego atrybutu kontroli poprawności

Sztuczka z użyciem atrybutu Range do odtworzenia działania atrybutu Required jest nieco dziwna. Na szczęście nie jesteśmy ograniczeni wyłącznie do wbudowanych atrybutów; można również tworzyć własne przez odziedziczenie po klasie `ValidationAttribute` i implementację własnej logiki kontroli poprawności. Przykład takiej klasy jest zamieszczony na listingu 18.12.

**Listing 18.12.** Tworzenie własnego atrybutu kontroli poprawności

```
public class MustBeTrueAttribute : ValidationAttribute {
    public override bool IsValid(object value) {
```

```

        return value is bool && (bool)value;
    }
}

```

Nasz nowy atrybut zmienia metodę `IsValid` z klasy bazowej. Jest to metoda wywoływana przez łącznik, do której przekazywana jest wartość wprowadzona przez użytkownika. W tym przykładzie nasza logika kontroli poprawności jest prosta — wartość jest prawidłowa, jeżeli jest typu `bool` o wartości `true`. Aby poinformować, że wartość jest prawidłowa, zwracamy `true` z metody `IsValid`. Atrybutu tego możemy użyć w poniższy sposób:

```

[MustBeTrue(ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
public bool TermsAccepted { get; set; }

```

Jest to przyjemniejsze i łatwiejsze niż nadużywanie atrybutu `Range`. Możemy również dziedziczyć po wbudowanych atrybutach kontroli poprawności w celu rozszerzania zakresu ich funkcji. Na listingu 18.13 pokazujemy, jak wykorzystać atrybut `Required` do upewnienia się, że została podana data spełniająca nasze wymagania.

**Listing 18.13.** Dziedziczenie po wbudowanym atrybucie kontroli poprawności

```

public class FutureDateAttribute : RequiredAttribute {

    public override bool IsValid(object value) {
        return base.IsValid(value) &&
            value is DateTime &&
                ((DateTime)value) > DateTime.Now;
    }
}

```

Oprócz funkcji zawartej w klasie bazowej dodaliśmy własną logikę kontroli poprawności. Atrybutu tego używamy identycznie jak poprzedniego:

```

[DataType(DataType.Date)]
[FutureDate(ErrorMessage="Proszę podać przyszłą datę.")]
public DateTime Date { get; set; }

```

## Tworzenie własnego atrybutu kontroli poprawności modelu

Przedstawione do tej pory atrybuty poprawności odnoszą się do poszczególnych właściwości modelu, co oznacza, że możemy generować tylko błędy kontroli poprawności na poziomie właściwości. Możemy również użyć metadanych do kontroli poprawności całego modelu w sposób pokazany na listingu 18.14.

**Listing 18.14.** Atrybuty kontroli poprawności modelu

```

public class AppointmentValidatorAttribute : ValidationAttribute {

    public AppointmentValidatorAttribute() {
        ErrorMessage = "Jan nie może rezerwować w poniedziałki.";
    }

    public override bool IsValid(object value) {

        Appointment app = value as Appointment;

        if (app == null || string.IsNullOrEmpty(app.ClientName) || app.Date == null) {
            // nie mamy modelu właściwego typu lub nie mamy
            // wartości wymaganych właściwości ClientName oraz Date
            return true;
        } else {
            return !(app.ClientName == "Jan" && app.Date.DayOfWeek == DayOfWeek.Monday);
        }
    }
}

```

```

    }
}

```

Parametr `object` przekazany przez łącznik modelu do metody `IsValid` powinien być typu `Appointment`. Nasz atrybut kontroli poprawności modelu dołączamy do samej klasy modelu:

**[AppointmentValidator]**

```

public class Appointment {

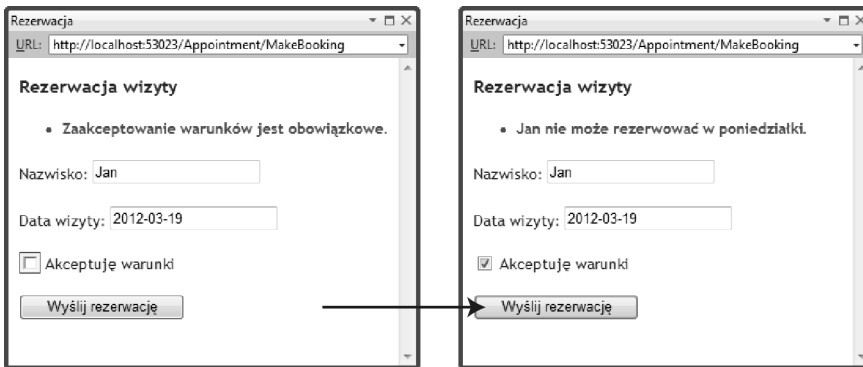
    [Required]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [FutureDate(ErrorMessage="Proszę podać przyszłą datę.")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
    public bool TermsAccepted { get; set; }
}

```

Nasz atrybut kontroli poprawności modelu nie będzie używany, jeżeli jakkolwiek z atrybutów poziomu właściwości zarejestrował błąd poprawności. Nie jest to dokładnie ten sam efekt, jaki osiągnęliśmy, umieszczając kod kontroli poprawności w metodzie akcji. Powoduje to ryzyko wydłużania procesu poprawiania błędów przez użytkownika. Jeżeli użytkownik na przykład poda wartości dla właściwości `ClientName` oraz `Date`, ale nie zaznaczy pola dla warunków, to atrybut `MustBeTrue` spowoduje wyświetlenie komunikatu o błędzie poprawności; jest on pokazany w lewej części rysunku 18.9.



**Rysunek 18.9.** Stopniowe zgłaszanie problemów z wartościami wejściowymi

Użytkownik poprawia błąd i ponownie wysyła dane — po czym otrzymuje inny komunikat, pokazany w prawej części rysunku 18.9. Z perspektywy użytkownika niejawnie akceptujemy wartość dla nazwiska i daty, nie oznaczając ich jako błędnych w pierwszym panelu. Może się to wydawać niewielkim problemem, ale warto przeanalizować każdą sytuację, która może frustrować użytkowników.

## Definiowanie modeli samokontrolujących się

Inną techniką kontroli poprawności jest utworzenie modeli samokontrolujących się, w których kod weryfikacji poprawności jest częścią klasy modelu. Samokontrolujące się klasy modelu tworzymy przez zaimplementowanie interfejsu `IValidatableObject` w sposób pokazany na listingu 18.15.

**Listing 18.15.** *Samokontrolująca się klasa modelu*

```

public class Appointment : IValidatableObject {

    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext) {

        List<ValidationResult> errors = new List<ValidationResult>();

        if (string.IsNullOrEmpty(ClientName)) {
            errors.Add(new ValidationResult("Proszę podać nazwisko."));
        }

        if (DateTime.Now > Date) {
            errors.Add(new ValidationResult("Proszę podać przyszłą datę."));
        }

        if (errors.Count == 0 && ClientName == "Jan"
            && Date.DayOfWeek == DayOfWeek.Monday) {

            errors.Add(new ValidationResult("Jan nie może rezerwować w poniedziałki."));
        }

        if (!TermsAccepted) {
            errors.Add(new ValidationResult("Zaakceptowanie warunków jest obowiązkowe."));
        }

        return errors;
    }
}

```

Interfejs `IValidatableObject` definiuje jedną z metod, `Validate`. Metoda ta otrzymuje parametr `ValidationContext`; choć typ ten nie jest specyficzny dla MVC, to nie ma problemów z jego użyciem. Wynikiem metody `Validate` jest kolekcja obiektów `ValidationResult`, z których każdy reprezentuje pojedynczy błąd kontroli poprawności.

Jeżeli nasza klasa modelu implementuje interfejs `IValidatableObject`, to metoda `Validate` zostanie wywołana po przypisaniu wartości do każdej z właściwości modelu. Podejście to łączy ze sobą elastyczność umieszczenia logiki kontroli poprawności w metodzie akcji ze spójnością wywoływania przez proces dołączania modelu przy tworzeniu obiektów modelu. Niektórzy programiści nie lubią umieszczać logiki kontroli poprawności w klasie modelu, ale uważamy, że nieźle pasuje to do wzorca projektowego MVC — dodatkowo podoba się nam elastyczność i spójność.

## Tworzenie własnego dostawcy kontroli poprawności

Alternatywnym podejściem do kontroli poprawności jest utworzenie własnego *dostawcy kontroli poprawności*. Realizujemy to przez odziedziczenie po klasie `ModelValidationProvider` i nadpisanie metody `GetValidators`. Na listingu 18.16 zamieszczony jest nasz dostawca kontroli poprawności, którego nazwaliliśmy `CustomValidationProvider`.

**Listing 18.16.** Własna klasa dostawcy kontroli poprawności

```
public class CustomValidationProvider : ModelValidatorProvider {

    public override IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
        ControllerContext context) {

        if (metadata.ContainerType == typeof(Appointment)) {
            return new ModelValidator[] {
                new AppointmentPropertyValidator(metadata, context)
            };
        } else if (metadata.ModelType == typeof(Appointment)) {
            return new ModelValidator[] {
                new AppointmentValidator(metadata, context)
            };
        }
        return Enumerable.Empty<ModelValidator>();
    }
}
```

Metoda `GetValidation` jest wywoływana jeden raz dla każdej właściwości modelu, a następnie dla całego modelu. Wynikiem metody jest lista obiektów `ModelValidator`. Każdy ze zwróconych obiektów `ModelValidator` będzie użyty do zweryfikowania właściwości lub modelu.

Możemy zdefiniować metodę `GetValidation` w dowolny sposób. Jeżeli nie chcemy oferować kontroli poprawności dla właściwości lub modelu, to po prostu zwracamy pustą listę. Aby zademonstrować funkcję dostawcy kontroli poprawności, zdecydowaliśmy się zaimplementować jeden obiekt dla właściwości klasy `Appointment` oraz jeden dla klasy `Appointment`.

Aby sprawdzić, do jakich celów został wywołany obiekt kontroli poprawności, odczytujemy wartość właściwości `ModelMetadata` przekazanej do metody jako parametr. Dostępne trzy parametry tej metody są opisane w tabeli 18.4.

**Tabela 18.4.** Przydatne właściwości klasy `ModelMetadata`

Właściwość	Opis
<code>ContainerType</code>	Gdy zostaniemy poproszeni o dostarczenie obiektu kontroli poprawności dla właściwości, zwraca typ zawierającego ją modelu. Jeżeli na przykład jesteśmy proszeni o obiekt kontroli poprawności dla właściwości <code>Appointment.ClientName</code> , <code>ContainerType</code> będzie zwracać typ klasy <code>Appointment</code> .
<code>PropertyName</code>	Zwraca nazwę właściwości, dla której dostarczany jest obiekt kontroli poprawności. Jeżeli na przykład jesteśmy proszeni o obiekt kontroli poprawności dla właściwości <code>Appointment.ClientName</code> , <code>PropertyName</code> będzie zwracać wartość <code>ClientName</code> .
<code>ModelType</code>	Jeżeli jesteśmy proszeni o obiekt kontroli poprawności dla modelu, to ta właściwość będzie zwracać typ obiektu modelu.

- **Wskazówka** Poniższy przykład pokazujemy wyłącznie jako ilustrację sposobu dołączania własnych dostawców kontroli poprawności do biblioteki. *Nie powinieneś* używać tej techniki w zwykłych scenariuszach kontroli poprawności, w których zastosowanie atrybutów metadanych lub `IValidatableObject` będzie wystarczające i znacznie prostsze. Z dostawców kontroli poprawności powinno się korzystać wyłącznie w skomplikowanych scenariuszach, na przykład przy dynamicznym ładowaniu zasad poprawności z bazy danych lub implementowaniu własnej biblioteki kontroli poprawności.

Na listingu 18.17 zamieszczona jest nasza klasa dziedzicząca po `ModelBinder`, używana dla właściwości.

**Listing 18.17.** *Obiekt kontroli poprawności działający na poziomie właściwości*

```

public class AppointmentPropertyValidator : ModelValidator {

    public AppointmentPropertyValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context) {
    }

    public override IEnumerable<ModelValidationResult> Validate(object container) {

        Appointment appt = container as Appointment;

        if (appt != null) {

            switch (Metadata.PropertyName) {
                case "ClientName":
                    if (string.IsNullOrEmpty(appt.ClientName)) {
                        return new ModelValidationResult[] {
                            new ModelValidationResult {
                                MemberName = "ClientName",
                                Message = "Proszę podać nazwisko."
                            }
                        };
                    }
                    break;
                case "Date":
                    if (appt.Date == null || DateTime.Now > appt.Date) {
                        return new ModelValidationResult[] {
                            new ModelValidationResult {
                                MemberName = "Date",
                                Message = "Proszę podać przyszłą datę."
                            }
                        };
                    }
                    break;
                case "TermsAccepted":
                    if (!appt.TermsAccepted) {
                        return new ModelValidationResult[] {
                            new ModelValidationResult {
                                MemberName = "TermsAccepted",
                                Message = "Zaakceptowanie warunków jest obowiązkowe."
                            }
                        };
                    }
                    break;
            }
        }
        return Enumerable.Empty<ModelValidationResult>();
    }
}

```

Obiekt `ModelData` przekazany do konstruktora informuje o tym, którą właściwość powinna kontrolować dana instancja klasy `AppointmentPropertyValidator`. Po wywołaniu metody `Validate` przełączamy się na wartość `Metadata.PropertyName` i wykonujemy odpowiednią weryfikację. Jeżeli wystąpi błąd, zwracamy listę zawierającą jeden obiekt `ModelValidationResult`. System kontroli poprawności modelu pozwala na zwrócenie w razie potrzeby różnych błędów, ale nie potrzebujemy tego, ponieważ wykonujemy tylko jedno sprawdzenie dla każdej właściwości klasy `Appointment`.

Proces kontroli poprawności całego modelu jest nieco inny. Na listingu 18.18 zamieszczona jest klasa kontroli poprawności.

