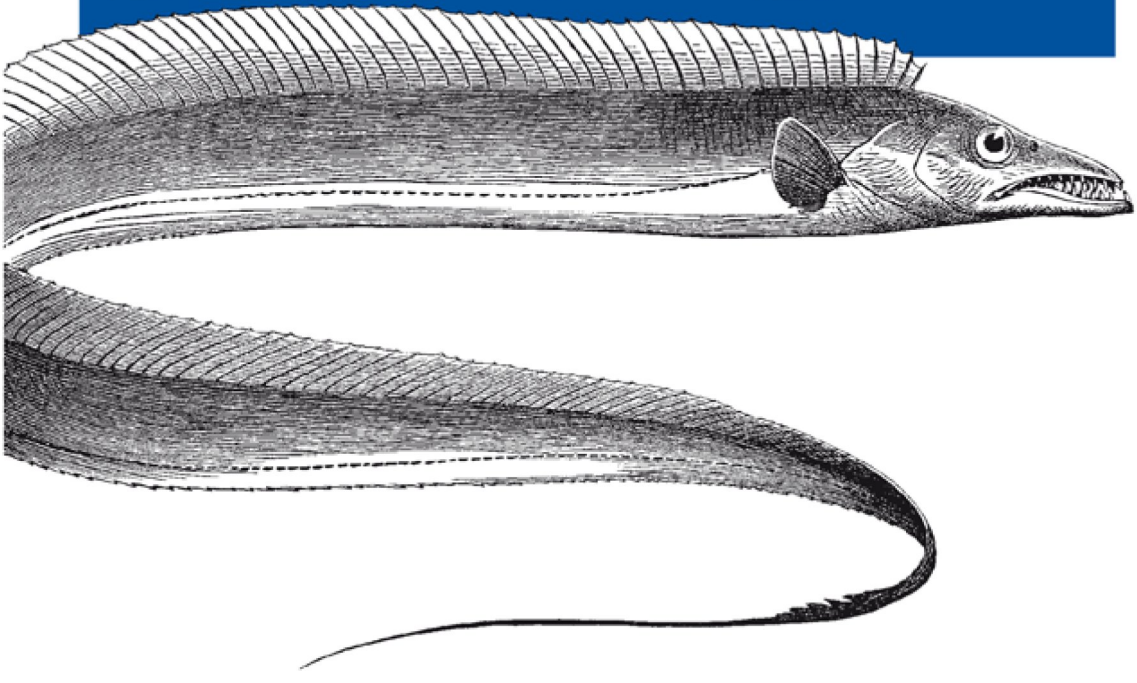
Wykorzystaj potencjał ASP.NET!

Programowanie

ASP.NET MVC 4



HELION

O'REILLY®

Jess Chadwick, Todd Snyder, Hrusikesh Panda

Tytuł oryginału: Programming ASP.NET MVC 4: Developing Real-World WebApplications with ASP.NET MVC

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-6647-8

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/aspm4p_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Wprowadzenie **11**

Część I. Rozkręcamy się **15**

Rozdział 1. Podstawy ASP.NET MVC **17**

Opracowane przez Microsoft platformy tworzenia aplikacji sieciowych 17
Active Server Pages (ASP) 18
ASP.NET Web Forms 18
ASP.NET MVC 18
Architektura MVC 19
Model 20
Widok 20
Kontroler 20
Co nowego w ASP.NET MVC 4? 20
Wprowadzenie do aplikacji EBuy 22
Instalacja ASP.NET MVC 23
Tworzenie aplikacji ASP.NET MVC 23
Szablony projektów 23
Konwencja przed konfiguracją 27
Uruchamianie aplikacji 28
Routing 28
Konfiguracja tras 29
Kontrolery 31
Akcje kontrolera 32
Obiekt ActionResult 32
Parametry akcji 33
Filtry akcji 36
Widoki 36
Wyszukiwanie widoków 37
Poznaj Razor 38
Odróżnianie kodu od znaczników 39
Układy graficzne 40
Widoki częściowe 41
Wyświetlanie danych 43
Metody pomocnicze HTML i URL 45
Modele 46

Zebranie wszystkich komponentów w całość 47
Trasa 47
Kontroler 47
Widok 49
Uwierzytelnianie 52
AccountController 54
Podsumowanie 55

Rozdział 2. ASP.NET MVC dla programistów formularzy sieciowych 57
Wszystko kręci się wokół ASP.NET 57
Narzędzia, języki i API 58
Moduły i procedury obsługi HTTP 58
Zarządzanie stanem 58
Wdrażanie i środowisko uruchomieniowe 59
Więcej różnic niż podobieństw 59
Oddzielanie logiki aplikacji od logiki widoku 60
Adresy URL i routing 60
Zarządzanie stanem 61
Generowanie kodu HTML 62
Tworzenie widoku ASP.NET MVC za pomocą składni Web Forms 66
Słowo ostrzeżenia 66
Podsumowanie 67

Rozdział 3. Praca z danymi 69
Tworzenie formularza 69
Obsługa akcji POST formularza 71
Zapis danych w bazie danych 71
Technika Code First — zasada „konwencja przed konfiguracją” 72
Tworzenie warstwy dostępu do danych z użyciem techniki Code First w Entity Framework 72
Weryfikacja danych 73
Określanie reguł biznesowych za pomocą adnotacji danych 74
Wyświetlanie komunikatów o błędach z procesu weryfikacji danych 77
Podsumowanie 80

Rozdział 4. Programowanie po stronie klienta 81
Praca z językiem JavaScript 81
Selektory 83
Udzielanie odpowiedzi na zdarzenia 86
Manipulacje modelem DOM 88
AJAX 89
Weryfikacja danych po stronie klienta 91
Podsumowanie 95

Część II. Kolejny poziom 97

Rozdział 5. Architektura aplikacji sieciowej 99
Wzorzec MVC 99
Zasada separacji zadań 99
MVC i platformy sieciowe 100
Architektura aplikacji sieciowej 102
Architektura logiczna 102
Architektura logiczna aplikacji sieciowej ASP.NET MVC 102
Najlepsze praktyki w zakresie architektury logicznej 104
Architektura fizyczna 105
Przestrzeń nazw projektu i nazwy podzespołów 105
Opcje wdrożenia 106
Najlepsze praktyki w zakresie architektury fizycznej 107
Reguły dotyczące architektury aplikacji 108
SOLID 109
Odwracanie sterowania 114
Nie powtarzaj się 121
Podsumowanie 122

Rozdział 6. Usprawnianie witryny poprzez użycie technologii AJAX 123
Częściowe generowanie strony 123
Generowanie widoków częściowych 124
Wygenerowanie kodu JavaScript 129
Wygenerowanie danych JSON 129
Żądanie danych JSON 131
Szablony po stronie klienta 131
Ponowne używanie tej samej logiki zarówno w żądaniach AJAX, jak i pozostałych 134
Udzielanie odpowiedzi na żądania AJAX 135
Udzielanie odpowiedzi na żądania JSON 136
Zastosowanie tej samej logiki w wielu akcjach kontrolera 137
Wysyłanie danych do serwera 138
Przekazywanie skomplikowanych obiektów JSON 140
Wybór łącznika modelu 141
Efektywne wysyłanie i odbieranie danych JSON 143
Wykonywanie żądań AJAX między domenami 144
JSONP 144
Włączanie Cross-Origin Resource Sharing 147
Podsumowanie 148

Rozdział 7. Platforma Web API ASP.NET 149
Tworzenie usługi danych 149
Rejestracja tras Web API 151
Wykorzystanie techniki „konwencja przed konfiguracją” 151
Nadpisanie konwencji 152
Użycie API 153

Stronicowanie i pobieranie danych 155
Obsługa wyjątków 156
Media 158
Podsumowanie 161

Rozdział 8. Zaawansowane dane ... 163
Wzorce dostępu do danych 163
Klasy POCO 163
Używanie wzorca repozytorium 164
Mapowanie obiektowo-relacyjne 166
Ogólny opis Entity Framework 168
Wybór podejścia w zakresie dostępu do danych 169
Współbieżność w bazie danych 169
Tworzenie warstwy dostępu do danych 171
Podejście Entity Framework Code First 171
Model domeny biznesowej aplikacji EBuy 173
Praca z kontekstem danych 176
Sortowanie, filtrowanie i stronicowanie danych 178
Podsumowanie 183

Rozdział 9. Zapewnianie bezpieczeństwa ... 185
Tworzenie bezpiecznej aplikacji sieciowej 185
Obrona 185
Nigdy nie ufaj danym wejściowym 186
Wymuszanie stosowania reguły najmniejszych uprawnień 186
Przyjmuj założenie, że zewnętrzne systemy są niebezpieczne 186
Ogranicz możliwości ataku 186
Wyłącz niepotrzebne funkcje 187
Zabezpieczanie aplikacji 187
Zabezpieczanie aplikacji intranetowej 188
Uwierzytelnianie formularzy 193
Ochrona przed atakami 200
SQL Injection 201
Cross-Site Scripting 206
Cross-Site Request Forgery 207
Podsumowanie 209

Rozdział 10. Programowanie na platformy mobilne ... 211
Funkcje mobilne platformy ASP.NET MVC 4 211
Większa przyjazność aplikacji mobilnej 213
Tworzenie widoku mobilnego dla aukcji 214
Rozpoczęcie pracy z jQuery Mobile 215
Usprawnianie widoku za pomocą jQuery Mobile 218
Unikanie widoków biurkowych w witrynie mobilnej 223
Usprawnianie wersji mobilnej witryny 224

Technika Adaptive Rendering 225
Znacznik viewport 225
Wykrywanie funkcji mobilnych 226
Zapytania mediów CSS 228
Widoki dla konkretnych przeglądarek internetowych 229
Tworzenie nowej aplikacji mobilnej zupełnie od początku 231
Platforma jQuery Mobile 232
Szablon aplikacji mobilnej w ASP.NET MVC 4 232
Używanie szablonu aplikacji mobilnej w ASP.NET MVC 4 233
Podsumowanie 236

Część III. Zagadnienia zaawansowane 237

Rozdział 11. Operacje na danych przeprowadzane równolegle, asynchronicznie i w czasie rzeczywistym 239
Kontroler asynchroniczny 239
Tworzenie kontrolera asynchronicznego 240
Kiedy używać kontrolera asynchronicznego? 242
Asynchroniczna komunikacja w czasie rzeczywistym 242
Porównanie modeli aplikacji 242
Model HTTP polling 243
Model HTTP long polling 244
Zdarzenia wysyłane przez serwer 245
WebSocket 246
Usprawnianie komunikacji w czasie rzeczywistym 247
Konfiguracja i dostrajanie 250
Podsumowanie 252

Rozdział 12. Buforowanie 253
Rodzaje buforowania 253
Buforowanie po stronie serwera 253
Buforowanie po stronie klienta 254
Techniki buforowania po stronie serwera 254
Buforowanie o zasięgu żądania 254
Buforowanie o zasięgu użytkownika 255
Buforowanie o zasięgu aplikacji 256
Bufor ASP.NET 256
Bufor danych wyjściowych 258
Buforowanie donut caching 261
Buforowanie donut hole caching 263
Buforowanie rozproszone 264
Techniki buforowania po stronie klienta 269
Działanie bufora przeglądarki internetowej 269
AppCache 271
Local Storage 273
Podsumowanie 274

Rozdział 13. Techniki optymalizacji po stronie klienta 275
Anatomia strony 275
Anatomia HttpRequest 276
Najlepsze praktyki 277
Wykonuj mniejszą liczbę żądań HTTP 278
Używaj CDN 278
Dodawaj nagłówek Expires lub Cache-Control 280
Komponenty skompresowane w formacie GZip 282
Umieszczaj arkusze stylów na początku pliku 283
Umieszczaj skrypty na końcu dokumentu 283
Korzystaj z zewnętrznych skryptów i arkuszy stylów 285
Zmniejszanie liczby zapytań DNS 286
Minimalizacja plików JavaScript i CSS 286
Unikaj przekierowań 287
Usunięcie powielających się skryptów 289
Konfiguracja nagłówka ETag 289
Pomiar wydajności po stronie klienta 290
Wykorzystanie platformy ASP.NET MVC do pracy 293
Tworzenie paczek i minimalizacja 294
Podsumowanie 297

Rozdział 14. Zaawansowany routing 299
Wayfinding 299
Adresy URL i techniki SEO 301
Tworzenie tras 302
Domyślne parametry i opcjonalne trasy 303
Priorytet i kolejność tras 305
Routing do istniejących plików 305
Ignorowanie tras 305
Trasy typu Catch-All 306
Ograniczenia trasy 307
Narzędzie Glimpse i trasy 309
Routing oparty na atrybutach 310
Rozszerzanie routingu 313
Mechanizm routingu 314
Podsumowanie 318

Rozdział 15. Ponownie używane komponenty interfejsu użytkownika 319
Co platforma ASP.NET MVC oferuje standardowo? 319
Widoki częściowe 319
Metody rozszerzające HtmlHelper czy własne metody? 319
Szablony Display i Editor 320
Html.RenderAction() 320
Przejście o krok dalej 321
Razor Single File Generator 321
Tworzenie wielokrotnie wykorzystywanych widoków ASP.NET MVC 323
Tworzenie wielokrotnie używanych metod pomocniczych ASP.NET MVC 327

Testy jednostkowe dla widoków Razor 328
Podsumowanie 330

Część IV. Kontrola jakości ... 331

Rozdział 16. Rejestrowanie informacji ... 333
Obsługa błędów na platformie ASP.NET MVC 333
Włączanie własnych błędów 334
Obsługa błędów w akcjach kontrolerów 335
Definiowanie globalnych procedur obsługi błędów 335
Rejestrowanie informacji i śledzenie 337
Rejestrowanie informacji o błędach 337
Monitorowanie stanu ASP.NET 340
Podsumowanie 342

Rozdział 17. Zautomatyzowane testowanie ... 343
Semantyka testowania 343
Ręczne testowanie 344
Zautomatyzowane testowanie 345
Poziomy zautomatyzowanego testowania 345
Testy jednostkowe 345
Szybkość (ang. fast) 347
Testy integracyjne 348
Testy akceptacyjne 349
Co to jest projekt zautomatyzowanych testów? 350
Tworzenie projektu testowego w Visual Studio 350
Tworzenie i przeprowadzanie testu jednostkowego 351
Testowanie aplikacji ASP.NET MVC 354
Testowanie modelu 354
Test-Driven Development 357
Tworzenie przejrzystych, zautomatyzowanych testów 358
Testowanie kontrolerów 360
Refaktoring testów jednostkowych 363
Symulacja spełnienia zależności 364
Testowanie widoków 368
Test pokrycia 370
Mit 100% wyniku testu pokrycia 372
Tworzenie kodu łatwego do testowania 372
Podsumowanie 374

Rozdział 18. Automatyzacja kompilacji ... 375
Tworzenie skryptów kompilacji 376
Projekty Visual Studio są skryptami kompilacji! 376
Dodanie prostego zadania kompilacji 376
Przeprowadzanie kompilacji 377
Możliwości są nieograniczone! 378

Automatyzacja kompilacji 378
Rodzaje zautomatyzowanej kompilacji 379
Definiowanie zautomatyzowanej kompilacji 380
Ciągła integracja 383
Wykrywanie problemów 383
Reguły ciągłej integracji 384
Podsumowanie 388

Część V. Umieszczanie aplikacji sieciowej w internecie 389

Rozdział 19. Wdrażanie 391
Co trzeba wdrożyć? 391
Podstawowe pliki witryny internetowej 391
Treść statyczna 394
Czego nie trzeba wdrażać? 394
Bazy danych oraz inne zewnętrzne zależności 395
Jakie są wymagania aplikacji EBuy? 396
Wdrażanie na serwerze Internet Information Server 396
Przygotowania 397
Tworzenie i konfiguracja witryny internetowej na serwerze IIS 397
Publikowanie witryny z poziomu Visual Studio 399
Wdrażanie za pośrednictwem Windows Azure 403
Tworzenie konta Windows Azure 403
Tworzenie nowej witryny internetowej Windows Azure 404
Publikacja witryny internetowej Windows Azure poprzez system kontroli wersji 404
Podsumowanie 406

Dodatki 407

Dodatek A. Integracja platform ASP.NET MVC i Web Forms 409

Dodatek B. Wykorzystanie NuGet jako platformy 417

Dodatek C. Najlepsze praktyki 435

Dodatek D. Odniesienia — tematy, funkcje i scenariusze 449

Skorowidz 453

Wprowadzenie

Dziedzina aplikacji sieciowych jest obszerna i zróżnicowana. Opracowana przez Microsoft platforma ASP.NET — zbudowana na bazie dojrzałej i solidnej platformy .NET — jest szczególnie godna zaufania. Z kolei ASP.NET MVC to najnowszy produkt Microsoftu z rodziny ASP.NET, zapewniający programistom sieciowym alternatywne podejście programistyczne i ułatwiający tworzenie aplikacji sieciowych.