**Listing 18.18.** *Obiekt kontroli poprawności działający na poziomie modelu*

```
public class AppointmentValidator : ModelValidator {

    public AppointmentValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context) {
    }

    public override void Validate(Appointment container,
        IList<ModelValidationResult> errors) {

        Appointment appt = (Appointment)Metadata.Model;

        if (appt.ClientName == "Jan" && appt.Date.DayOfWeek == DayOfWeek.Monday) {
            errors.Add(new ModelValidationResult {
                MemberName = "",
                Message = "Jan nie może rezerwować w poniedziałki."
            });
        }
    }
}
```

W czasie kontroli modelu nie istnieje kontener, więc jego parametr ma wartość null. Na początek odczytujemy model z właściwości `Modeldata.Model`, a następnie wykonujemy kontrolę poprawności. Aby zaraportować błąd na poziomie modelu, ustawiamy właściwość `MemberName` w obiekcie `ModelValidationResult` na pusty ciąg znaków ("").

- 
- **Wskazówka** Biblioteka MVC wywoła nasz obiekt kontroli poprawności modelu wyłącznie wtedy, gdy żaden z obiektów kontroli poprawności nie zwróci błędu. Jest to rozsądne, ponieważ bazuje na założeniu, że nie można sprawdzać poprawności modelu, jeżeli wystąpiły błędy we właściwościach.
- 

## Rejestrowanie własnego dostawcy kontroli poprawności

Musimy teraz zarejestrować własnego dostawcę kontroli poprawności w bibliotece MVC, co realizujemy w metodzie `Application_Start` znajdującej się w pliku `Global.asax`. Na listingu 18.19 zamieszczony jest sposób rejestracji naszej klasy dostawcy kontroli poprawności dla klasy `Appointment`.

**Listing 18.19.** *Rejestrowanie własnego dostawcy kontroli poprawności*

```
protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    ModelValidatorProviders.Providers.Add(new CustomValidationProvider());

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

Naszą klasę dodajemy do zbioru dostawców kontroli poprawności za pomocą metody `ModelValidatorProviders.Providers.Add`. Nasz dostawca będzie użyty równolegle z wbudowanymi dostawcami, co oznacza, że można również wykorzystać inne techniki przedstawione w tym rozdziale. Jeżeli chcesz usunąć pozostałych dostawców, możesz to zrobić za pomocą metody `Clear`, zanim dodasz własnego dostawcę.

```
ModelValidatorProviders.Providers.Clear();
```



## Użycie kontroli poprawności po stronie klienta

Do tej pory demonstrowaliśmy techniki kontroli poprawności będące przykładami *kontroli poprawności po stronie serwera*. Techniki te wymagają przesłania danych na serwer, skontrolowania ich na serwerze, a następnie odesłania wyniku kontroli (informacji o udanym przetworzeniu danych lub listy błędów wymagających skorygowania).

W aplikacji WWW użytkownicy wymagają zwykle szybkiej kontroli poprawności — bez konieczności wysyłania czegokolwiek na serwer. Taki mechanizm *kontroli poprawności po stronie klienta* jest zwykle implementowany z użyciem JavaScriptu. Dane wprowadzone przez użytkownika są kontrolowane przed wysłaniem na serwer, dzięki czemu reakcja aplikacji jest natychmiastowa i użytkownik od razu może korygować problemy.

Biblioteka MVC wspiera *nieprzeszkadzającą kontrolę poprawności po stronie klienta*. Słowo *nieprzeszkadzająca* oznacza, że zasady kontroli poprawności są wyrażane za pomocą atrybutów dodawanych do generowanych elementów HTML. Są one interpretowane przez bibliotekę JavaScript będącą częścią biblioteki MVC, wykorzystującą wartości atrybutów do skonfigurowania biblioteki jQuery Validation, która realizuje faktyczną kontrolę poprawności.

- 
- **Wskazówka** W MVC 3 skorzystano z biblioteki jQuery Validation, natomiast we wcześniejszych wersjach używane były biblioteki JavaScript napisane w firmie Microsoft. Nie były one najlepsze i choć są nadal dostępne w bibliotece, nie ma powodu z nich korzystać.
- 

Słowo *nieprzeszkadzający* jest używane powszechnie w kontekście kodu JavaScript. Jest to luźny termin wskazujący na trzy cechy kodu. Pierwszą jest oddzielenie kodu JavaScript realizującego kontrolę poprawności od elementów HTML, dzięki czemu nie musimy dołączać logiki kontroli poprawności do naszych widoków, a generowany kod HTML jest czytelniejszy.

Drugą cechą jest realizacja kontroli poprawności z użyciem *progresywnego rozszerzania*. Oznacza to, że w przypadku braku obsługi wszystkich funkcji JavaScript przez przeglądarkę kontrola poprawności będzie realizowana z zastosowaniem prostszych technik. Jeżeli użytkownik na przykład zablokował JavaScript, to będzie wykorzystywana kontrola poprawności po stronie serwera bez wpływania w jakikolwiek sposób na działanie aplikacji (nie będą wyświetlane nieprzyjemne komunikaty o błędach ani nie będzie wymagane wykonanie dodatkowych akcji).

Trzecią cechą jest użycie zbioru praktyk pozwalających na ograniczenie wpływu niespójności w działaniu przeglądarek. Do tego tematu wrócimy w czasie przedstawiania biblioteki jQuery w rozdziale 20.

W kolejnych punktach pokażemy, w jaki sposób działają wbudowane mechanizmy kontroli poprawności, oraz zademonstrujemy sposoby ich rozszerzenia w celu zapewnienia własnych mechanizmów kontroli poprawności po stronie klienta.

- 
- **Wskazówka** Kontrola poprawności po stronie klienta skupia się na weryfikowaniu pojedynczych właściwości. Faktycznie trudno jest skonfigurować kontrolę poprawności po stronie klienta dla modelu, używając mechanizmów dostarczonych w bibliotece MVC. Z tego powodu większość aplikacji MVC 3 korzysta z kontroli poprawności po stronie klienta dla właściwości i bazuje na kontroli poprawności po stronie serwera dla całego modelu, stosując jedną z technik przedstawionych w poprzedniej części tego rozdziału.
- 

## Aktywowanie i wyłączanie kontroli poprawności po stronie klienta

Kontrola poprawności po stronie klienta jest sterowana za pomocą dwóch ustawień w pliku *Web.config*, pokazanych na listingu 18.20.

**Listing 18.20.** Sterowanie kontrolą poprawności po stronie klienta

```
<configuration>
  <appSettings>
    <add key="ClientValidationEnabled" value="true"/>
```

```

    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>

```

...

Aby kontrola poprawności po stronie klienta działała, oba te ustawienia muszą mieć wartość true. Podczas generowania projektu MVC 3 Visual Studio tworzy te wpisy i przypisuje im wartość true. Możemy na przykład programowo kontrolować te ustawienia w pliku *Global.asax* w sposób pokazany na listingu 18.21.

**Listing 18.21.** Programowe sterowanie kontrolą poprawności po stronie klienta

```

protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    HtmlHelper.ClientValidationEnabled = true;
    HtmlHelper.UnobtrusiveJavaScriptEnabled = true;

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}

```

Możemy również włączać i wyłączać kontrolę poprawności po stronie klienta dla pojedynczych widoków. Powoduje to nadpisanie pokazanych powyżej opcji konfiguracji. Na listingu 18.22 przedstawiony jest sposób sterowania programowego wewnątrz bloku kodu Razor, w którym wyłączamy kontrolę poprawności dla danego widoku.

**Listing 18.22.** Sterowanie kontrolą poprawności po stronie klienta dla pojedynczego widoku

```

@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
    HtmlHelper.ClientValidationEnabled = false;
}
...

```

Oba te ustawienia muszą mieć wartość true, aby działała kontrola poprawności po stronie klienta, co oznacza, że wystarczy jedno z nich ustawić na false, aby wyłączyć funkcję. Ponadto należy upewnić się, że są dodane odwołania do trzech bibliotek JavaScript, wyróżnionych na listingu 18.23.

**Listing 18.23.** Odwołania do bibliotek JavaScript wymaganych do kontroli poprawności po stronie klienta

```

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
        type="text/javascript"></script>

    <script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
        type="text/javascript"></script>

    <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
        type="text/javascript"></script>
</head>
<body>

```

```
@RenderBody ()
</body>
</html>
```

Możemy dodawać te pliki do każdego widoku, w którym chcemy użyć kontroli poprawności po stronie klienta, ale zwykle prościej jest umieścić te odwołania w pliku układu, tak jak na zamieszczonym listingu.

- 
- **Ostrzeżenie** Kolejność, w której są umieszczone pliki jQuery, jest znacząca. Jeżeli ją zmienisz, okaże się, że kontrola poprawności po stronie klienta nie działa.
- 

Katalog *scripts* zawiera dwie wersje każdej z bibliotek JavaScript. Wersje, których nazwy kończą się na *min.js*, są wersjami *zminimalizowanymi*, co oznacza, że wszystkie odstępy, komentarze i inne niekrytyczne dane są usunięte, co zmniejsza rozmiar pliku biblioteki. Wersje zminimalizowane mogą być znacznie mniejsze i zwykle są używane w środowiskach produkcyjnych w celu ograniczenia ilości danych pobieranych przez klienta. W czasie tworzenia aplikacji zwykle są wykorzystywane wersje niezminimalizowane, dzięki czemu można debugować (lub po prostu czytać) JavaScript w przypadku wystąpienia problemów.

## Użycie CDN dla bibliotek JavaScript

Na listingu 18.22 odwołujemy się do bibliotek jQuery znajdujących się w folderze *~/Scripts* naszej aplikacji. Alternatywnym podejściem jest załadowanie tych plików z sieci Content Delivery Network (CDN) firmy Microsoft. Jest to usługa bezpłatna. Dostępnych jest kilka rozszaniach geograficznie serwerów, które obsługują żądania pobrania plików bibliotek JavaScript dla aplikacji MVC, używając serwera znajdującego się najbliżej użytkownika.

Istnieje kilka zalet korzystania z CDN. Otóż czas, jaki zajmuje załadowanie aplikacji przez przeglądarkę użytkownika, może się zmniejszyć, ponieważ serwery CDN są szybsze i znajdują się bliżej użytkownika niż serwery aplikacji. Dodatkowo zalety te łączą się, jeżeli użytkownik pobrał już wymagane pliki w innej aplikacji, która korzysta z plików z tej samej lokalizacji CDN. Kolejną zaletą jest zmniejszenie obciążenia serwera oraz zużycia łącza wykorzystywanego do dostępu do aplikacji. Pliki jQuery są zwykle największymi elementami, jakie musimy dostarczyć do przeglądarki w ramach aplikacji MVC, więc w przypadku pobierania tych plików z serwerów firmy Microsoft zmniejszamy własne koszty. Aby skorzystać z CDN, musimy zmienić atrybut `src` znacznika `script`, by wskazywał na następujące adresy URL:

```
http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.5.1.min.js
http://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js
http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.min.js
```

Dla przykładu znacznik `script` odwołujący się do podstawowej biblioteki jQuery będzie wyglądał następująco:

```
<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.5.1.min.js"
type="text/javascript"></script>
```

Serwery CDN obsługują wiele wersji biblioteki jQuery, więc dla każdego pliku musimy wybrać prawidłowy adres URL. Listę dostępnych plików (oraz ich adresów URL) można zobaczyć na stronie [www.asp.net/ajaxlibrary/cdn.ashx](http://www.asp.net/ajaxlibrary/cdn.ashx).

Usługa CDN jest przydatna w aplikacjach udostępnianych w internecie, ale jeżeli projekt jest przeznaczony dla intranetu, to użycie CDN nie ma większego sensu. W takim przypadku szybciej i taniej jest pobrać biblioteki JavaScript z serwera aplikacji.

---

## Użycie kontroli poprawności po stronie klienta

Po włączeniu kontroli poprawności po stronie klienta i upewnieniu się, że w widoku istnieją odwołania do biblioteki jQuery, możemy zacząć weryfikację danych użytkownika. Najprostszym sposobem realizacji tego zadania jest użycie atrybutów metadanych wykorzystywanych wcześniej do kontroli poprawności po stronie serwera, takich jak `Required`, `Range` czy `StringLength`. Na listingu 18.24 zamieszczona jest klasa modelu `Appointment` z dodanymi tymi adnotacjami.

**Listing 18.24.** Atrybuty kontroli poprawności zastosowane w klasie modelu `Appointment`

```
public class Appointment {

    [Required]
    [StringLength(10, MinimumLength=3)]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [Required(ErrorMessage="Proszę podać datę")]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }
}
```

Więcej informacji na temat tych atrybutów można znaleźć w punkcie „Definiowanie zasad poprawności za pomocą metadanych”, we wcześniejszej części rozdziału. Mechanizmy kontroli poprawności działające po stronie klienta wyświetlają błędy kontroli poprawności nieco inaczej w przypadku użycia podsumowań oraz komunikatów, więc zmodyfikowaliśmy widok `MakingBooking`, aby korzystał z obu mechanizmów (listing 18.25).

**Listing 18.25.** Dodanie podsumowania kontroli poprawności oraz elementów do formularza HTML

```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

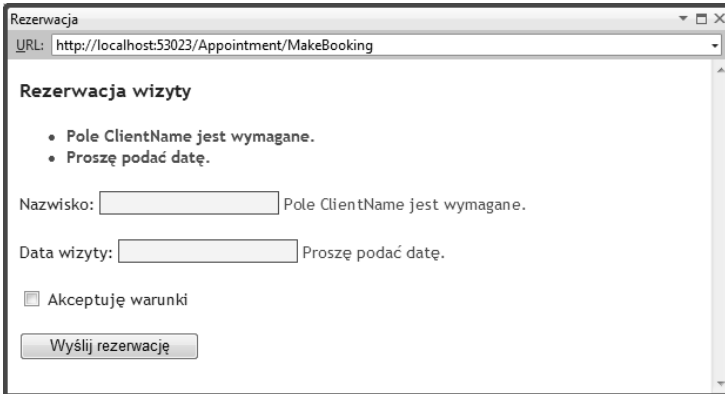
    @Html.ValidationSummary()

    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)
        @Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Data wizyty: @Html.EditorFor(m => m.Date)
        @Html.ValidationMessageFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki
        @Html.ValidationMessageFor(m => m.TermsAccepted)</p>