Główny cel niniejszej książki jest prosty — chcemy pomóc Ci w poznaniu platformy ASP.NET MVC 4 zupełnie od początku. Jednak nie poprzestajemy na tym — w książce połączono prezentację podstawowych koncepcji ASP.NET MVC z rzeczowym spojrzeniem na nie, z nowoczesnymi technologiami sieciowymi (takimi jak HTML5 i platforma JavaScript jQuery), a także z oferującymi duże możliwości wzorcami architekturalnymi. Dzięki temu będziesz przygotowany nie tylko do utworzenia witryny internetowej opartej na platformie ASP.NET, ale także do utworzenia stabilnej i skalowalnej aplikacji sieciowej, która będzie łatwa do rozbudowy i obsługi, gdy wystąpi taka konieczność.

Czytelnicy

Niniejsza książka jest skierowana do Czytelników, którzy chcą się dowiedzieć, jak wykorzystać opracowaną przez Microsoft platformę ASP.NET MVC do tworzenia solidnych i łatwych w obsłudze witryn internetowych. W książce przedstawiono wiele przykładowych fragmentów kodu pokazujących szczegóły procesu tworzenia witryn internetowych. Jest ona skierowana nie tylko do programistów aplikacji sieciowych. Zaprezentowano w niej koncepcje i techniki przydatne zarówno dla programistów tworzących kod aplikacji, jak i dla liderów kierujących pracami nad projektami.

Założenia

Wprawdzie książka ma Ci dostarczyć wszystkie informacje wymagane do tworzenia solidnych i łatwych w obsłudze aplikacji sieciowych opartych na technologii ASP.NET MVC, ale przyjęto założenie, że masz już pewną podstawową wiedzę z zakresu tworzenia aplikacji na platformie .NET. Innymi słowy, powinieneś umieć używać technologii HTML, CSS i JavaScript do przygotowania prostych witryn internetowych i posiadać podstawową wiedzę z zakresu platformy .NET i języka C#, która pozwala Ci na utworzenie aplikacji typu „Witaj, świecie”.

Przykładowe fragmenty kodu używane w aplikacji zostały umieszczone w serwisie GitHub pod adresem *https://github.com/ProgrammingAspNetMvcBook/CodeExamples*.

Konwencje zastosowane w książce

W książce tej zastosowano następujące konwencje typograficzne:

Kursywa
: Wskazuje na nowe pojęcia, adresy URL i e-mail, bazy danych, tabele, nazwy plików, rozszerzenia plików itd.

`Czcionka o stałej szerokości`
: Użyta w przykładowych fragmentach kodu, a także w samym tekście, aby odwołać się do pewnych elementów programu, takich jak nazwy zmiennych lub funkcji, typów danych, zmiennych środowiskowych, poleceń i słów kluczowych.

`Pogrubiona czcionka o stałej szerokości`
: Użyta w celu zwrócenia uwagi na dany fragment kodu, wyeksponowania poleceń bądź innego tekstu, który powinien być wprowadzony przez Czytelnika.

`Pochylona czcionka o stałej szerokości`
: Wskazuje tekst, który powinien być zastąpiony wartościami podanymi przez użytkownika bądź wynikającymi z kontekstu.

Taka ikona oznacza wskazówkę, sugestię lub ogólną uwagę.

Taka ikona oznacza ostrzeżenie.

Użycie przykładowych kodów

Książka ta ma na celu pomóc Ci w pracy. Ogólnie rzecz biorąc, można wykorzystywać przykłady z tej książki w swoich programach i dokumentacji. Nie trzeba kontaktować się z nami w celu uzyskania zezwolenia, dopóki nie powiela się znaczących ilości kodu. Na przykład pisanie programu, w którym znajdzie się kilka fragmentów kodu z tej książki, nie wymaga zezwolenia, jednak sprzedawanie lub rozpowszechnianie płyty CD-ROM zawierającej przykłady z książki wydawnictwa O'Reilly wymaga zezwolenia. Odpowiedź na pytanie przez cytowanie tej książki lub przykładowego kodu nie wymaga zezwolenia, ale już włączenie znaczących ilości przykładowych kodów z tej książki do dokumentacji produktu Czytelnika wymaga zezwolenia.

Jesteśmy wdzięczni za przypisy, ale nie wymagamy ich. Przypis zwykle zawiera tytuł, autora, wydawcę i ISBN. Na przykład: J. Chadwick, T. Snyder, H. Panda, *ASP.NET MVC 4. Programowanie*, ISBN 978-83-246-6644-7, Helion, Gliwice 2013.

Jeżeli jednak masz jakiekolwiek wątpliwości dotyczące użycia przykładowych kodów, po prostu skontaktuj się z nami, korzystając z adresu *permissions@oreilly.com*.

CZĘŚĆ I

Rozkręcamy się

ROZDZIAŁ 1.

Podstawy ASP.NET MVC

Microsoft ASP.NET MVC to platforma służąca do tworzenia aplikacji sieciowych opartych na popularnej i dopracowanej platformie .NET. ASP.NET MVC w dużej mierze wykorzystuje wzorce projektowe i praktyki, które kładą nacisk na luźną architekturę aplikacji oraz tworzenie łatwego w obsłudze kodu.

W tym rozdziale zapoznasz się z podstawami platformy ASP.NET MVC, od jej rodowodu i koncepcji architektonicznych, na bazie których została utworzona, aż po sposoby wykorzystania oprogramowania Microsoft Visual Studio 2012 do tworzenia w pełni funkcjonalnych aplikacji sieciowych ASP.NET MVC. Następnie zagłębisz się w projekt aplikacji sieciowej ASP.NET MVC, aby poznać możliwości oferowane przez platformę — między innymi funkcjonujące strony internetowe oraz wbudowane formularze uwierzytelniania pozwalające użytkownikom na rejestrowanie się i logowanie do witryny internetowej.

Gdy ukończysz lekturę tego rozdziału, będziesz nie tylko dysponował działającą aplikacją sieciową w technologii ASP.NET MVC, ale również będziesz miał wiedzę z zakresu podstaw ASP.NET MVC, pozwalającą na natychmiastowe rozpoczęcie tworzenie własnych aplikacji. Pozostała część książki po prostu opiera się na przedstawionych w tym rozdziale podstawach i pokazuje, jak wykorzystać możliwości platformy ASP.NET MVC w niemal każdej aplikacji sieciowej.

Opracowane przez Microsoft platformy tworzenia aplikacji sieciowych

Zrozumienie przeszłości może w ogromnym stopniu pomóc w afirmacji teraźniejszości. Dlatego też przed przejściem do przedstawienia platformy ASP.NET MVC i sposobu jej działania poświęcimy chwilę jej genezie.

Dawno temu Microsoft dostrzegł potrzebę opracowania platformy opartej na Windows, przeznaczonej do tworzenia aplikacji sieciowych, i rozpoczął intensywne prace nad jej tworzeniem. W przeciągu ostatnich dwóch dekad Microsoft dostarczył społeczności programistów kilka platform przeznaczonych do tworzenia aplikacji sieciowych.

Active Server Pages (ASP)

Pierwszą platformą wydaną przez Microsoft był ASP, czyli język skryptowy, w którym kod oraz znaczniki zostały umieszczone w jednym pliku odpowiadającym stronie witryny internetowej. Podejście stosowane przez ASP (język skryptowy działający po stronie serwera) stało się niezmiernie popularne i na jego podstawie powstało wiele witryn internetowych. Niektóre z nich działają do dnia dzisiejszego. Jednak po początkowym zachwycie apetyt programistów wzrósł. Najczęściej pojawiające się propozycje dotyczyły większych możliwości w zakresie ponownego wykorzystywania kodu, lepszego oddzielenia treści strony od tworzącego ją kodu oraz łatwiejszego tworzenia kodu w sposób zorientowany obiektowo. Odpowiedzią firmy Microsoft na wymienione propozycje było wydanie w 2002 roku platformy ASP.NET.

ASP.NET Web Forms

Podobnie jak w przypadku ASP, witryna w technologii ASP.NET stosuje podejście oparte na stronach, w którym poszczególne strony witryny są przedstawiane w postaci fizycznych plików (nazywanych Web Forms), dostępnych poprzez swoje nazwy. W przeciwieństwie do stron opartych na ASP strony Web Forms pozwalają na pewne oddzielenie kodu od znaczników poprzez stosowanie dwóch oddzielnych plików — po jednym dla znaczników (treści strony) i kodu. Podejście ASP.NET Web Forms wystarczyło programistom na wiele lat i nadal przez licznych programistów .NET jest wybierane jako rozwiązanie do tworzenia aplikacji sieciowych. Jednak pewna część programistów .NET uznała podejście ASP.NET Web Forms za zbyt dużą abstrakcję w stosunku do technologii HTML, JavaScript i CSS. Niektórzy programiści mogą być niezadowoleni, prawda?

ASP.NET MVC

Microsoft szybko dostrzegł nowe potrzeby wysuwane przez społeczność programistów ASP.NET domagającą się innego podejścia niż oparte na stronach Web Forms. W roku 2008 została więc wydana pierwsza wersja ASP.NET MVC. Ta platforma jest zupełnie inna niż podejście stosowane w Web Forms. Na platformie ASP.NET MVC całkowicie zrezygnowano z architektury opartej na stronach i zamiast niej zastosowano architekturę MVC (ang. *Model-View-Controller*, model-widok-kontroler).

W przeciwieństwie do platformy ASP.NET Web Forms, która zastąpiła swoją poprzedniczkę, ASP, platforma ASP.NET MVC w żaden sposób *nie zastępuje* istniejącej platformy Web Forms. Zarówno aplikacje ASP.NET MVC, jak i Web Forms są tworzone na podstawie platformy ASP.NET dostarczającej API, które jest intensywnie wykorzystywane przez obie wymienione platformy.

Idea polega na tym, że ASP.NET MVC i Web Forms to po prostu różne sposoby tworzenia witryn internetowych ASP.NET — jest ona powszechnie stosowana w niniejszej książce. W rozdziale 2. i w dodatku A znajdziesz więcej dokładniejszych informacji na temat tej koncepcji.

Architektura MVC

Architektura MVC to wzorzec architektoniczny zachęcający do ścisłego oddzielenia poszczególnych części aplikacji. Izolacja ta jest lepiej znana jako *zasada separacji zadań* (ang. *separation of concerns*) lub *luźne powiązania*. Ogólnie rzecz ujmując, wszystkie aspekty architektury MVC — a tym samym platformy ASP.NET MVC — mają na celu rozdzielenie poszczególnych części aplikacji.

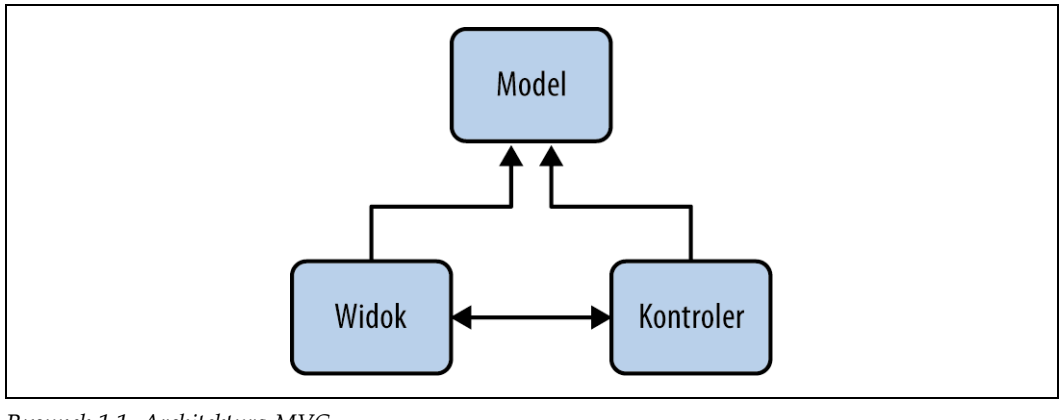
Tworzenie aplikacji w oparciu o architekturę luźnych powiązań charakteryzuje się pewnymi krótko- i długoterminowymi zaletami, które wymieniono poniżej.

Faza tworzenia
: Poszczególne komponenty aplikacji bezpośrednio nie zależą od innych, co pozwala na ich łatwe tworzenie w izolacji. Komponenty mogą być szybko zastąpione lub zamienione, programista unika komplikacji polegających na tym, że jeden komponent wpływa na sposób tworzenia innych komponentów, z którymi współpracuje.

Faza testowania
: Luźne powiązanie komponentów pozwala na testowanie implementacji, aby stały się komponentami produkcyjnymi. W ten sposób można łatwiej uniknąć np. wykonywania wywołań do bazy danych poprzez zastąpienie komponentu wykonującego tego rodzaju wywołania innym, który po prostu zwraca dane statyczne. Możliwość łatwego zastępowania komponentów znacznie ułatwia proces testowania, co z kolei przekłada się na zwiększenie niezawodności całej aplikacji.

Faza obsługi
: Odizolowana logika komponentu oznacza, że zmiany zwykle dotyczą niewielkiej liczby komponentów, najczęściej tylko jednego. Ponieważ ryzyko związane z wprowadzanymi zmianami jest skorelowane z zakresem tych zmian, to zalecane jest przeprowadzanie modyfikacji mniejszej liczby komponentów.

W architekturze MVC aplikacja jest podzielona na trzy warstwy — model, widok i kontroler (rysunek 1.1). Każda z tych warstw ma własne zadanie do wykonania, a co najważniejsze, jego wykonanie nie zależy od sposobu działania pozostałych warstw.



Rysunek 1.1. Architektura MVC

Model

Model przedstawia podstawową logikę biznesową oraz dane. Model hermetyzuje właściwości oraz zachowanie domeny encji i udostępnia właściwości opisujące tę encję. Na przykład klasa `Auction` przedstawia w aplikacji koncepcję aukcji i może udostępniać właściwości, takie jak `Title` i `CurrentBid`, a także zachowanie w postaci metody, takiej jak `Bid()`.

Widok

Widok jest odpowiedzialny za przekształcenie modelu lub modeli na postać wizualną. W aplikacjach sieciowych najczęściej oznacza to wygenerowanie kodu HTML w przeglądarce internetowej użytkownika, choć widok może być wyrażony także w wielu innych postaciach. Ten sam widok można przedstawić jako HTML, PDF, XML i prawdopodobnie nawet jako arkusz kalkulacyjny.

Widok powinien koncentrować się jedynie na *wyświetlaniu* danych i nie powinien zawierać żadnej logiki biznesowej, co wynika z zasady separacji zadań. Logika biznesowa pozostaje w modelu i dostarcza widokowi wszystkich niezbędnych danych.

Kontroler

Kontroler, jak sama nazwa wskazuje, kontroluje logikę aplikacji i działa w charakterze koordynatora pomiędzy widokiem i modelem. Kontroler otrzymuje poprzez widok dane wejściowe od użytkownika, a następnie pracuje z modelem, wykonując określone akcje, których wynik jest z powrotem przekazywany do widoku.

Co nowego w ASP.NET MVC 4?