    <input type="submit" value="Wyślij rezerwację" />

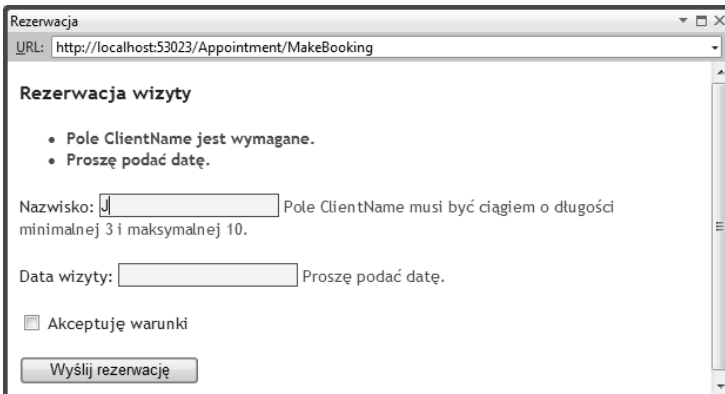
}
```

Gdy otrzymamy żądanie wywołujące metodę akcji `MakeBooking` z naszego kontrolera, generujemy widok `MakeBooking.cshtml` — jest to standardowe działanie. Jednak gdy klikniemy przycisk wysłania danych, zasady kontroli poprawności danych dodane do klasy `Appointment` będą wykorzystywane w przeglądarce z użyciem JavaScript, jak pokazano na rysunku 18.10.



**Rysunek 18.10.** Komunikaty o błędach kontroli poprawności po stronie klienta

Wyświetlane komunikaty wyglądają identycznie jak te generowane po stronie serwera, ale do ich wygenerowania nie była konieczna komunikacja z serwerem. Jeżeli masz serwer i przeglądarkę zainstalowaną na tym samym komputerze, trudno jest zauważyć różnicę w szybkości działania, ale w przypadku połączeń internetowych lub zatłoczonych sieci korporacyjnych różnica może być spora. Istnieje również jeszcze jedna różnica. Kontrola poprawności po stronie klienta jest wykonywana przy pierwszej próbie przesłania danych formularza, ale później jest realizowana po każdym naciśnięciu klawisza lub zmianie fokusu w formularzu HTML. Gdy wpisujemy literę *J* do pola `ClientName`, spowoduje to spełnienie warunku z atrybutu `Required` dodanego do tej właściwości w klasie `Appointment`. Jednak zauważone jest naruszenie zasady z atrybutu `StringLength`, więc obok pola pojawia się następny komunikat, pokazany na rysunku 18.11.

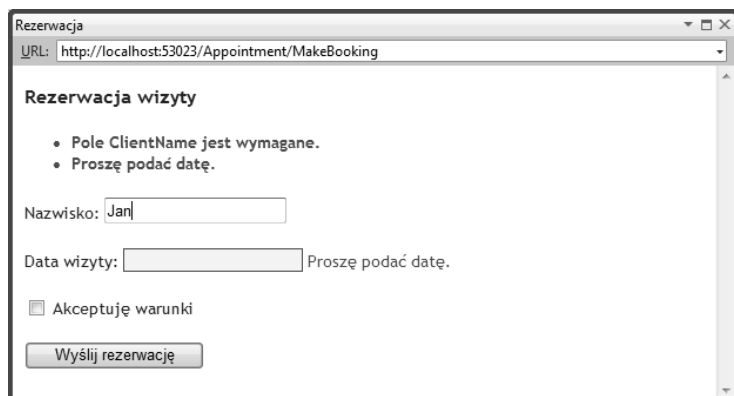


**Rysunek 18.11.** Automatyczna aktualizacja komunikatów kontroli poprawności po stronie klienta

Komunikat o błędzie obok pola się zmienił. Nie wykonaliśmy żadnych specjalnych akcji, aby się to stało; po prostu wpisaliśmy znak do pola. Kontrola poprawności została wykonana ponownie i wyświetlił się komunikat o nowym błędzie.

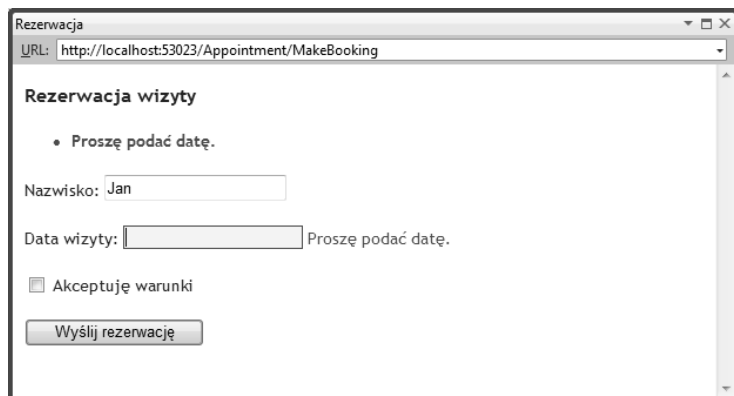
- 
- **Wskazówka** Zwróć uwagę, że podsumowanie kontroli poprawności utworzone za pomocą metody `Html.ValidationSummary` nie zmienia się — nie jest ono aktualizowane do momentu ponownego kliknięcia przycisku *Wyślij rezerwację*. To właśnie mieliśmy na myśli, mówiąc, że podsumowania działają inaczej niż komunikaty dla właściwości w przypadku użycia kontroli poprawności po stronie klienta.
-

Kontynuujemy pisanie i gdy dojdziemy do litery *n* w słowie *Jan*, spełnione zostaną obie zasady poprawności zdefiniowane dla właściwości `ClientName` w klasie `Appointment`. Została podana wartość mająca długość od trzech do dziesięciu znaków. W tym momencie komunikaty o błędach przy polu znikają, a wyróżnienie pola jest usuwane, jak pokazano na rysunku 18.12.



**Rysunek 18.12.** Automatyczne usuwanie sygnalizacji błędów kontroli poprawności

Użytkownik otrzymuje natychmiastową informację o błędzie kontroli poprawności, ponadto komunikat o błędzie jest aktualizowany w czasie wprowadzania danych. Trudno to pokazać w postaci serii ekranów, ale jest to znacznie bardziej przyjazny sposób kontroli poprawności danych. Nie tylko zapewnia lepsze działanie interfejsu użytkownika, ale również oznacza, że aplikacja otrzymuje mniej żądań POST, które muszą być przetworzone przez serwer. Gdy mamy satysfakcjonującą nas wartość właściwości `ClientName`, ponownie przesyłamy dane formularza. Jest to jedyny moment, w którym jest aktualizowane podsumowanie, co jest pokazane na rysunku 18.13.



**Rysunek 18.13.** Przesłanie danych formularza aktualizuje podsumowanie kontroli poprawności

Rzadsza aktualizacja podsumowań powoduje, że są one mniej atrakcyjne niż komunikaty przy polach, szczególnie gdy dla pola jest zdefiniowanych kilka zasad poprawności. Nie oznacza to, że nie powinniśmy używać podsumowań, ale należy to robić rozważnie, biorąc pod uwagę efekt, jaki zobaczą użytkownicy.

## Jak działa kontrola poprawności po stronie klienta?

Jedną z zalet użycia funkcji kontroli poprawności po stronie klienta w bibliotece MVC jest brak konieczności tworzenia kodu JavaScript. Zamiast tego zasady poprawności są definiowane za pomocą atrybutów HTML.

Poniżej zamieszczony jest kod HTML wygenerowany przez metodę pomocniczą `Html.EditorFor` dla właściwości `ClientName` przy wyłączonej kontroli poprawności po stronie klienta.

```
<input class="text-box single-line" id="ClientName" name="ClientName"
  type="text" value="" />
```

Po włączeniu kontroli poprawności i wygenerowaniu edytora otrzymamy następujący kod:

```
<input class="text-box single-line" data-val="true" data-val-length="The field ClientName must be
↳ a string with a minimum length of 3 and a maximum length of 10." data-val-length-max="10"
↳ data-val-length-min="3" data-val-required="The ClientName field is required." id="ClientName"
↳ name="ClientName" type="text" value="" />
```

Kontrola poprawności po stronie klienta nie generuje bezpośrednio żadnego kodu JavaScript ani danych JSON sterujących procesem kontroli poprawności; podobnie jak w pozostałych częściach biblioteki MVC są wykorzystywane konwencje.

Pierwszym dodanym atrybutem jest `data-val`. Biblioteka jQuery Validation identyfikuje pola wymagające kontroli poprawności przez wyszukiwanie tego atrybutu.

Poszczególne zasady poprawności są specyfikowane za pomocą atrybutu w postaci `data-val-<nazwa>`, gdzie *nazwa* jest zasadą do zastosowania. Na przykład atrybut `Required` dodany do klasy modelu powoduje wygenerowanie w HTML-u atrybutu `data-val-required`. Wartość skojarzona z atrybutem stanowi komunikat o błędzie skojarzony z zasadą. Niektóre zasady wymagają dodatkowych atrybutów. Można zauważyć, że definicja długości pola została przekształcona na atrybuty `data-val-length-min` oraz `data-val-length-max`, w których zdefiniowana została minimalna i maksymalna długość napisu.

Interpretacja zasad poprawności jest realizowana przez bibliotekę jQuery Validation, na bazie której są zbudowane mechanizmy kontroli poprawności po stronie klienta.

Jedną z lepszych cech kontroli poprawności po stronie klienta jest użycie tych samych atrybutów do tworzenia zasad poprawności wykorzystywanych po stronie klienta *oraz* po stronie serwera. Oznacza to, że dane w przeglądarkach nieobsługujących JavaScriptu podlegają takiej samej kontroli poprawności jak w przeglądarkach obsługujących ten język, bez wykonywania żadnych dodatkowych operacji.

## Modyfikowanie kontroli poprawności na kliencie

Wbudowana obsługa kontroli poprawności na kliencie sprawdza się bardzo dobrze, ale jest ograniczona do sześciu atrybutów zamieszczonych w tabeli 18.3, we wcześniejszej części rozdziału. Biblioteka jQuery Validation obsługuje nieco więcej zasad poprawności, a biblioteka nieprzeszkadzającej kontroli poprawności w MVC pozwala nam skorzystać z nich przy stosunkowo niewielkim nakładzie pracy.

## Jawne tworzenie atrybutów kontroli poprawności w HTML

Najbardziej bezpośrednią metodą wykorzystania dodatkowych zasad poprawności jest ręczne wygenerowanie w widoku wymaganych atrybutów, jak pokazano na listingu 18.26.

**Listing 18.26.** Ręczne generowanie atrybutów kontroli poprawności w HTML

```
@model MvcApp.Models.Appointment
@{
  ViewBag.Title = "Rezerwacja";
}
<h4>Rezerwacja wizyty</h4>
@using (Html.BeginForm()) {
    @Html.ValidationSummary()
```

## Kontrola poprawności po stronie klienta w MVC a kontrola poprawności w jQuery

Funkcja kontroli poprawności po stronie klienta w bibliotece MVC jest zbudowana w oparciu o bibliotekę jQuery Validation i jeżeli masz takie życzenie, możesz korzystać z niej bezpośrednio, ignorując funkcje MVC. Biblioteka ta jest bardzo elastyczna i bogata w możliwości. Warto się z nią zapoznać, choćby po to, aby zrozumieć, jak można modyfikować funkcje MVC w celu najlepszego możliwego wykorzystania dostępnych opcji kontroli poprawności. Do użycia biblioteki jQuery Validation niezbędna jest pewna znajomość JavaScriptu. Poniżej zamieszczony jest przykład wymaganego skryptu.

```
$(document).ready(function () {

    $('form').validate({
        errorLabelContainer: '#validationSummary',
        wrapper: 'li',
        rules: {
            ClientName: {
                required: true,
            }
        },
        messages: {
            ClientName: "Proszę podać nazwisko."
        }
    });
});
```

Funkcje kontroli poprawności po stronie klienta dostępne w MVC ukrywają kod JavaScript, a dodatkowo pozwalają realizować identyczne funkcje kontroli poprawności na kliencie i serwerze. Oba podejścia mogą być wykorzystane w aplikacji MVC, choć powinniśmy ostrożnie podchodzić do ich mieszania w jednym widoku, ponieważ mogą pojawić się niechciane interakcje. Więcej informacji na temat biblioteki jQuery Validation można znaleźć na stronie <http://bassistance.de/jquery-plugins>.