W niniejszej książce znajdziesz dokładne omówienie platformy ASP.NET MVC i przekonasz się, w jaki sposób wykorzystać maksymalnie oferowane przez nią funkcje. Ponieważ mamy już czwartą wersję platformy, większa część książki przedstawia funkcje istniejące już we wcześniejszych wersjach. Jeżeli znasz wcześniejsze wersje ASP.NET MVC, możesz pominąć znane Ci tematy i od razu przejść do poznawania nowych.

Poniżej przedstawiono krótkie omówienie wszystkich nowych funkcji wprowadzonych w ASP.NET MVC 4 wraz z odniesieniami do fragmentów książki, w których zostały one zaprezentowane.

Kontrolery asynchroniczne
: Serwer IIS (ang. *Internet Information Services*) przetwarza każde otrzymane żądanie w nowym wątku, więc każde nowe żądanie dodaje jeden wątek do nieskończonej liczby wątków dostępnych dla IIS, nawet jeśli ten wątek pozostaje bezczynny (np. oczekując na udzielenie odpowiedzi przez bazę danych lub usługę sieciową). Ostatnio wprowadzone usprawnienia w .NET 4.0 i serwerze IIS 7 znacznie zwiększyły domyślną liczbę wątków dostępnych w puli dla IIS, ale nadal dobrą praktyką jest unikanie blokowania zasobów systemowych dłużej, niż jest to konieczne. W 4. wersji platformy ASP.NET MVC wprowadzono tak zwane

kontrolery asynchroniczne, które lepiej sprawdzają się w asynchronicznej obsłudze długo wykonywanych żądań. Dzięki użyciu kontrolerów asynchronicznych możesz nakazać platformie zwolnienie wątku przetwarzającego żądanie i wykorzystać ten wątek do wykonania innych zadań w trakcie oczekiwania na zakończenie początkowego zadania danego wątku. Po wykonaniu zadań dodatkowych dany wątek powraca do wykonywania zadania początkowego i zwraca taką samą odpowiedź jak w przypadku zwykłego kontrolera synchronicznego. Jedyna różnica polega na jednoczesnym obsłużeniu większej liczby żądań! Jeżeli jesteś zainteresowany poznaniem kontrolerów asynchronicznych, ich dokładne omówienie znajdziesz w rozdziale 11.

Tryby wyświetlania

Nieustannie zwiększa się liczba urządzeń posiadających połączenie z internetem, za których pośrednictwem mogą być przeglądane witryny internetowe. Powinieneś więc być na to przygotowany. Dane wyświetlane przez owe urządzenia są nierzadko takie same jak w urządzeniach biurkowych, z wyjątkiem tego, że elementy graficzne powinny uwzględniać mniejsze ekrany w urządzeniach mobilnych. Dzięki trybom wyświetlania w ASP.NET MVC otrzymujesz łatwe w użyciu, oparte na konwencjach podejście pozwalające na dostosowanie widoku i układu do różnych urządzeń docelowych. W rozdziale 10. omówiono stosowanie trybów wyświetlania jako część całościowego podejścia w zakresie dodania do witryny internetowej obsługi urządzeń mobilnych.

Łączenie i minimalizacja

Wprawdzie może się wydawać, że szybkie połączenie szerokopasmowe to obecnie jedyny sposób uzyskiwania dostępu do internetu, ale nie wolno zupełnie przypadkowo traktować zasobów po stronie klienta, od których zależy poprawne działanie przygotowanej przez Ciebie witryny. Jeśli weźmiesz pod uwagę zwiększenie się całkowitej ilości czasu potrzebnego na pobranie komponentów witryny, zmarnowane ułamki sekund sumują się i zaczynają mieć negatywny wpływ na postrzeganie danej witryny internetowej. Koncepcje takie jak łączenie skryptów i arkuszy stylów oraz minimalizacja nie są niczym nowym, ale wraz z wydaniem .NET Framework 4.5 stały się podstawową funkcją platformy. Co więcej, ASP.NET MVC jeszcze bardziej rozszerza podstawowe funkcje platformy .NET, aby były jeszcze użyteczniejsze w aplikacjach sieciowych utworzonych w technologii ASP.NET MVC. W rozdziale 13. omówiono wszystkie wymienione powyżej koncepcje i pokazano, jak używać nowych funkcji oferowanych przez platformy ASP.NET i ASP.NET MVC.

Web API

Proste usługi danych HTTP coraz częściej stają się podstawowym sposobem dostarczania danych do różnorodnych aplikacji, urządzeń oraz platform. Platforma ASP.NET MVC zawsze zapewniała możliwość zwrotu danych w różnych formatach, między innymi JSON i XML. Jednak ASP.NET Web API przenosi tę interakcję o krok dalej i zapewnia znacznie nowocześniejszy model programowania, który koncentruje się na dostarczeniu w pełni wyposażonych *usług* danych zamiast akcji kontrolera przeznaczonych do zwracania danych. W rozdziale 6. przekonasz się, jak wykorzystać zalety technologii AJAX po stronie klienta — do tego celu użyjesz ASP.NET Web API!

Czy wiedziałeś, że...?

Platforma ASP.NET MVC jest dostępna jako open source! To prawda, cały kod źródłowy platform ASP.NET MVC, Web API i Web Pages jest dostępny do przeglądania i pobierania na witrynie CodePlex (*http://aspnetwebstack.codeplex.com/*). Co więcej, programiści mają możliwość tworzenia własnych gałęzi tych platform, a nawet wysyłania poprawek do kodu źródłowego oryginalnych platform.

Wprowadzenie do aplikacji EBuy

Niniejsza książka ma za zadanie nie tylko przedstawić wady i zalety platformy ASP.NET MVC, ale również sposoby jej wykorzystania w rzeczywistych aplikacjach. Problem z tego rodzaju aplikacjami polega na tym, że znaczenie słowa „rzeczywiste" oznacza pewien poziom skomplikowania i unikalności, który nie może być łatwo przedstawiony na przykładzie jednej aplikacji.

Zamiast próbować przedstawić rozwiązania wszystkich problemów, które możesz napotkać, zdecydowaliśmy się na przygotowanie listy najczęstszych scenariuszy. Owa lista nie zawiera każdej możliwej sytuacji, z którą możesz się zetknąć, ale jesteśmy przekonani, że przedstawia najważniejsze problemy, na jakie natrafiają programiści zajmujący się tworzeniem własnych aplikacji ASP.NET MVC.

Wcale nie żartujemy — wspomniana lista naprawdę została przez nas napisana i znajdziesz ją na końcu książki. W dodatku D zamieściliśmy listę wszystkich funkcji i scenariuszy przedstawionych w książce oraz podaliśmy rozdziały, w których dany scenariusz został omówiony.

Przygotowaliśmy aplikację sieciową, na której przykładzie omawiamy te scenariusze. Jej działanie powinno być zrozumiałe dla każdego — jest to witryna będąca portalem aukcyjnym.

Przedstawiamy więc aplikację EBuy, tj. utworzoną za pomocą ASP.NET MVC witrynę internetową pozwalającą na prowadzenie aukcji w internecie. Cel witryny jest całkiem jasny — umożliwienie użytkownikom przygotowania listy przedmiotów do sprzedaży oraz umożliwienie licytowania tych, które chcą kupić od innych użytkowników. Jeśli spojrzysz uważniej, to dostrzeżesz aplikację znacznie bardziej skomplikowaną, niż może wydawać się na początku. Wymaga ona nie tylko funkcji oferowanych przez ASP.NET MVC, ale również integracji z innymi technologiami.

EBuy to nie tylko kod dostarczany wraz z książką. Każdy rozdział przedstawia omówienie kolejnych funkcji platformy służących do budowy poszczególnych komponentów — od nowego projektu aż do wdrożonej aplikacji — podążaj za nami i jednocześnie twórz kod!

Przyznajemy, że EBuy to także po prostu kod. Gotową aplikację możesz pobrać z witryny *http://www.programmingaspnetmvc.com/*, poświęconej niniejszej książce.

Wystarczy już tego mówienia o nieistniejącej jeszcze aplikacji, najwyższy czas przystąpić do jej tworzenia!

Instalacja ASP.NET MVC

Aby rozpocząć tworzenie aplikacji ASP.NET MVC, musisz pobrać i zainstalować platformę ASP.NET MVC 4. To bardzo łatwe zadanie, które sprowadza się do przejścia na stronę *http://www.asp.net/mvc* i kliknięcia zielonego przycisku *Install*.

Spowoduje to uruchomienie programu instalacyjnego Web Platform Installer, czyli bezpłatnego narzędzia upraszczającego proces instalacji wielu narzędzi sieciowych i aplikacji. Wykonując kroki wyświetlane przez program instalacyjny, pobierzesz i zainstalujesz na komputerze ASP.NET MVC 4 wraz ze wszystkimi wymaganymi komponentami.

Zwróć uwagę, że w celu zainstalowania i używania ASP.NET MVC 4 musisz mieć programy PowerShell 2.0, Visual Studio 2010 Service Pack 1 lub Visual Web Developer Express 2010 Service Pack 1. Jeśli wymienione programy nie są jeszcze zainstalowane w systemie, program instalacyjny Web Platform Installer powinien automatycznie pobrać i zainstalować najnowsze wersje PowerShell i Visual Studio.

Jeżeli używasz poprzedniej wersji ASP.NET MVC i chciałbyś mieć możliwość tworzenia aplikacji ASP.NET MVC 4 i jednocześnie dalszej pracy z ASP.NET MVC 3, to nie musisz się obawiać problemów. Nowa wersja ASP.NET MVC 4 może być zainstalowana i używana wraz z poprzednią ASP.NET MVC 3.

Po zakończeniu instalacji możesz przejść do kolejnego kroku, jakim jest utworzenie pierwszej aplikacji na platformie ASP.NET MVC 4.

Tworzenie aplikacji ASP.NET MVC

Program instalacyjny ASP.NET MVC 4 dodaje do Visual Studio nowy typ projektu o nazwie *Aplikacja sieci Web platformy ASP.NET MVC 4*. Wymieniony typ projektu to Twoje drzwi do świata ASP.NET MVC i jednocześnie punkt początkowy dla tworzonej w książce aplikacji EBuy.

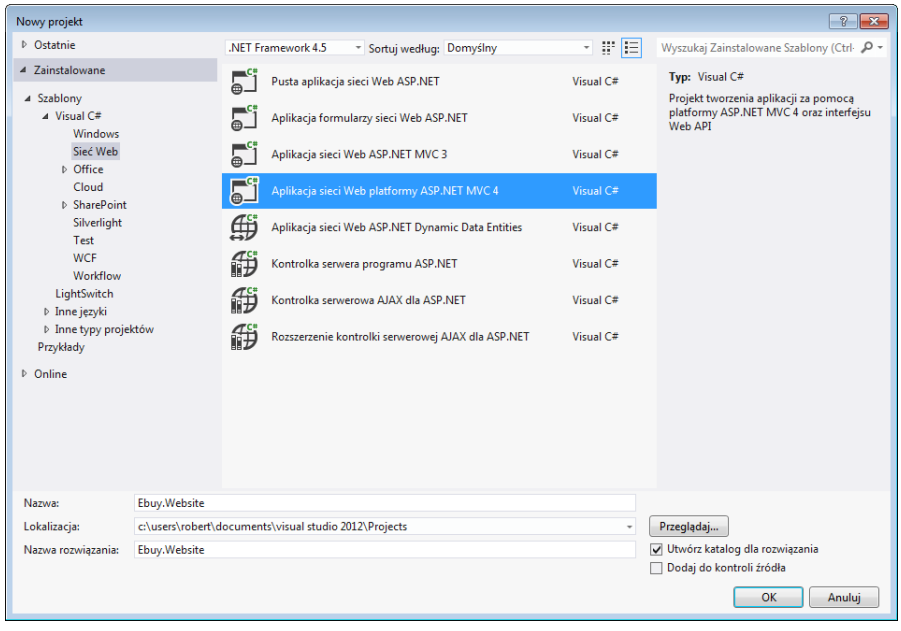
Aby utworzyć nowy projekt, wybierz wersję Visual C# szablonu *Aplikacja sieci Web platformy ASP.NET MVC 4*, a następnie w polu *Nazwa* podaj nazwę *Ebuy.Website* (rysunek 1.2).

Po kliknięciu przycisku *OK* na ekranie zostanie wyświetlone kolejne okno dialogowe, zawierające dalsze opcje (rysunek 1.3).

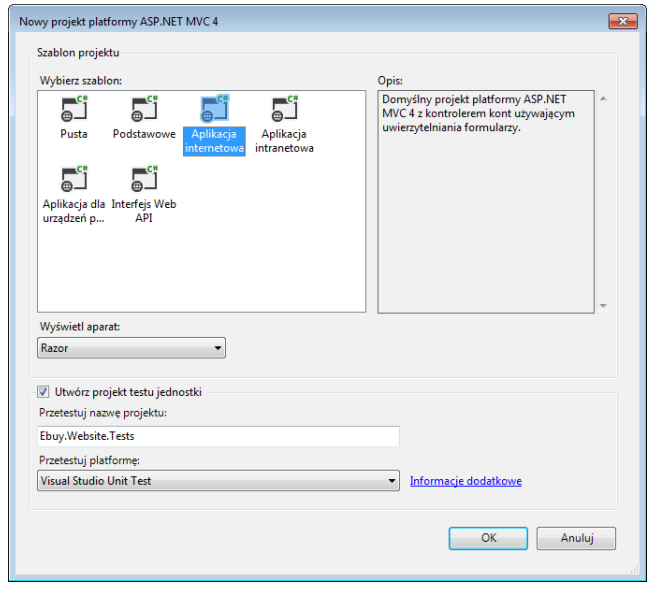
W wyświetlonym oknie dialogowym możesz dostosować generowaną przez Visual Studio aplikację ASP.NET MVC 4 do własnych potrzeb poprzez określenie typu tworzonej witryny internetowej ASP.NET MVC.

Szablony projektów

Platforma ASP.NET MVC 4 oferuje kilka szablonów projektów przeznaczonych do różnych celów.



Rysunek 1.2. Utworzenie projektu EBuy



Rysunek 1.3. Dostosowanie projektu EBuy do własnych potrzeb

Pusta

Szablon *Pusta* powoduje utworzenie podstawowej aplikacji ASP.NET MVC 4 wraz z odpowiednią strukturą katalogów zawierającą odniesienia do podzespołów ASP.NET MVC,

jak również pewne biblioteki JavaScript, z których prawdopodobnie będziesz korzystał. Oprócz tego szablon zawiera domyślny układ widoku oraz wygenerowany plik *Global.asax*, przechowujący kod konfiguracji standardowej wymaganej przez większość aplikacji ASP.NET MVC.

Podstawowe
: Szablon *Podstawowe* tworzy strukturę katalogów zgodną z konwencjami ASP.NET MVC 4 i zawiera odniesienia do podzespołów ASP.NET MVC. Ten szablon to absolutne minimum wymagane do rozpoczęcia tworzenia projektu ASP.NET MVC 4 i nic poza tym — reszta pozostaje w Twoich rękach!

Aplikacja internetowa
: Szablon *Aplikacja internetowa* stanowi rozbudowę wymienionego wcześniej szablonu *Pusta* i zawiera kontroler domyślny (`HomeController`), kontroler `AccountController` wraz z całą logiką wymaganą do rejestrowania i logowania się użytkowników witryny internetowej, a także domyślne widoki dla obu wymienionych kontrolerów.

Aplikacja intranetowa
: Szablon *Aplikacja intranetowa* jest bardzo podobny do szablonu *Aplikacja internetowa* z wyjątkiem tego, że został skonfigurowany do używania uwierzytelniania opartego na Windows, co jest zalecanym rozwiązaniem w aplikacjach intranetowych.

Aplikacja dla urządzeń przenośnych
: Szablon *Aplikacja dla urządzeń przenośnych* to kolejna odmiana szablonu *Aplikacja internetowa*. Jednak ten szablon został zoptymalizowany pod kątem urządzeń mobilnych, zawiera platformę jQuery Mobile i widoki wykorzystujące kod HTML najlepiej dostosowany do współdziałania z jQuery Mobile.