```
<p>Nazwisko:
    @Html.TextBoxFor(m => m.ClientName, new { data_val = "true",
        data_val_email = "Podaj prawidłowy adres e-mail.",
        data_val_required = "Podaj swoje nazwisko."})

    @Html.ValidationMessageFor(m => m.ClientName)</p>
<p>Data wizyty: @Html.EditorFor(m => m.Date)
    @Html.ValidationMessageFor(m => m.Date)</p>
<p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki
    @Html.ValidationMessageFor(m => m.TermsAccepted)</p>

<input type="submit" value="Wyślij rezerwację" />
}
```

Jeżeli chcemy dodać kolejne atrybuty do znacznika, nie możemy użyć metod pomocniczych widoku szablonowego do wygenerowania właściwości dla edytora, więc zamiast tego zastosowaliśmy metodę `Html.TextBoxFor` w wersji akceptującej typ anonimowy z atrybutami HTML.

- **Wskazówka** Segmenty nazw atrybutów HTML są rozdzielane łącznikami (-), ale łącznik jest znakiem nielegalnym w nazwach zmiennych C#. Aby obejść ten problem, podajemy nazwy atrybutów z użyciem podkreśleń (\_), które są automatycznie konwertowane na łączniki w czasie generowania kodu HTML.



Po wygenerowaniu widoku otrzymamy następujący kod HTML dla właściwości `ClientName`:

```
<input data-val="true" data-val-email="Podaj prawidłowy adres e-mail."
  data-val-required="Podaj swoje nazwisko." id="ClientName"
  name="ClientName" type="text" value="" />
```

Zasada `required` jest również generowana przez atrybut `Required`. Zasada `email` pozwala upewnić się, że wartość prowadzona do pola ma format prawidłowego adresu e-mail. W tabeli 18.5 opisane są zasady kontroli poprawności, z których możemy skorzystać.

**Tabela 18.5.** Użyteczne zasady kontroli poprawności dostępne w *jQuery*

Zasada poprawności	Atrybut kontroli poprawności	Opis
Required	Required	Wymaga podania wartości; jest to zasada wykorzystywana przez atrybut <code>Required</code> .
Length	StringLength	Liczba znaków w podanej wartości musi być większa lub równa <code>minimum</code> i (lub) mniejsza lub równa <code>maximum</code> . Minimalna długość jest definiowana za pomocą atrybutu <code>data-val-length-min</code> , a maksymalna długość za pomocą atrybutu <code>data-val-length-max</code> . Możemy podać jedynie atrybut <code>-min</code> lub <code>-max</code> , co spowoduje ograniczenie tylko jednego aspektu długości wartości.
Range	Range	Wartość musi znajdować się pomiędzy granicami zdefiniowanymi za pomocą atrybutów <code>data-val-required-min</code> i <code>data-val-required-max</code> . Można podać tylko jeden z tych atrybutów, co spowoduje kontrolę tylko górnego lub dolnego limitu wartości.
Regex	RegularExpression	Wartość musi pasować do wyrażenia regularnego zdefiniowanego za pomocą atrybutu <code>data-val-regex-pattern</code> .
EqualTo	Compare	Wartość musi być taka sama jak wartość w elemencie wskazanym za pomocą atrybutu <code>data-val-equalto-other</code> .
Email	-	Wartość musi być prawidłowym adresem e-mail.
Url	-	Wartość musi być prawidłowym adresem URL.
Date	-	Wartość musi być prawidłową datą.
Number	-	Wartość musi być liczbą (może zawierać cyfry po przecinku).
Digits	-	Wartość musi zawierać wyłącznie cyfry.
Creditcard	-	Wartość musi być prawidłowym numerem karty kredytowej.

Zasady kontroli poprawności, dla których nie istnieją odpowiednie atrybuty C#, pozwalają na sprawdzenie formatu, ale nie są w stanie sprawdzić, czy wartość jest naprawdę prawidłowa. Na przykład zasada poprawności `creditcard` sprawdza, czy wartość jest w prawidłowym formacie dla numeru karty kredytowej oraz czy wartość spełnia schemat kodowania *Luhn*, który jest używany do sprawdzania sumy kontrolnej. Oczywiście nie ma gwarancji, że wartość podana przez użytkownika reprezentuje kartę kredytową faktycznie wydaną przez instytucję finansową.

Podobnie zasady `email` oraz `url` pozwalają sprawdzić, czy format adresu e-mail lub URL jest prawidłowy, ale nie sprawdzają, czy konto e-mail lub strona WWW są dostępne. Jeżeli potrzebujesz bardziej rygorystycznej kontroli, to przydatną funkcją jest *zdalna kontrola poprawności*, którą przedstawimy w dalszej części rozdziału.

## Tworzenie atrybutów modelu obsługujących kontrolę poprawności po stronie klienta

Dodawanie atrybutów HTML do naszych elementów widoku jest proste, ale powoduje, że kontrola poprawności jest realizowana wyłącznie po stronie klienta. Możemy temu przeciwdziałać przez wykonanie tej samej kontroli w metodzie akcji lub łączniku, ale lepszą techniką jest utworzenie własnego atrybutu kontroli poprawności, który działa w taki sam sposób jak wbudowane atrybuty i uruchamia kontrolę poprawności realizowaną na serwerze i kliencie. Na listingu 18.27 zamieszczony jest atrybut kontroli poprawności realizujący kontrolę formatu adresu e-mail na serwerze.

**Listing 18.27.** Atrybut kontroli poprawności realizujący kontrolę poprawności formatu adresu e-mail na serwerze

```
public class EmailAddressAttribute : ValidationAttribute {
    private static readonly Regex emailRegex = new Regex(".+@.+\\.+");

    public EmailAddressAttribute() {
        ErrorMessage = "Podaj prawidłowy adres e-mail.";
    }

    public override bool IsValid(object value) {
        return !string.IsNullOrEmpty((string)value) &&
            emailRegex.IsMatch((string)value);
    }
}
```

Jest to podejście takie same jak w przypadku tworzenia atrybutów kontroli poprawności po stronie serwera, które przedstawialiśmy we wcześniejszej części rozdziału. Tworzymy klasę dziedziczącą po `ValidationAttribute` i zmieniamy metodę `IsValid`, w której umieszczamy naszą logikę kontroli poprawności.

---

■ **Uwaga** Aby zachować prostotę, w przykładzie tym użyliśmy bardzo prostego wyrażenia regularnego do kontroli adresu e-mail. W sieci można łatwo znaleźć bardziej zaawansowane wzorce.

---

Aby umożliwić działanie kontroli poprawności po stronie klienta, musimy zaimplementować interfejs `IClientValidatable`, zamieszczony na listingu 18.28.

**Listing 18.28.** Interfejs `IClientValidatable`

```
public interface IClientValidatable {
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}
```

Interfejs ten definiuje jedną metodę, `GetClientValidationRules`, która zwraca listę obiektów `ModelClientValidationRule`. Każdy obiekt `ModelClientValidationRule` opisuje zasadę kontroli poprawności po stronie klienta, jaką należy zastosować, komunikat o błędzie wyświetlany w przypadku złamania tej zasady oraz wszystkie inne parametry wymagane do jej działania. Na listingu 18.29 zamieszczony jest kod ilustrujący sposób dodania kontroli poprawności po stronie klienta w klasie `EmailAddressAttribute` z listingu 18.27.

**Listing 18.29.** Dodanie obsługi strony klienckiej w klasie `EmailAddressAttribute`

```
public class EmailAddressAttribute : ValidationAttribute, IClientValidatable {
    private static readonly Regex emailRegex = new Regex(".+@.+\\.+");

    public EmailAddressAttribute() {
```

```

        ErrorMessage = "Podaj prawidłowy adres e-mail.";
    }

    public override bool IsValid(object value) {
        return !string.IsNullOrEmpty((string)value) &&
            emailRegex.IsMatch((string)value);
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context) {

        return new List<ModelClientValidationRule> {
            new ModelClientValidationRule {
                ValidationType = "email",
                ErrorMessage = this.ErrorMessage
            },
            new ModelClientValidationRule {
                ValidationType = "required",
                ErrorMessage = this.ErrorMessage
            }
        };
    }
}

```

Możemy zwrócić tyle obiektów `ModelClientValidationRule` potrzebnych do zdefiniowania zbioru zasad po stronie klienta, ile potrzeba do wymuszenia naszych zasad poprawności. W przedstawionym przykładzie użyliśmy zasad `email` oraz `required` (ustawiając właściwość `ValidationType` w obiekcie `ModelClientValidationRule`), które korzystają z komunikatu o błędzie zdefiniowanego w atrybucie (za pomocą właściwości `ErrorMessage`). Nasz nowy atrybut możemy przypisać do klasy modelu identycznie jak każdy inny atrybut kontroli poprawności:

```

public class Appointment {

    [EmailAddress]
    public string ClientName { get; set; }
    ...
}

```

Gdy tworzony jest edytor dla właściwości `ClientName`, silnik widoku analizuje użyte przez nas metadane, wyszukuje naszą implementację `IClientValidatable` i generuje atrybuty HTML przedstawione w poprzednim punkcie. Przesłane dane są ponownie sprawdzane za pomocą naszej metody `IsValid`. Nasz nowy atrybut jest używany do kontroli poprawności zarówno na serwerze, jak i na kliencie, co jest przyjemniejsze, bezpieczniejsze i bardziej spójne niż jawne generowanie atrybutów HTML.

## Tworzenie własnych zasad poprawności na kliencie

Przedstawione w tabeli 18.5 wbudowane zasady kontroli poprawności na kliencie są użyteczne, ale nie wyczerpują wszystkich możliwości. Na szczęście możesz przygotować własne zasady, jeżeli jesteś w stanie napisać kilka wierszy kodu JavaScript.

Obsługa kontroli poprawności na kliencie jest ograniczona w bibliotece MVC do zasad dostępnych w bazowej bibliotece jQuery. Sprowadza się to do tego, że możemy dostosowywać istniejące zasady na wiele sposobów, ale jeżeli chcemy utworzyć coś bardziej skomplikowanego, musimy zostawić wbudowaną obsługę kontroli poprawności z MVC i operować bezpośrednio na jQuery. Mimo że mamy narzucone ograniczenie, możemy tworzyć nowe, użyteczne mechanizmy.

Na przykład funkcje kontroli poprawności bazujące na kliencie nie obsługują pól wyboru w sposób prawidłowy, podobnie jak przedstawione wcześniej atrybuty kontroli poprawności na serwerze. Możemy utworzyć nową, kliencką zasadę poprawności, która będzie korzystała z zasady jQuery rule dla pól wyboru, jak pokazano na listingu 18.30.

**Listing 18.30.** Tworzenie własnego odwzorowania pomiędzy funkcjami kontroli poprawności w MVC i jQuery

```
<!DOCTYPE html>
<html>
<head>
  <title>@ViewBag.Title</title>
  <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.5.1.min.js"
type="text/javascript"></script>
<script src="http://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js"
type="text/javascript"></script>
<script src="http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.js"
type="text/javascript"></script>

<script type="text/javascript">
  jQuery.validator.unobtrusive.adapters.add("checkboxtrue", function (options) {
    if (options.element.tagName.toUpperCase() == "INPUT" &&
        options.element.type.toUpperCase() == "CHECKBOX") {

      options.rules["required"] = true;
      if (options.message) {
        options.messages["required"] = options.message;
      }
    }
  });
</script>

</head>

<body>
  @RenderBody()
</body>
</html>
```

Utworzyliśmy tu nową zasadę o nazwie `checkboxtrue`, która za pomocą zasady `required` z jQuery Validation ma za zadanie sprawdzić, czy pole wyboru jest zaznaczone. Skrypt ten dodaliśmy do pliku układu w projekcie (`_Layout.cshtml`), dzięki czemu jest on dostępny we wszystkich widokach.

- 
- **Uwaga** Dodawanie nowych zasad kontroli poprawności jest zaawansowanym procesem, który wymaga dobrej znajomości biblioteki jQuery Validation oraz obsługi kontroli poprawności po stronie klienta z biblioteki MVC. Nie będziemy wyjaśniać, jak działa skrypt z listingu 18.29, ale jeżeli chcesz nauczyć się dodawać nowe zasady poprawności, to na początek warto zapoznać się z kodem źródłowym z pliku `jQuery.validate.unobtrusive.js`.
- 

Po utworzeniu klienckiej zasady poprawności możemy utworzyć korzystający z niej atrybut. We wcześniejszej części rozdziału pokazaliśmy, jak utworzyć serwerowy atrybut kontroli poprawności, który pozwala upewnić się, że pole wyboru jest zaznaczone. Na listingu 18.29 rozszerzyliśmy tę klasę atrybutu o obsługę strony klienckiej, wykorzystując zasadę `checkboxtrue` z listingu 18.31.

**Listing 18.31.** Dodanie kontroli poprawności po stronie klienckiej w klasie `MustBeTrueAttribute`

```
public class MustBeTrueAttribute : ValidationAttribute, IClientValidatable {

  public override bool IsValid(object value) {
    return value is bool && (bool)value;
  }
}
```

```

    }
    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context) {

        return new ModelClientValidationRule[] {
            new ModelClientValidationRule {
                ValidationType = "checkboxtrue",
                ErrorMessage = this.ErrorMessage
            }
        };
    }
}

```

Teraz możemy korzystać z atrybutu `MustBeTrue` dla właściwości typu `bool` w klasach modelu, dzięki czemu upewnimy się, że użytkownik zaznaczy pole wyboru przed przesłaniem danych na serwer.

## Wykonywanie zdalnej kontroli poprawności

Ostatnią funkcją kontroli poprawności, jaką zajmiemy się w tym rozdziale, jest *zdalna kontrola poprawności*. Jest to technika kontroli poprawności po stronie klienta, która wywołuje na serwerze metodę akcji wykonującą kontrolę poprawności.

Częstym przykładem stosowania zdalnej kontroli poprawności jest sprawdzanie w czasie rejestracji, czy nazwa użytkownika jest dostępna. Po przesłaniu danych przez użytkownika wykonywana jest kontrola poprawności na serwerze. W procesie tym do serwera wysyłane są żądania Ajax, za pomocą których sprawdzamy, czy podana przez użytkownika nazwa jest dostępna. Jeżeli nazwa użytkownika jest zajęta, wyświetlany jest komunikatu o błędzie kontroli poprawności, a użytkownik może podać inną wartość.

Może to wyglądać jak zwykła kontrola poprawności po stronie klienta, ale podejście to ma kilka zalet. Po pierwsze, tylko część właściwości trzeba kontrolować zdalnie; w przypadku pozostałych możemy korzystać z zalet kontroli na kliencie. Po drugie, żądania są względnie niewielkie i skupione na kontroli poprawności, a nie przetwarzaniu całego modelu obiektu. Oznacza to, że możemy minimalizować spadek wydajności, jaki będą wywoływać te żądania.

Trzecią różnicą jest to, że zdalna kontrola poprawności jest realizowana w tle. Użytkownik nie musi klikać przycisku *Wyślij* i czekać na wygenerowanie i załadowanie nowego widoku. Dzięki temu interfejs użytkownika szybciej reaguje, szczególnie gdy połączenie pomiędzy przeglądarką a serwerem jest powolne.

Trzeba pamiętać, że zdalna kontrola poprawności jest kompromisem; pozwala zachować równowagę pomiędzy kontrolą poprawności na kliencie i serwerze, ale wymaga wysłania żądań do serwera aplikacji i nie jest tak szybka jak kontrola poprawności na serwerze.

---

■ **Wskazówka** Obsługę Ajax oraz JSON w bibliotece MVC przedstawiamy w rozdziale 19.

---

Pierwszym krokiem w kierunku użycia zdalnej kontroli poprawności jest utworzenie metody akcji, która będzie sprawdzała poprawność jednej z właściwości modelu. W naszej klasie modelu `Appointment` będziemy kontrolować poprawność właściwości `Date`, upewniając się, że podana data jest datą przyszłą (jest to początkowa zasada poprawności, jaką zdefiniowaliśmy na początku tego rozdziału). Na listingu 18.32 pokazana jest metoda `ValidateDate`, którą dodaliśmy do klasy `AppointmentController`.

**Listing 18.32.** Dodanie metody kontroli poprawności do kontrolera