Interfejs Web API
: Szablon *Interfejs Web API* to jeszcze inna odmiana szablonu *Aplikacja internetowa* i zawiera wstępnie skonfigurowany kontroler Web API. Kontroler Web API to nowa, lekka platforma usług sieciowych RESTful HTTP, doskonale integrująca się z ASP.NET MVC. Web API stanowi bardzo dobry wybór, pozwalający na szybkie i łatwe tworzenie usług danych wykorzystywanych przez aplikacje sieciowe oparte na technologii AJAX. Więcej informacji na temat Web API znajdziesz w rozdziale 6.

Okno dialogowe *Nowy projekt platformy ASP.NET MVC 4* pozwala także na wybór tak zwanego *silnika widoku* (ang. *view engine*), czyli składni używanej do tworzenia widoków. W trakcie tworzenia omawianej tutaj aplikacji EBuy będziemy korzystać z nowej składni Razor, więc możesz pozostawić wartość domyślną (*Razor*). W dowolnej chwili możesz zmienić silnik widoku używany przez aplikację. Wspomniana opcja w kreatorze istnieje jedynie po to, aby wskazać *generowany* rodzaj widoku, a nie na stałe zdefiniować w aplikacji silnik widoku.

Wreszcie masz możliwość określenia, czy kreator ma wygenerować projekt tekstów jednostkowych dla tworzonego rozwiązania. Tutaj też warto podkreślić, że nie musisz zbyt przejmować się podjętą teraz decyzją. Podobnie jak w przypadku innych rozwiązań Visual Studio, projekt testów jednostkowych możesz w dowolnej chwili dodać do aplikacji sieciowej ASP.NET MVC.

Kiedy będziesz zadowolony z wybranych opcji, kliknij przycisk *OK* — kreator wygeneruje nowy projekt.

Zarządzanie pakietami NuGet

Jeżeli zwrócisz uwagę na pasek stanu, gdy Visual Studio będzie tworzyć nowy projekt aplikacji sieciowej, to możesz zauważyć komunikaty, np. *Trwa dodawanie AspNetMvc…*. Szablon projektu po prostu odwołuje się do menedżera pakietów NuGet w celu instalacji znajdujących się w aplikacji odniesień do podzespołów i zarządzania nimi. Użycie menedżera pakietów w celu zarządzania zależnościami aplikacji — zwłaszcza jako część fazy tworzenia nowego projektu na podstawie szablonu — daje bardzo duże możliwości i jednocześnie jest jedną z nowości w typach projektów ASP.NET MVC 4.

Wprowadzony jako część programu instalacyjnego ASP.NET MVC 3 menedżer pakietów NuGet oferuje alternatywny sposób zarządzania zależnościami aplikacji. Chociaż tak naprawdę nie jest częścią platformy ASP.NET MVC, to jednak wykonuje w tle sporą ilość pracy, aby utworzenie projektu było możliwe.

W pakiecie NuGet mogą się znajdować różne komponenty — podzespoły, treść, a nawet narzędzia pomagające w pracy nad aplikacją. Kiedy pakiet NuGet jest instalowany, dodaje podzespoły do listy *Odwołania* znajdującej się w projekcie, kopiuje wszelką treść do struktury katalogów aplikacji oraz rejestruje wszystkie narzędzia w bieżącej ścieżce dostępu, aby można było łatwo je uruchamiać z poziomu konsoli menedżera pakietów.

Jednak najważniejszym aspektem pakietów NuGet — a wręcz głównym powodem ich tworzenia — jest *zarządzanie zależnościami*. Aplikacje .NET nie są monolitycznymi, składającymi się z jednego podzespołu programami — w celu wykonania zadania większość podzespołów opiera się na odniesieniach do innych podzespołów. Co więcej, podzespoły są zależne od konkretnej *wersji* (lub przynajmniej wersji minimalnej) innych podzespołów.

Ujmując rzecz najkrócej, pakiet NuGet sprawdza potencjalnie skomplikowane relacje pomiędzy wszystkimi podzespołami wymaganymi przez aplikację. Następnie gwarantuje, że wszystkie potrzebne aplikacji podzespoły są dostępne we właściwych wersjach.

Możliwości pakietów NuGet możesz wykorzystać dzięki menedżerowi pakietów NuGet, do którego dostęp uzyskasz na jeden z dwóch wymienionych poniżej sposobów.

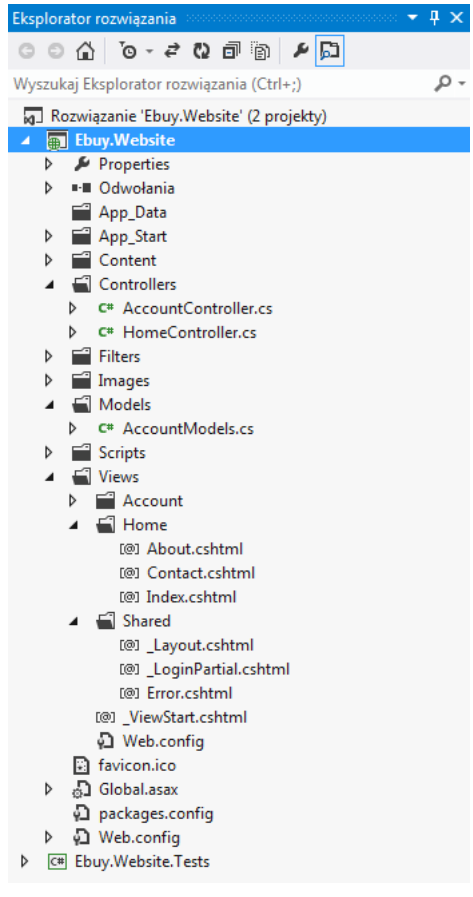
Graficzny interfejs użytkownika
: Menedżer pakietów NuGet ma graficzny interfejs użytkownika (ang. *Graphical User Interface*, GUI), znacznie ułatwiający operacje wyszukiwania, instalacji, uaktualniania i odinstalowywania pakietów w projekcie. Dostęp do interfejsu graficznego menedżera pakietów uzyskasz po kliknięciu prawym przyciskiem myszy projektu witryny w oknie *Eksplorator rozwiązania* i wybraniu opcji *Zarządzaj pakietami NuGet…*.

Konsola
: Konsola menedżera pakietów bibliotek to okno w Visual Studio zawierające zintegrowaną powłokę PowerShell, skonfigurowaną specjalnie w celu uzyskania dostępu do menedżera pakietów bibliotek. Jeżeli po uruchomieniu Visual Studio nie widzisz otwartego okna konsoli menedżera pakietów, możesz je wyświetlić po wybraniu w menu *Narzędzia/Menedżer pakietów bibliotek/Konsola Menedżera pakietów*. Aby zainstalować pakiet z poziomu okna konsoli, po prostu wydaj polecenie `Install-Package Nazwa_Pakietu`. Na przykład w celu zainstalowania pakietu *Entity Framework* wydaj polecenie `Install-Package EntityFramework`. Konsola menedżera pakietów pobierze pakiet *Entity Framework*, a następnie zainstaluje go w projekcie. Po wykonaniu polecenia `Install-Package` podzespoły pakietu *Entity Framework* będą widoczne na liście *Odwołania* projektu.

Konwencja przed konfiguracją

Aby ułatwić tworzenie witryny internetowej i zwiększyć produktywność programistów, gdzie tylko jest to możliwe, platforma ASP.NET MVC opiera się na koncepcji *konwencja przed konfiguracją*. Tak więc zamiast polegać na wyraźnych ustawieniach konfiguracyjnych, na platformie ASP.NET MVC przyjęto założenie, że podczas tworzenia aplikacji programiści będą stosowali określone konwencje.

Pokazana na rysunku 1.4 struktura katalogów projektu to doskonały przykład zastosowania konwencji przed konfiguracją. W strukturze projektu znajdują się trzy katalogi specjalne odpowiadające elementom architektury MVC — *Controllers*, *Models* i *Views*. Przeznaczenie wymienionych katalogów jest oczywiste.