```

public class AppointmentController : Controller {
    private IAppointmentRepository repository;

    public AppointmentController(IAppointmentRepository repo) {
        repository = repo;
    }
}

```

```

    }

    public ActionResult MakeBooking() {
        return View(new Appointment { Date = DateTime.Now });
    }

    public JsonResult ValidateDate(string Date) {

        DateTime parsedDate;

        if (!DateTime.TryParse(Date, out parsedDate)) {
            return Json("Proszę podać prawidłową datę (mm/dd/yyyy).",
                JsonRequestBehavior.AllowGet);
        } else if (DateTime.Now > parsedDate) {
            return Json("Proszę podać przyszłą datę.", JsonRequestBehavior.AllowGet);
        } else {
            return Json(true, JsonRequestBehavior.AllowGet);
        }
    }

    [HttpPost]
    public ActionResult MakeBooking(Appointment appt) {

        if (ModelState.IsValid) {
            repository.SaveAppointment(appt);
            return View("Completed", appt);
        } else {
            return View();
        }
    }
}

```

Metody akcji obsługujące kontrolę poprawności muszą zwracać wartość typu `JsonResult`, a nazwa parametru metody musi być taka sama jak kontrolowane pole. W naszym przypadku jest to `Date`. Musimy upewnić się, że możemy uzyskać obiekt `DateTime` z przesłanej przez użytkownika wartości, a jeżeli się to uda, sprawdzamy, czy data jest datą przyszłą.

- 
- **Wskazówka** Moglibyśmy skorzystać z dołączania modelu, definiując parametr tej metody akcji jako `DateTime`, ale w takim przypadku nasza metoda kontroli poprawności nie będzie wywołana, jeżeli użytkownik poda nieprawidłową wartość, jak na przykład *jablko*. Dzieje się tak, ponieważ łącznik modelu nie będzie w stanie utworzyć obiektu `DateTime` z podanej wartości i zgłosi wyjątek. Funkcja zdalnej kontroli poprawności nie ma możliwości przedstawienia wyjątku, więc zakończy działanie. W takim przypadku pole nie zostanie wyróżnione, więc użytkownik będzie miał wrażenie, że wprowadzona wartość jest prawidłowa. Jako naczelną zasadę w zdalnej kontroli poprawności należy przyjąć, że metody akcji oczekują parametru typu `string` i jawnie wykonują konwersje typów, parsowanie lub dołączanie modelu.
- 

Wyniki kontroli poprawności wyrażamy za pomocą metody `Json`, która tworzy wynik JSON analizowany przez skrypt zdalnej kontroli poprawności działający na serwerze. Jeżeli przetwarzana wartość spełnia nasze wymagania, przekazujemy `true` jako parametr do metody `Json` w następujący sposób:

```
return Json(true, JsonRequestBehavior.AllowGet);
```

Jeżeli wartość nie spełnia naszych oczekiwań, przekazujemy komunikat kontroli poprawności w następujący sposób:

```
return Json("Proszę podać przyszłą datę.", JsonRequestBehavior.AllowGet);
```

W obu przypadkach musimy również przekazać wartość `JsonRequestBehavior.AllowGet` jako parametr. Jest to wymagane, ponieważ biblioteka MVC domyślnie nie pozwala na żądania GET zwracające JSON, więc musimy zmienić to ustawienie. Bez tego parametru żądania kontroli poprawności zostaną bez żadnego komunikatu odrzucone i informacje o błędach nie będą wyświetlane na kliencie.

- 
- **Ostrzeżenie** Metoda akcji kontroli poprawności będzie wywołana przy pierwszym przesłaniu formularza, a następnie za każdym razem, gdy użytkownik zmieni dane. Każde naciśnięcie klawisza spowoduje wysłanie żądania do serwera. W przypadku niektórych aplikacji może to spowodować znaczną liczbę żądań, więc trzeba o tym pamiętać przy określaniu mocy obliczeniowej serwera oraz przepustowości łącza wymaganego dla aplikacji. Dodatkowo można zdecydować, aby *nie* korzystać ze zdalnej kontroli poprawności właściwości, które są zbyt kosztowne (na przykład jeżeli musimy odpytać powolną usługę sieciową w celu sprawdzenia, czy nazwa jest unikatowa).
- 

## Podsumowanie

W rozdziale tym przedstawiliśmy szeroką gamę technik pozwalających na realizowanie kontroli poprawności modelu, dzięki którym możemy upewnić się, że podane przez użytkownika dane spełniają ograniczenia, jakie zdefiniowaliśmy dla naszego modelu danych.

Kontrola poprawności jest ważnym zagadnieniem — utworzenie prawidłowego mechanizmu weryfikacji danych w aplikacji jest niezbędne, aby oszczędzić użytkownikom frustracji przy pracy z aplikacją. Równie ważne jest to, że zachowujemy spójność danych modelu i w naszym systemie nie zapisujemy danych o niskiej jakości.





# Skorowidz

## A

- adresy
  - kwifikowane, 298
  - URL, 165, 311
    - przychodzące, 293
    - wychodzące, 293
  - względne, 298
- agregaty, 74
- Ajax, Asynchronous JavaScript and XML, 515
- akcja asynchroniczna, 393
- akcje podrzędne, 431
- aktualizacje
  - komunikatów, 503
  - testów jednostkowych, 172
- aktywator kontrolerów, 375
- antywzorzec, 70
- anulowanie żądania, 357
- aplikacja
  - smart UI, 69
  - SportsStore, 141, 168, 607
- aplikacje
  - kontrolery, 68
  - modele, 68
  - widoki, 68
- architektura
  - model-widok, 70
  - model-widok-prezenter, 71
  - model-widok-widok-model, 72
  - MVC, 26, 45
  - trójwarstwowa, 71
- asercje, 81
- ASP.NET, 21
- ASP.NET Development Server, 36
- ASP.NET Forms, 235
- ASP.NET MVC, 21
- ASP.NET SQL Server Setup Wizard, 591
- ASP.NET Web Forms, 21, 31
- asynchroniczne
  - metody akcji, 389
  - programowanie, 393
  - zadania, 389
- atak
  - CSRF, 578
  - na witryny, 408
  - XSS, 408, 570, 576
- atrybut
  - AdditionalMetadata, 455
  - Authorize, 237, 586
  - Bind, 466, 582
  - Debug, 259
  - DisplayName, 442, 443
  - enctype, 245, 472
  - HiddenInput, 224, 440
  - HttpPost, 380
  - Locator, 616
  - ModelBinder, 478
  - MustBeTrue, 494
  - NonAction, 380, 582
  - path, 605
  - SessionState, 386
  - UIHint, 445, 452
  - ValidateInput, 573
- atrybuty
  - kontroli poprawności, 492, 508
  - selektora metody akcji, 381
  - testów jednostkowych, 139

automatyczne  
 wnioskowanie typów, 98  
 implementowanie właściwości, 90

automatyzacja  
 interfejsu użytkownika, 87  
 przeglądarki, 87

autoryzacja, 583  
 bazująca na adresach URL, 604

**B**

baza  
 danych uwierzytelniania, 592  
 SQL Server, 591

BDD, 24

bezpieczeństwo, 567

bezpiecznie zakodowany znacznik, 571

bezzstanowa natura HTTP, 29

bezwzględne wyrażenie XPath, 618

biblioteka  
 ASP.NET, 28, 604  
 ASP.NET MVC, 21  
 jQuery, 537  
 jQuery Validation, 505  
 LINQ, 100  
 Moq, 136  
 MVC Framework, 27  
 ORM, 151  
 Rails, 25

blok  
 @section, 426  
 get, 90  
 set, 90  
 try...catch, 65

blokowanie  
 dostępu do cookie, 576  
 wątków, 389

błąd 404, 275  
 kontroli poprawności, 63, 485, 488, 494  
 niejednoznaczności klas, 284  
 niejednoznaczności kontrolera, 309

błędy  
 dołączania modelu, 471  
 weryfikacji, 62

buforowanie, 368

**C**

CDN, Content Delivery Network, 27, 501

Chrome, 543

ciąg połączenia, 619, 620

cookie  
 .ASPXANONYMOUS, 602  
 .ASPXAUTH, 585  
 HttpOnly, 577  
 zapobiegające fałszerstwom, 579

CRUD, 141, 213

czas nieaktywności, 392

czas oczekiwania, 392

**D**

dane  
 identyfikacyjne, 589  
 JSON, 531  
 profilu, 600  
 stronicowania, 162  
 tekstowe, 333  
 ViewState, 27

DDD, 72

debuger, 259

debugowanie, 146, 258–266  
 testów jednostkowych, 266  
 widoków Razor, 262

definiowanie  
 encji koszyka, 185  
 interfejsu, 121, 204  
 klasy metadanych, 447  
 kodu CSS, 167  
 kodu Razor, 109  
 kontrolera, 106  
 metody rozszerzającej, 93  
 modelu, 106  
 prefiksów, 464  
 sekcji, 425  
 struktury danych profilu, 600  
 typu w Ninject, 120  
 użytkownika i hasła, 236  
 widoku w sekcjach, 427  
 właściwości, 89, 90  
 zasad poprawności, 491

degradowanie łączы Ajax, 526

DI, dependency injection, 77

dodawanie  
 ASP.NET MVC, 609  
 atrybutów HTML, 571  
 atrybutu Authorize, 237  
 danych do bazy, 154  
 danych modelu widoku, 161  
 DI do aplikacji, 266  
 elementów do DOM, 553  
 elementów konfiguracji, 612  
 elementów script, 518

- filtra, 347
  - filtrów do akcji, 363
  - funkcji do roli serwera WWW, 585
  - imitacji implementacji, 148
  - klasy modelu, 53
  - kontrolek nawigacji, 171
  - kontrolera, 45, 148
  - kontrolera do obszaru, 308
  - kontroli poprawności, 61
  - łącza do formularza, 53
  - łącza do widoku, 54
  - łączy Ajax, 525
  - metadanych, 445
  - metody akcji, 149
  - metody akcji do kontrolera, 54
  - metody do interfejsu repozytorium, 225
  - modelu widoku, 159
  - modelu widoku ProductsListViewModel, 161
  - Moq, 135
  - obsługi kategorii, 172
  - obsługi strony klienckiej, 508
  - obszaru, 306
  - pakietu EntityFramework, 155
  - podsumowania koszyka, 198
  - powiązań Ninject, 119, 239
  - powiązań Ninject dla IOrderProcessor, 206
  - powiązania z repozytorium, 156
  - procesu zamawiania, 201
  - przestrzeni nazw, 161, 407
  - przycisku usuwania, 197
  - przycisku Zamówienie, 201
  - punktu zatrzymania, 261
  - referencji, 143
  - roli do Windows Server, 38
  - silnie typowanego widoku, 54
  - stronicowania, 157
  - stylu, 165
  - ścieżki, 291
  - treści, 50, 554
  - treści dynamicznych, 406
  - widoku, 59
    - Checkout, 202
    - HTML, 52
    - Index, 108, 192
    - List, 150
    - LogOn, 241
    - Summary, 199
  - właściwości, 122
  - właściwości do klasy Product, 244
  - zależności, 121
  - zasad CSS, 167
  - zasad kontroli poprawności, 510
  - zdjęcia, 248
  - znacznika @using, 407
  - dołączanie
    - danych, 461
    - klasy, 125
    - kolekcji
      - indeksowanych, 468
      - o niesekwencyjnych indeksach, 468
      - typów niestandardowych, 467
    - modelu, 59, 72, 193, 461, 472
    - selektywne właściwości, 466
    - tablic i kolekcji, 467
    - typów prostych, 463
    - typów złożonych, 464
  - domena, 68
  - domena aplikacji, 52
  - domyślny łącznik modelu, 462
  - dopasowanie
    - adresów URL, 272, 277, 282
    - ścieżek, 294
  - dostawca
    - danych o członkostwie, 589, 596
    - danych profilu, 602
    - kontroli poprawności, 498
    - łączników modelu, 477
    - metadanych modelu, 457
    - ról, 599
    - wartości, 318
  - dostęp do
    - cookie, 576
    - dostawców wartości, 476
    - funkcji administracyjnych, 569
    - funkcji silnika widoku, 396
  - dynamiczna
    - kompilacja stron, 608
    - zmiana widoczności, 563
  - dynamiczne tworzenie
    - treści, 50
    - zapytań SQL, 580
  - dynamicznie typowany widok, 410
  - dystrybucja aplikacji, 607
  - dziedziczenie
    - po klasie DefaultModelBinder, 489
    - własnego dostawcy metadanych, 459
- ## E
- edycja produktu, 229
  - edycja żądania HTTP, 569
  - EF, Entity Framework, 151

efekty wizualne jQuery, 556  
 efektywność wykorzystania łącza, 30  
 elementy  
   ipSecurity, 605  
   projektu MVC 3, 255, 256  
   Select, 418  
 encja Cart, 185

## F

fabryka  
   dostawcy wartości, 474  
   kontrolerów, 372, 375  
   kontrolerów wbudowana, 374  
 fałszowanie żądań, CSRF, 567, 577  
 fałszywe poczucie bezpieczeństwa, 573  
 Fiddler, 569  
 filtr, 237, 345  
   akcji, 357, 363  
   Authorize, 346  
   autoryzacji, 579  
   wbudowany, 351  
   własny, 350  
   globalny, 365  
   HandleErrorAttribute, 355  
   OutputCache, 366, 368  
   parametrów It, 137  
   RequireHttps, 366  
   uwierzytelniania, 236, 345  
   wbudowany wyjątków, 354  
   wyniku, 359  
 filtrowanie  
   bez użycia atrybutów, 361  
   kolekcji obiektów, 96  
   listy produktów, 171  
   według kategorii, 173, 174  
 filtrująca metoda rozszerzająca, 96  
 filtry  
   akcji i wyniku, 356  
   autoryzacji, 348  
   formularzy jQuery, 548  
   globalne, 362  
   jQuery, 547  
   parametrów Moq, 136  
   treści jQuery, 548  
   wbudowane, 366  
   wyjątków, 353  
 Firebug, 542, 569  
 Firefox, 542  
 formatowanie danych, 454

formularz  
   HTML, 55  
   serwerowy, 57  
   wysyłki, 203  
 funkcja  
   Edit and Continue, 264  
   inicjalizatora obiektów, 92  
   jQuery(), 544  
   kontroli poprawności, 227  
   o nazwie code-first, 154  
   publikowania, 628  
   stronicowania, 158  
   ViewBag, 112, 264  
   ViewData, 264  
   Visual Web Developer, 34  
 funkcje  
   aplikacji, 151  
   dla wywołań zwrotnych, 527  
   filtrów, 361  
   JavaScript, 527  
   kontenera DI, 79  
   kontroli poprawności, 510  
   przycisków jQuery UI, 559

## G

generowanie  
   adresu URL, 298–300, 419  
   edytora, 439  
   elementów HTML, 437  
   kodu HTML, 436  
   liczby produktów, 183  
   listy kategorii, 177, 178  
   łączy, 418  
   rusztowania, 437  
   sekcji opcjonalnych, 428  
   szablonów, 449  
   tabeli, 215  
   widoków, 48, 59, 109  
   wychodzących adresów URL, 294, 299, 303  
   wyników, 319  
   wyniku akcji, 322  
   znaczników HTML, 438  
 generyczne typy parametrów, 137  
 gettery, 90  
 globalne priorytety, 375  
 główna encja, 74  
 graficzny interfejs użytkownika, GUI, 21

## H

HTTP, 568

## I

IDE, zintegrowane środowisko programistyczne, 117  
 identyfikatory elementów, 546  
 IIS, Internet Information Services, 38  
 IIS 7.5, 37  
 IIS Express, 36  
 implementacja
 

- filtra wyjątku, 354
- filtra wyniku, 358
- funkcji, 134
- GetImage, 246
- IControllerActivator, 376
- IDependencyResolver, 266
- interfejsu, 77, 94, 204
- IPriceReducer, 128
- IValueCalculator, 125
- IValueProvider, 470
- IView, 397
- IViewEngine, 397
- metody
  - AddBid, 85
  - GetVirtualPath, 304
  - Menu, 177
  - SaveProduct, 226
- MVC, 69
- własnego filtra autoryzacji, 352
- widoku listy, 218

 importowanie przestrzeni nazw, 406  
 informacje o błędach systemu, 210  
 inicjalizator obiektu, 91  
 inicjowanie kolekcji, 92  
 instalacja bazy danych, 619  
 instalacja binarna, 609  
 instalowanie
 

- aplikacji MVC, 37
- dotychczasowych komponentów, 38
- opcjonalnych komponentów, 35
- pakietu, 624–626
- podstawowego oprogramowania, 34
- Visual Studio 2010, 33

 instancje baz danych użytkowników, 590  
 instrukcja using, 160  
 IntelliSense, 56  
 IntelliSense dla jQuery, 541  
 interakcje w aplikacji MVC, 69  
 interfejs
 

- IActionFilter, 356
- IActionInvoker, 377
- IAuthorizationFilter, 348
- IAuthProvider, 238

ICalculator, 403  
 IClientValidatable, 508  
 IController, 313  
 IControllerActivator, 375  
 IControllerFactory, 372, 373  
 IDiscountHelper, 121  
 IEmailSender, 79  
 IEmailService, 77  
 IEnumerable<T>, 96  
 IExceptionHandler, 353  
 IMembersRepository, 78, 81  
 IModelBinder, 194, 462  
 IOrderProcessor, 204  
 IProductRepository, 147, 156  
 IResultFilter, 358  
 IRouteConstraint, 288  
 IRouteHandler, 304  
 IValidatableObject, 495  
 IValueCalculator, 118  
 IValueProvider, 473  
 IView, 396  
 IViewEngine, 395  
 System.Web.Mvc.IController, 314  
 użytkownika, UI, 21  
 użytkownika typu CRUD, 213  
 IoC, inversion of control, 77

## J

jawna kontrola poprawności modelu, 481  
 język C#, 89  
 jQuery, 537
 

- filtry, 547
- filtry formularzy, 548
- filtry treści, 548
- metody, 549
- metody CSS, 552
- metody efektów wizualnych, 558
- metody przeciążone, 551
- selektory, 545
- selektory atrybutów, 546

 jQuery UI, 558  
 JSON, JavaScript Object Notation, 529

## K

katalog
 

- .NET Framework, 591
- Abstract, 204
- Admin, 306
- App\_Data, 591

## katalog

- Concrete, 204
- Content, 217
- Controllers, 45, 314
- Derived, 315
- Infrastructure, 302
- Models, 53, 239
- scripts, 501
- Shared, 113, 218
- Views, 48

katalogi wirtualne, 621

kierowanie żądań, 303

klamry, 92

## klasa

- AccountController, 240
- ActionExecutedContext, 358
- ActionExecutingContext, 356
- ActionFilterAttribute, 347, 360
- ActionMethodSelectorAttribute, 381
- ActionResult, 50
- AdminAreaRegistration, 306
- AdminController, 78, 80, 215, 245
- AjaxOptions, 520, 527
- Appointment, 479, 502, 515
- AppointmentController, 480, 516
- AreaRegistrationContext, 307
- AuthorizationContext, 350
- BasicController, 314
- Cart, 186
- CartController, 193, 206
- CartItemViewModel, 191
- CartItemBinder, 194
- Chart, 423
- Controller, 315
  - filtry, 315
  - metody akcji, 315
  - wynik akcji, 315
- Controller. ViewBag, 50
- ControllerContext, 194, 350
- CustomControllerFactory, 372
- DataAnnotationsModelMetadataProvider, 459
- DefaultControllerFactory, 374, 375
- DefaultDiscountHelper, 121
- DefaultModelBinder, 462, 463, 489
- EfDbContext, 155
- EFProductRepository, 156
- EmailOrderProcessor, 204
- EmailSettings, 206
- ExceptionContext, 353
- FakeRepository, 129, 135
- FormCollection, 471
- FormsAuthentication, 238
- FormsAuthProvider, 238

GuestResponse, 53

HandleErrorAttribute, 355

HomeController, 46

IControllerFactory, 385

It, 137

JsonResult, 336

LegacyController, 301

LegacyRoute, 302

LinqValueCalculator, 120

LogOnViewModel, 239

ModelMetadata, 496

MvcHtmlString, 572

MyEmailSender, 77

MyPriceReducer, 129

NinjectControllerFactory, 126, 144, 145, 156, 239

PagingHelpers, 159

PagingInfo, 159

PasswordResetHelper, 77

Product, 119, 147

ProductController, 148

ProductsListViewModel, 171

RedirectResult, 321

RoleProvider, 599

RouteCollection, 271

RouteData, 303

ShippingDetails, 200, 202

ShoppingCart, 93, 94, 118, 124

SqlProfileProvider, 600, 601

SqlRoleProvider, 597

System.Activator, 475

System.Console, 89

System.Web.Mvc.Controller, 315

System.Web.Mvc.RedirectResult, 321

TagBuilder, 413

TemplateInfo, 454

ViewData, 330

ViewEngineResult, 396

ViewResult, 227

WebGrid, 419, 420, 422

WebMail, 64

ModelMetadata, 456

FormsAuthentication, 238

## klasy

bazowe kontrolerów, 45

dostawcy członkostwa, 595

dostawcy kontroli poprawności, 496

fabryki dostawców wartości, 473

modelu, 435, 450

zagnieżdżone, 464

złożone, 464

osprzętu testów, 80

pomocnicze, 419, 423

pomocnicze wbudowane, 424

- repozytorium, 156
    - z testami jednostkowymi, 131
    - zwracająca skrypt, 408
  - klucz UseNamespaceFallback, 286
  - kod
    - HTML, 538
    - HTTP 301, 331
    - HTTP 302, 331
    - źródłowy biblioteki MVC, 36, 41
  - kodowanie
    - ciągów znaków, 408, 574
    - HTML, 409
    - HTML w Razor, 571
  - kolejność
    - ścieżek, 279
    - widoków, 481
    - wykonywania filtrów, 364
  - kompilacja, 54
  - komunikat
    - kontroli poprawności, 488
    - o błędzie, 355, 400, 486, 503
    - o błędzie 404, 586
  - konfigurowanie
    - .NET 4, 624
    - Debug, 259
    - dostawcy członkostwa, 589, 590
    - IIS, 622
    - kontenera DI, 144
    - Moq, 136
    - Ninject, 120
    - nowej witryny, 623
    - ograniczeń, 605
    - profilu, 600
    - routingu, 47, 309
    - ról, 597
    - SqlMembershipProvider, 590
    - SqlProfileProvider, 600
    - SqlRoleProvider, 597
    - ścieżki, 150, 628
    - uwierzytelniania
      - Forms, 235, 586, 588
      - w IIS, 584
    - Web Deployment, 39
    - wyszukiwania lokalizacji widoków, 404
    - zachowania, 136
  - konflikt
    - nazw, 309
    - powiązań, 624
  - konsola JavaScript w Chromie, 544
  - kontener
    - DI, 79, 117
    - ViewBag, 227
  - kontrola
    - plików, 290
    - poprawności, 61, 72, 210, 229, 484
    - danych modelu, 479
    - na poziomie modelu, 490
    - po stronie klienta, 231, 242, 499–506, 510
    - w łączniku modelu, 488
    - zadań, 27, 572
  - kontroler, 45, 68, 313
    - AccountController, 239, 256
    - asynchroniczny, 388
    - HomeController, 256
    - koszyka, 189
    - obsługujący obiekty JSON, 534
    - ProductController, 107
    - UserProfileController, 577
  - kontrolery
    - asynchroniczne, 386, 387, 394
    - bezzstanowe, 385
    - dziedziczące, 319
  - kontrolka
    - GridView, 164
    - Slider, 560
  - kontrolki, 69
  - konwencje
    - dla klas kontrolerów, 257
    - dla układów, 258
    - dla widoków, 257
    - MVC, 257
  - kopiowanie elementów script, 540
  - koszyk, 184
- L**
- licznik stron, 182
  - LINQ, Language Integrated Query, 29, 100
  - LINQ to Entities, 106
  - lista kategorii, 177
  - logika
    - aplikacji, 406
    - prezentacji, 406
    - proceduralna, 406
  - lokalne proxy, 569
  - luźne powiązanie, 76
- Ł**
- ładowanie
    - jQuery, 542
    - pakietu instalacyjnego, 627

## łącza

- do akcji, 310
- kategorii, 180
- niewłaściwe, 183

łącze Enable editing, 265

## łączenie

- metod akcji, 53
- metod rozszerzających, 97
- segmentów URL, 278

## łączenie

- selektorów, 545
- selektorów atrybutów, 546
- warunkowe, 125
- ze słownikiem, 469
- żądań z kontrolerami, 373

łącznik domyślny modelu, 475

łącznik modelu, 193, 318, 472, 475

**M**

## manipulowanie

- atributami elementów konfiguracji, 615
- klasami CSS, 552
- modelem DOM, 552

## mechanizm

- akcji potomnych, 175
- filtrów, 236
- ViewBag, 180, 328

mechanizmy członkostwa, 589, 600

menedżer IIS, 39, 586, 588

menu nawigacji, 175

metadane, 224

metadane modelu, 72, 440

## metoda

- ActionLink, 53, 294
- AddBid, 83, 84
- AddBinding, 206
- AddBindings, 127, 156
- addClass, 551
- AddSeries, 424
- Ajax.ActionLink, 525
- Ajax.BeginForm, 521
- All\_Prices\_Are\_Changed, 132
- Application\_Start, 271, 307
- AreaRegistration, 307
- Async, 389
- AuthorizeCore, 351
- BeginForm, 189
- BindModel, 462
- button, 560
- CanChangeLoginName, 81
- CannotAddLowerBid, 84

Change, 581

ChangeLoginName, 80

Checkout, 202

ChildAction, 368

Completed, 389

Create, 232

CreateController, 373

CreateMetadata, 458, 459

DataCompleted, 391

Delete, 233, 235

DoPasswordChange(), 582

Edit, 221, 578

EditorForModel, 224

ExecuteResult, 324

File, 338

FilterByCategory, 97

FindView, 400

FormsAuthentication.Decrypt, 585

GetCart, 194

GetControllerInstance, 127

GetControllerSessionBehavior, 373

GetConverter, 447

GetImage, 246

GetLegacyURL, 301

GetProducts, 136

GetRemoteData, 388

GetRolesForUser, 599

GetValidation, 496

GetValue, 476

HandleUnauthorizedRequest, 351

HandleUnknownAction, 382

Html.ActionLink, 298

Html.AntiForgeryToken(), 579

Html.AntiForgeryToken(), 579

Html.BeginForm, 56

Html.EditorFor, 203, 436

Html.EditorForModel, 441

Html.ValidationSummary, 62, 242

IgnoreRoute, 292

Index, 191, 215, 216

Is<T>, 137

JavaScriptStringEncode, 575

Json, 512

List, 161

LogOn, 239

MapRoute, 273, 286

Match, 289

Menu, 176

ModelState.IsValidField, 485

Ninject WithPropertyValue, 123

OnActionExecuted, 357

OnActionExecuting, 356



OnAuthorization, 351  
 OnException, 353  
 OrderByDescending, 102  
 Partial, 428  
 RegisterArea, 306  
 RegisterGlobalFilters, 362  
 RegisterRoutes, 107, 271, 291, 293  
 ReleaseController, 373  
 RenderAction, 176, 177  
 RenderBody, 426  
 RenderPartial, 169  
 SaveProduct, 226  
 slider, 562  
 string.IsNullOrEmpty, 482  
 Summary, 198  
 UpdateModel, 469  
 UpdateProduct, 139  
 Url.Action, 174, 522  
 UrlHelper.GenerateURL, 295  
 ValidationSummary, 484  
 View, 324  
 WebGrid.Column, 421  
 WebGrid.GetHtml, 421  
 WebMail.Send, 65  
 WebRequest.BeginGetResponse, 394  
 WithPropertyValue, 123

#### metody

akcji, 45, 69  
 akcji kontroli poprawności, 513  
 CSS z jQuery, 552  
 dołączania warunkowego, 126  
 efektów wizualnych z jQuery, 558  
 jQuery, 548  
 klasy DefaultControllerFactory, 377  
 klasy DefaultModelBinder, 489  
 metody kontroli poprawności, 511  
 mające parametry generyczne, 379  
 manipulowania modelem DOM, 555  
 nawigowania w modelu DOM, 555  
 obsługi zdarzeń, 69  
 osprzętu testów, 83  
 pomocnicze, 422  
 pomocnicze
 

- do wprowadzania danych, 415
- HTML, 410, 418
- szablonów rusztowań, 438
- wbudowane, 413
- wewnętrzne, 411
- widoku szablonowego, 435, 438
- zewnętrzne, 411

 rozszerzające, 28, 93, 94  
 rozszerzające LINQ, 102, 103

rozszerzające opóźnione, 102  
 statyczne klasy Assert, 133  
 statyczne klasy It, 137  
 szablonowe, 436  
 testowe, 81  
 uznane za akcje, 378  
 zapobiegające fałszerstwom, 579  
 międzywytynnowe fałszowanie żądań, CSRF, 567, 577  
 migracja z Web Forms do MVC, 30  
 model
 

- domeny, 52, 68, 73
- domeny aukcji, 75
- systemu z agregatami, 74
- widoku, 327

 modelowanie domeny, 72  
 moduł administracyjny, 141  
 modyfikowanie
 

- danych POST, 569
- DOM, 569
- domyślnego widoku Razor, 166

 MonoRail, 30  
 Moq, 135  
 MSDN, Microsoft Developer Network, 41  
 MVC, model-widok-kontroler, 45, 67

## N

nadawanie stylu polom wyboru, 483  
 nadpisywanie metod HTTP, 384  
 nagłówek Referer, 588  
 narzędzia
 

- automatyzacji UI, 24
- testów jednostkowych, 24

 narzędzie
 

- IIS, 592
- imitujące, 117
- Kompilacja, 609
- Menedżer serwera, 584
- ORM, 30
- testów automatycznych, 30
- ThemeRoller, 559
- WAT, 592, 593
- WebPI, 34

 nawiasy klamrowe, 99  
 nazwa widoku, 50  
 nazwy ścieżek, 272  
 nazwy własne akcji, 379  
 niejawnie typowanie, 98  
 niejednoznaczność
 

- kontrolerów, 308
- metod akcji, 382

nieprzeszkadzające wywołania Ajax, 515  
 sposób działania, 520  
 włączanie obsługi, 518

Ninject, 117

Node.js, 25

## O

obiekt

Cart, 195

domeny, 68

EFDbContext, 156

EmailSettings, 206

ExceptionContext, 353

FormsAuthenticationTicket, 585

HtmlHelper, 412

kernel Ninject, 119

kontekstu, 316, 317

kontroli poprawności, 497, 498

ModelData, 497

modelu, 109

modelu widoku, 327, 356

MvcHtmlString, 413

MvcRouteHendler, 272

obsługi ścieżki, 305

Product, 97

Route, 286, 295

RouteData, 303

TemplateInfo, 455

ViewBag, 50

ViewEngineResult, 396

wbudowany wywołujący akcje, 378

wywołujący akcje, 377, 462

zarządzający zależnościami, 375

obrony przed atakami CSRF, 578

obsługa

adresów URL, 293

błędów, 471

danych zdjęcia, 245

degradacji, 526

formularzy, 57

JSON, 530

komunikatu w pliku szablonu, 229

kontroli poprawności, 512

nieznanych akcji, 382

prób logowania, 586

stanu, 29

ścieżki, 304

usług sieciowych REST, 383

właściwości, 447, 448

wyjątków, 262, 392

zależności, 78

żądania POST, 58

żądań, 383

żądań JSON, 532

obszar MVC, 305

odbieranie danych JSON, 533

odczyt

danych

z ViewBag, 113, 329

z ViewData, 330

zdjęć, 246

odwołanie do jQuery, 540

odwrócenie kontroli, 77

odzworowanie obiektowo-relacyjne, 581

ograniczanie

adresów IP, 605

dołączania, 470

dostępu, 604, 605

dostępu do lokalizacji, 605

łącnika do danych formularza, 470

ograniczenia niestandardowe, 289

ograniczenia ścieżek, 286

okno

Add Controller, 214

Add View, 49

Configuration Manager, 619

New ASP.NET MVC 3 Project, 254

New Project, 142

obsługi wyjątku, 263

Publish Web, 629

Server Explorer, 152

Solution Explorer, 144, 148

Test Results, 134

ominięcie kodowania danych, 572

ominięcie kontroli poprawności, 569

opcja

compilation, 608

Edit and Continue, 263

MvcBuildViews, 608

requireSSL, 586

operacje na domenie, 68

operator @Model, 60

operator Href, 63

OSI, Open Source Initiative, 29

osprzęt testu, 80

## P

parametr

Cart, 207

ControllerContext, 194

fileDownloadName, 338

Func, 97

- konstruktora, 123
- ModelBindingContext, 194
- object, 494
- opcjonalny, 157
- out, 318
- ref, 318
- ShippingDetails, 207
- typu generycznego, 410
- parametry
  - filtra OutputCache, 367
  - metody CreateMetadata, 458
  - metody WebGrid.Column, 421
  - metody WebGrid.GetHtml, 421
  - obowiązkowe, 318
  - opcjonalne, 318
  - przekazywane do metody File, 338
- platforma testów jednostkowych, 117
- plik
  - \_AdminLayout.cshtml, 216, 218, 227, 231
  - \_Layout.cshtml, 113, 114, 167
  - \_viewstart, 113
  - \_ViewStart.cshtml, 113
  - Admin.css, 217
  - Administration.config, 596
  - Global.asax, 107, 231, 270, 307
  - Global.asax.cs, 47, 172
  - Index.cshtml, 48, 219
  - jquery.unobtrusive-ajax.js, 518
  - jquery-1.5.1.min.js, 518
  - List.cshtml, 163
  - MyView.cshtml, 315
  - Site.css, 167, 179
  - StaticContent.html, 290
  - układu, 227
  - Web.config, 155, 160, 206, 407, 586, 610, 612
  - Web.Debug.config, 610
  - Web.Release.config, 610
- pliki
  - .cshtml, 49
  - .mdf, 590
  - .vbhtml, 404
  - jQuery, 539
  - przekształceń Web.config, 610
  - wyszukiwane przez Razor, 404
  - .eml, 206
- pobieranie
  - danych, 316
  - pobieranie danych z ViewBag, 51
  - podsumowanie instalacji, 628
  - pole PageSize, 157
  - połączenie z bazą danych, 155
  - połączenie z SQL Server Express, 156
  - powiązanie z repozytorium, 156
  - powrót do właściwości standardowej, 91
  - prefiksy HTML, 454
  - priorytety kontrolerów, 284
  - priorytety przestrzeni nazw, 374
  - proces dołączania modelu, 469
  - profile, 589, 600
  - profile anonimowe, 601
  - program
    - aspnet\_regsql.exe, 591
    - Fiddler, 569
    - Menedżer IIS, 598
    - telnet, 569
  - programowanie
    - .NET, 41
    - asynchroniczne, 393
    - sterowane domeną, 72
    - sterowane testami, TDD, 24, 82
    - sterowane zachowaniami, BDD, 24
    - zwinne, 25
  - projekt
    - ControllersAndActions, 313
    - DebuggingDemo, 259
    - MvcApp, 515
    - Razor MVC, 160
    - SportsStore, 143
    - SportsStore.Domain, 156, 200
    - SportsStore.WebUI, 156, 161, 179
    - WorkingWithAreas, 306
  - prototypowanie interfejsu użytkownika, 70
  - przechwytywanie żądania, 291
  - przejęcie sesji, 576
  - przekazywanie danych, 112
    - metadanych, 455
    - parametrów, 391
    - z metody akcji do widoku, 327
    - z użyciem ViewBag, 328
    - z użyciem ViewData, 329
  - przekierowanie, 238, 331, 332
    - do adresu URL, 300
    - do innej akcji, 299
    - trwale, 331
    - z użyciem literału znakowego, 332
    - żądania URL, 290
  - przekształcenie
    - InsertAfter, 613
    - InsertBefore, 613
    - Remove, 614
    - RemoveAll, 614
  - przerywanie operacji asynchronicznej, 393
  - przesyłanie metod HTTP, 384

przestrzenie nazw, 285  
 PartyInvites.Models, 58  
 System.Linq, 102  
 System.Web.Mvc, 224  
 przesyłanie danych edycji, 228  
 przesyłanie zdjęć produktów, 242  
 przeszukiwanie danych parametrów, 462  
 przetwarzanie JSON, 531, 532  
 przetwarzanie ścieżek, 290  
 przycisk, 560  
 Dodaj do koszyka, 184  
 Kontynuuj zakupy, 193  
 Publish, 629  
 Zamówienie, 201  
 przykłady użycia jQuery, 544  
 publikowanie jednym kliknięciem, 628  
 pule aplikacji, 621  
 punkty zatrzymania, 260

## R

raportowanie kategorii, 181  
 referencja do System.Web.Mvc, 224  
 rejestrowanie  
 aktywatora kontrolerów, 376  
 dostawcy ról, 599  
 dostawcy uwierzytelniania, 596  
 implementacji, 206  
 implementacji RouteBase, 303  
 klasy CartModelBinder, 194  
 klasy NinjectControllerFactory, 145  
 klasy w bibliotece, 127  
 ścieżki, 272  
 własnego dostawcy metadanych, 458  
 własnego dostawcy profili, 603  
 własnego silnika widoku, 398  
 własnej fabryki kontrolerów, 374  
 relacje pomiędzy typami, 119  
 repozytoria, 75  
 reprezentacja JSON, 529  
 REST, Representational State Transfer, 24  
 role, 589  
 role w aplikacjach MVC, 597  
 routing żądań dla plików dyskowych, 290, 291  
 rozdzielanie komponentów, 76  
 rozszerzalność, 26  
 rozszerzanie modelu domeny, 244  
 Ruby on Rails, 25, 30  
 rzutowanie parametru, 301

## S

samodzielne łączenie, 124  
 samokontrolująca się klasa modelu, 495  
 schemat  
 adresów URL, 310  
 bazy danych, 153  
 segment  
 mieszany, 278  
 opcjonalny, 282  
 własny, 280  
 segmenty adresu URL, 271, 277  
 sekcja authentication, 236  
 sekcje opcjonalne, 428  
 sekcje w pliku układu, 425  
 selektor \$(this), 553  
 selektory jQuery, 545  
 Selenium RC, 87  
 separacja zadań, 75, 406  
 serwer IIS, 38, 621  
 serwer WWW, 36  
 settery, 90  
 sfalszowane dane, 567  
 silnie typowane  
 kontrolki, 416  
 metody pomocnicze, 417  
 widoki, 409  
 widoki częściowe, 430  
 silnik  
 ASPX, 43, 395, 401  
 Brail, 400  
 NHaml, 400  
 NVelocity, 400  
 Razor, 30, 43, 105, 395, 401  
 Spark, 400  
 Sinatra, 25  
 składane adresy URL, 165  
 składanie zamówień, 200  
 składnia silnika Razor, 105  
 składnia zapytania, 101  
 składniki klasy  
 ModelMetadata, 456  
 TemplateInfo, 454  
 skrypt jQuery, 533  
 skrypty międzywytynowe XSS, 408, 567  
 słowo kluczowe  
 model, 327  
 select, 101  
 this, 93  
 using, 102  
 var, 98, 100  
 yield, 96

słowo nieprzeszkadzający, 499  
 sortowanie, 101  
 sprawdzanie
 

- adresu IP klienta, 576
- istnienia sekcji, 427
- konfiguracji, 258
- uwierzytelniania, 345

 SQL Server, 36, 151  
 stan sesji, 385  
 standardy WWW, 24  
 sterowanie zawartością tabeli, 561  
 stosowanie filtrów, 348  
 stosowanie metody rozszerzającej, 95  
 stronicowanie, 157  
 style CSS, 482
 

- dla elementów edytora, 225
- dla widoków administracyjnych, 217

 system
 

- dołączania danych, 472
- routingu, 47, 269, 290
- zarządzania migracjami, 30

 szablon
 

- CRUD, 214
- DateTime.cshtml, 454
- Empty, 254
- Internet Application, 80, 254
- Intranet Application, 254, 583
- MVC Internet Application, 236
- Object, 447

 szablon
 

- projektu, 44
- rusztowań, 437
- wbudowane widoku, 446
- widoku edytora, 450, 452
- wyświetlania, 451

 szkielec klasy MyPriceReducer, 129

## Ś

ścieżka domyślna, 107, 150, 461  
 ścieżki, 175, 272
 

- nazwane, 300
- o zmiennej długości, 282

 środowisko
 

- niezależnych dostawców oprogramowania, 24
- testowe dla jQuery, 542

## T

TDD, 82, 86  
 techniki instalacji, 607

technologia
 

- „przeciągnij i upuść”, 30
- Ajax, 25
- ASP.NET, 21
- ASP.NET Web Forms, 21

 technologie programowania WWW, 22  
 telnet, 569  
 test jednostkowy
 

- adresy URL, 273
- akcja Index, 216
- dane stronicowania, 162
- filtrowanie według kategorii, 174
- generowanie listy kategorii, 178
- generowanie widoku, 325
- kody statusu HTTP, 341
- kontroler koszyka, 196
- łącza stron, 160
- metoda Edit, 222
- obiekty modelu widoku, 328
- odczyt zdjęć, 246
- ograniczenia ścieżek, 288
- opcjonalne segmenty URL, 283
- przekierowania, 333
- przekierowanie, 332
- przesyłanie danych, 228
- raportowanie kategorii, 181
- segmenty statyczne, 279
- stronicowanie, 158
- testowanie koszyka, 186
- usuwanie produktów, 234
- uwierzytelnianie, 243
- ViewBag, 329
- ViewData, 330
- wartości domyślne, 277
- wychodzące adresy URL, 295
- wyniki plikowe, 340
- zliczanie produktów, 183
- zmiennych segmentów przechwytyjących, 284
- zmiennych własnych segmentów, 281
- zwracanie wyników, 335
- funkcji stronicowania, 158
- integracyjne, 87
- jednostkowe, 128
- jednostkowe kontrolerów i akcji, 322
- klasy CartController, 196, 208
- kontrolera AccountController, 243
- PageLinks, 160
- regresyjne, 80
- segmentów statycznych, 279

 testowanie
 

- widoku, 50
- wzorca URL, 276

## testy

- automatyczne, 80
- integracyjne, 80, 87
- jednostkowe, 80, 85, 132
- jednostkowe z użyciem Moq, 137

treści dynamiczne, 405

treści statyczne, 405

## tryb

- Condition, 618
- debug, 608

tryby atrybutu Locator, 617

tryby uwierzytelniania IIS, 584

## tworzenie

- abstrakcyjnego repozytorium, 147
- agregatów, 75
- akcji podrzędnych, 431
- aplikacji SportsStore, 242
- atrybutu kontroli poprawności, 492, 505
- bazy danych, 152
- danych JSON, 337
- danych stronicowania, 182
- danych wyjściowych, 319
- domyślnych ścieżek, 47
- dostawcy
  - kontroli poprawności, 495
  - łączników modelu, 477
  - metadanych, 457
  - profilu, 602
  - ról, 599
  - uwierzytelniania, 238
  - wartości, 473
- elementów, 554
- elementów HTML, 437
- fabryki kontrolerów, 372
- filtra autoryzacji, 349
- filtra globalnego, 363
- filtra wyjątku, 353
- filtrów akcji, 359
- filtrów wyniku, 359
- filtrujących metod rozszerzających, 96
- formularza HTML, 519
- formularzy, 413, 414
- imitacji, 135
- imitacji repozytorium, 148
- implementacji interfejsu, 121
- implementacji RouteBase, 301
- interfejsu użytkownika, 244
- klasy dziedziczącej, 124
- klasy modelu domeny, 106
- klasy Product, 147
- kodu jQuery, 541
- kodu zabezpieczeń, 349
- kont dla użytkowników, 584

kontekstu Entity Framework, 154

kontrolera, 107, 313

AccountController, 239

asynchronicznego, 387

CRUD, 214

koszyka, 189

nawigacji, 176

koszyka, 184

łańcucha zależności, 121

łącznika modelu, 193, 474, 475

łączy Ajax, 524

łączy stron, 160

metody akcji, 54

metody akcji Edit, 221

metody pomocniczej, 411

modelu domeny, 146

obiektów, 91

obiektu ViewResult, 326

obszaru, 306

ograniczenia ścieżki, 289

pakietu instalacyjnego, 626

parametrów, 391

pliku Admin.css, 218

pliku układu, 216

pomocniczej aplikacji, 594

produktów, 232

projektu, 43, 105, 119, 128

projektu DebuggingDemo, 259

projektu routingu, 270

projektu testów jednostkowych, 143

prostych adresów URL, 28

przycisków, 559

przycisków koszyka, 188

przyjaznych adresów URL, 310

pustego rozwiązania, 142

repozytorium produktów, 156

sekwencji obiektów, 419

silnika widoku, 395

szablonów, 447

szablonu edytora, 448

szablonu ogólnego, 452

szablonu wyświetlania, 450

ścieżki, 272

tablicy, 93

testowych obiektów danych, 138

testów jednostkowych, 130

typu anonimowego, 99

układu Razor, 217

widoku, 47, 48, 107, 240

Completed, 211

częściowego, 167, 177, 428, 449

Edit, 223

- edycji, 221
- Index, 219
  - z formularzem, 55
- wielu formularzy, 189
- własnego dostawcy danych członkostwa, 595
- własnego wyniku akcji, 341
- własnych zasad autoryzacji, 351
- własnych zasad braku autoryzacji, 352
- wykresu, 424
- wyrażeń XPath, 618
- zasad poprawności, 509
- znaczników select, 417

tylne wejście, 582

typ

- ActionResult, 323
- DataType, 444
- JsonRequestBehavior, 530
- nullable, 61

typy

- anonimowe, 29, 99
- dynamiczne, 29
- filtrów, 347
- literalowe, 319
- wyliczeniowe, 385, 444, 530

## U

- udostępnianie plików, 256
- układ Razor, 114
- ukrycie skryptu, 572
- ukrywanie elementów <a>, 543
- ukrywanie łączy, 543
- ulepszanie URL, 172
- uproszczona metoda akcji, 490
- uruchamianie aplikacji, 151
- uruchamianie testów, 133
- usługa
  - hostingowa, 37
  - roli ASP.NET, 38
  - roli Usługa zarządzania, 38
  - zarządzania, 41
- ustawienie Layout, 115
- usunięcie
  - atrybutu debug, 611
  - czasu oczekiwania, 392
  - wywołania ViewBag, 265
- usuwanie
  - atrybutów, 615
  - elementów konfiguracji, 614
  - produktów, 232, 235
  - produktów z koszyka, 197

- sygnalizacji błędów, 504
- zależności, 77
- uwierzytelnianie, 583
  - bez cookie, 587
  - Forms, 236, 576, 585
  - Windows, 236, 583
- użycie
  - atrybutu Locator, 617
  - metadanych, 440–444
  - metod CSS z jQuery, 550
  - przekształceń atrybutów, 615
  - zdarzeń jQuery, 555

## V

- Visual Studio, 142
  - obsługa HTML5, 254
- Visual Web Developer 2010 Express, 34

## W

- W3C, Wide Web Consortium, 311
- wady
  - ścieżek nazwanych, 300
  - Web Forms, 23
- wartości
  - domyślne parametrów, 318
  - zmiennych segmentów, 296
- wartość
  - domyślna dla kontrolera i akcji, 276
  - domyślna w ścieżce, 276
  - returnUrl, 571
- warunkowe dołączanie w Ninject, 126
- WAT, Web Site Administration Tool, 592
- WatiN, 87
- Web Developer Toolbar, 570
- WebPI, Web Platform Installer, 34
- wersje metody ValidationSummary, 485
- weryfikowanie przy użyciu Moq, 139
- wiązanie witryn
  - adres IP, 622
  - nazwa hosta, 622
  - numer portu, 622
- widoczność elementu, 557
- widok
  - AppointmentData.cshtml, 517
  - Checkout.cshtml, 202
  - ChildAction.cshtml, 368
  - Completed.cshtml, 211
- widok
  - Edit, 223
  - GetLegacyURL, 301

## widok

- Index, 191
- Index.cshtml, 220, 516, 537
- List.cshtml, 149, 163
- LogOn, 240, 241
- MakeBooking.cshtml, 480
- Menu, 178
- ProductSummary.cshtml, 190
- Razor, 108, 109, 401
- RsvpForm, 57
- Summary, 199
- Thanks, 60

## widoki

- beztypowe, 327
- częściowe, 169, 428
- edytora, 480
- silnie typowane, 54, 327
- tylko do odczytu, 441
- typowane dynamicznie, 410

## Windows Server 2008 R2, 37

## witryna

- ASP.NET, 41
- IIS, 41
- jQuery, 41
- stackoverflow.com, 41

## witryny Web, 621

## właściwości

- automatyczne, 91
- klasy
  - ActionExecutingContext, 358
  - ActionExecutingContext, 356
  - AuthorizationContext, 350
  - AuthorizeAttribute, 351
  - ControllerContext, 350
  - ExceptionContext, 353
  - HandleErrorAttribute, 355
  - ModelMetadata, 496
- obiektu cookie uwierzytelniania, 585, 587
- profilu, 601
- w C#, 89
- wyszukiwania, 404
- wywołań zwrotnych, 528

## właściwość

- AjaxOptions, 519, 520, 527
- AjaxOptions.Confirm, 523
- AjaxOptions.LoadingElementId, 523
- Category, 96
- Controller.Response, 319
- CurrentCategory, 172
- DayOfWeak, 327
- Exception, 353
- FormattedModelValue, 455

## Greeting, 51

- HttpContext, 194
- IsApproved, 439
- Mock.Object, 137
- ModelState, 207
- ModelState.IsValid, 207, 482
- Name, 90
- Person.Role, 448
- ProductId, 224
- Products, 163
- Response, 320
- RouteExistingFiles, 291, 292
- UpdateTargetId na tabledata, 519
- ViewBag.Message, 260
- ViewBag.Title, 114
- ViewData.TemplateInfo, 454
- WillAttend, 53
- WillAttend, 61

## włączanie

- profilu anonimowych, 601
- roli serwera WWW, 37
- usługi zarządzania, 41

## wnioskowanie typów, 98, 99

## wsparcie dla testów automatycznych, 30

## wstrzykiwanie

- HTML, 570
- SQL, 567, 580
- wartości do konstruktora, 123
- zależności, 77, 266, 402, 470

## wybieranie bazy danych, 591

## wybór

- konfiguracji, 253
- typu instalacji, 620
- typu projektu, 44

## wygenerowana klasa C#, 402

## wyjątek ArgumentNullException, 318

## wyjątki, 262

## wykrywanie

- błędów, 607
- żądań Ajax, 532

## wyłączanie

- kontroli żądań, 573
- właściwości, 442

## wymiana szablonu wbudowanego, 453

## wynik zapytania, 105

## wyniki akcji, 50, 320

## wyrażenia lambda, 29, 56, 97

## wyróżnianie bieżącej kategorii, 182

## wyróżnienie elementów, 483

## wysyłanie

- kodów HTTP, 339
- kodu 401, 340



- kodu 404, 340
  - obiektu JSON, 533
  - pliku, 337
  - tablicy bajtów, 339
  - zawartości strumienia, 339
- wyszukiwanie
  - szablonów, 451
  - widoków, 324
- wyświetlanie
  - informacji, 52, 522
  - informacji o błędach, 486
  - komunikatów, 229, 486, 487
  - listy produktów, 148
  - łączy stron, 158, 163
  - zawartości koszyka, 192
  - zdjęć produktów, 247, 249
- wywołania Ajax, 521
- wywołanie @RenderSection, 426
- wywołanie zwrotne, 527
- wywoływanie
  - akcji podrzędnych, 432
  - metod akcji, 371
- wzorce projektowe, 69
- wzorzec
  - A/A/A, arrange/act/assert, 81, 131
  - architektury trójwarstwowej, 71
  - model-widok, 70
  - MVC, 26, 67
  - POST-Redirect-GET, 331
  - programowania asynchronicznego, 393
  - Smart UI, 69, 70
  - TDD, 132
  - URL, 271
  - URL z segmentem mieszanym, 278
  - URL z segmentem statycznym, 277

## Z

- zachowywanie danych, 334
- zakodowany HTML, 408
- zalety ASP.NET MVC, 26–29
- zaprzyżniona klasa metadanych, 446
- zapytania opóźnione, 102
- zapytanie parametryczne, 581
- zarządzanie
  - czasem nieaktywności, 392
  - członkostwem, 592, 593
  - rolami, 598
  - stanem sesji, 385
- zasada czerwone-zielone-refaktoryzacja, 82
- zasady biznesowe, 70
- zasady kontroli poprawności, 491, 507

- zawartość koszyka, 190
- zdalna kontrola poprawności, 507, 511
- zdarzenia
  - JavaScript, 556
  - jQuery, 555
- zgodność pomiędzy przeglądarkami, 30
- zintegrowane środowisko programistyczne, IDE, 117
- zmiana
  - ciągu połączenia, 611
  - elementów konfiguracji, 616
  - lokalizacji wyszukiwania, 405
  - wrażliwych właściwości, 582
  - żądania, 373
- zmiennie
  - przechwytyjące, 284
  - segmentów, 297
  - w debugerze, 262
  - we wzorcu URL, 280
  - własne, 280
- znacznik
  - @:, 110, 571
  - @section, 425
  - @using, 407
  - div, 525
  - inherits, 403
  - łącza z atrybutami, 297
  - script, 518
- znaczniki deklaratywne, 406
- znak @, 31, 109, 160
- znaki =>, 98
- znaki cudzysłowu, 160
- zwracanie
  - błędów, 339
  - danych, 333
  - danych binarnych, 337
  - danych JSON, 336
  - danych XML, 336
  - plików, 337

## Ż

- żądania
  - Ajax, 522
  - asynchroniczne, 523
  - przesłania pliku statycznego, 290
- żądanie
  - GET, 58, 311, 568
  - HTTP, 570
  - POST, 58, 227, 311, 568
  - REST, 383



# 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 3 to kolejny krok w ewolucji platformy ASP.NET firmy Microsoft. W tej bibliotece programowania witryn WWW połączono efektywność i schludność architektury model-widok-kontroler (MVC), najnowsze pomysły i techniki programowania zwinnego oraz najlepsze części istniejącej platformy ASP.NET. Jest ona więc alternatywą dla tradycyjnych projektów ASP.NET Web Forms i ma nad tą platformą znaczną przewagę. Szkielet ASP.NET MVC 3 pozwoli Ci błyskawicznie stworzyć skalowalne, łatwe w utrzymaniu i rozwijaniu aplikacje internetowe, a jego wykorzystanie da Ci niepowtarzalną szansę dziecinnie łatwego stosowania testów jednostkowych i podejścia TDD (ang. Test Driven Development). Co jeszcze możesz zyskać dzięki ASP.NET MVC 3?

O tym przekonasz się w trakcie lektury tej wyjątkowej książki. Jej autorzy przyjęli słuszną zasadę, że dobry przykład mówi więcej niż kilka stron suchej teorii. Zobaczysz więc, jak wzorzec MVC sprawdza się w praktyce, jakie są jego zalety i wady oraz jak unikać typowych pułapek. Nauczysz się korzystać z filtrów, konfigurować kontrolery oraz projektować wydajny model. W dzisiejszych czasach bezpieczeństwo aplikacji jest stawiane na pierwszym miejscu, dlatego szczególną uwagę powinieneś zwrócić na rozdział poświęcony zabezpieczeniu stworzonego przez Ciebie rozwiązania. Ta książka to kompletna pozycja dla każdego programisty ASP.NET. Sięgnij po nią i przekonaj się, że:

- wzorzec MVC ułatwia pracę
- jego zastosowanie korzystnie wpływa na jakość kodu
- bezpieczeństwo aplikacji jest najważniejsze
- implementacja wzorca MVC nie musi być trudna!

# Apress®

Nr katalogowy: 11679

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 najchętniej czytane:  
• <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
• <http://helion.pl/nowosci>

**Helion**

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

Cena 99,00 zł

ISBN 978-83-246-4822-1



9 788324 648221