Rysunek 1.4. Struktura katalogów w projekcie ASP.NET MVC

Kiedy spojrzysz na zawartość tych katalogów, od razu dostrzeżesz stosowanie kolejnych konwencji. Na przykład katalog *Controllers* zawiera wszystkie klasy kontrolerów, które z kolei stosują konwencję polegającą na dodaniu przyrostka *Controller* do nazwy klasy. Platforma wykorzystuje

tę konwencję do rejestrowania kontrolerów aplikacji podczas jej uruchamiania oraz do powiązania kontrolerów z odpowiadającymi im trasami.

Następnie spójrz na katalog *Views*. Oprócz oczywistej konwencji polegającej na umieszczeniu wszystkich widoków aplikacji w tym katalogu zawiera on kilka podkatalogów. Znajdziesz tutaj podkatalog *Shared* oraz opcjonalne podkatalogi przeznaczone do przechowywania widoków dla poszczególnych kontrolerów. Taka konwencja ułatwia pracę programistom, ponieważ zwalnia ich z konieczności wyraźnego podawania lokalizacji, z których widoki mają być wyświetlane użytkownikom. Zamiast tego programista po prostu podaje nazwę widoku, np. *Index*, natomiast platforma stara się odnaleźć wymieniony widok w katalogu *Views*. Najpierw szuka widoku w podkatalogu przeznaczonym dla danego kontrolera; jeśli widok nie znajduje się we wspomnianym podkatalogu, to platforma przechodzi do podkatalogu *Shared*.

W pierwszej chwili konwencja przed konfiguracją może wydawać się trywialna. Jednak wszystkie wprowadzone optymalizacje, nawet niewielkie, sumują się i mogą pomóc w zaoszczędzeniu czasu, zwiększyć niezawodność kodu i produktywność programisty.

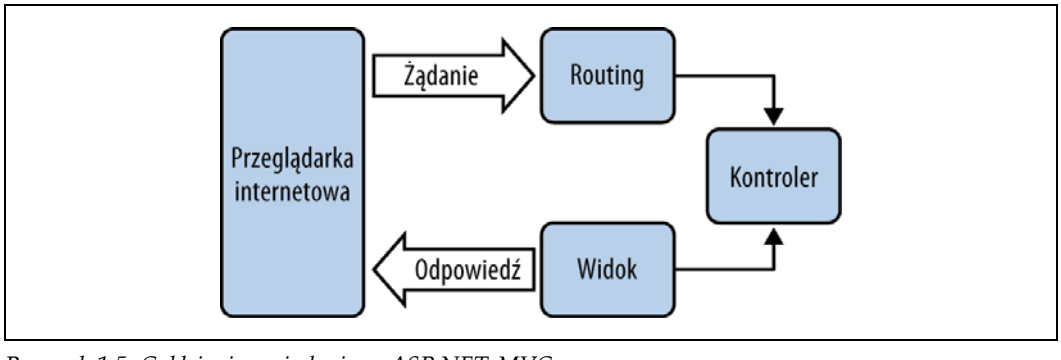
Uruchamianie aplikacji

Po utworzeniu projektu możesz nacisnąć klawisz *F5*. Nastąpi wówczas uruchomienie witryny internetowej ASP.NET MVC i jej wygenerowanie przez przeglądarkę internetową.

Nasze gratulacje! Właśnie utworzyłeś pierwszą aplikację ASP.NET MVC 4!

Gdy opadną Twoje emocje związane z wyświetleniem witryny w przeglądarce internetowej, prawdopodobnie będziesz się zastanawiał, co się właściwie stało? *Jak to działa?*

Na rysunku 1.5 pokazano proces przetwarzania żądania przez platformę ASP.NET MVC.



Rysunek 1.5. Cykl życiowy żądania w ASP.NET MVC

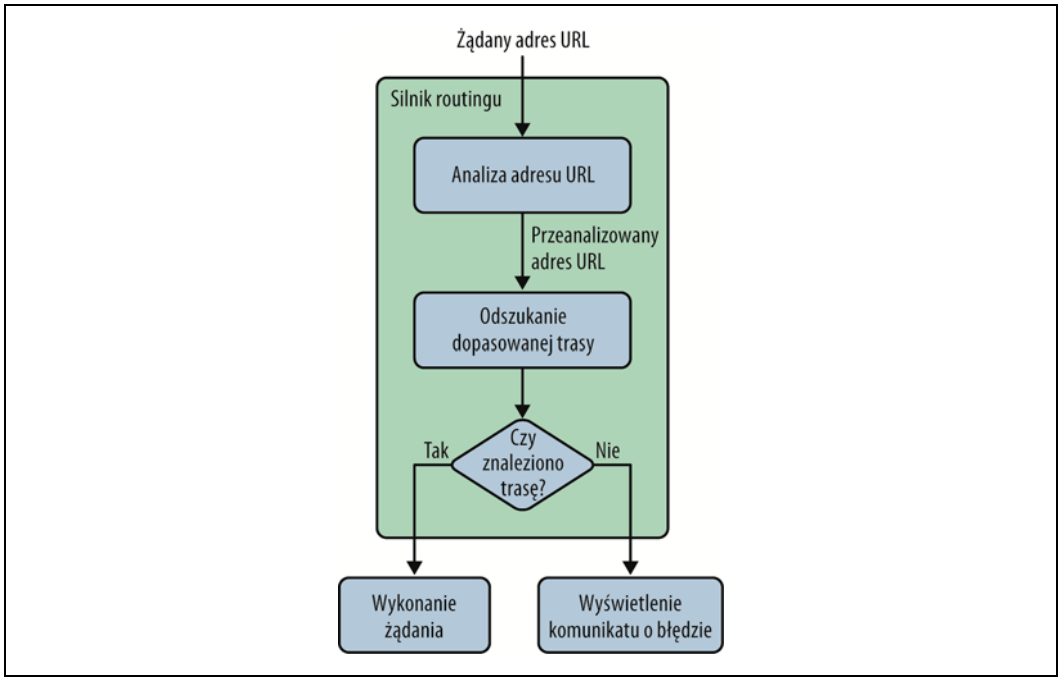
Wprawdzie pozostała część książki zawiera dokładne omówienie komponentów pokazanych na rysunku 1.5, ale w kolejnych kilku punktach będzie przedstawione ogólne objaśnienie tych podstawowych bloków stosowanych w ASP.NET MVC.

Routing

Cały ruch sieciowy ASP.NET MVC rozpoczyna się w taki sam sposób jak każdy inny ruch sieciowy, czyli od wykonania żądania do danego adresu URL. Oznacza to, że choć struktura

ASP.NET Routing nie została tutaj wymieniona z nazwy, to jednak jest podstawowym elementem każdego żądania ASP.NET MVC.

Ogólnie rzecz biorąc, ASP.NET Routing to po prostu system dopasowania do wzorca. W trakcie uruchamiania aplikacji w *tabeli routingu* struktury następuje rejestracja wzorców. Dzięki temu system routingu będzie wiedział, jak się zachować po otrzymaniu żądań dopasowanych do tych wzorców. Kiedy silnik routingu otrzyma żądanie, dopasowuje adres URL tego żądania do zarejestrowanych wzorców adresów URL (rysunek 1.6).



Rysunek 1.6. Routing ASP.NET

Jeśli silnik routingu znajdzie w tabeli routingu dopasowany wzorzec, to żądanie zostaje przekazane do odpowiedniej procedury w celu jego obsłużenia.

W przeciwnym razie, gdy adres URL w żądaniu nie zostanie dopasowany do żadnego zarejestrowanego wzorca, silnik routingu powinien zwrócić kod 404 odpowiedzi HTTP wraz z komunikatem o braku możliwości obsłużenia danego żądania.

Konfiguracja tras

Trasy ASP.NET MVC są odpowiedzialne za określenie, która metoda kontrolera (inaczej nazywana także *akcją kontrolera*) powinna zostać wykonana dla danego adresu URL. Trasa składa się z wymienionych poniżej właściwości.

Unikalna nazwa
: Nazwa, która będzie używana w charakterze odniesienia do danej trasy.

Wzorzec URL
: Prosta składnia wzorca analizująca adres URL i dzieląca go na segmenty.

Wartości domyślne
: Opcjonalny zestaw wartości domyślnych dla segmentów zdefiniowanych we wzorcu URL.

Ograniczenia
: Zestaw ograniczeń stosowanych w odniesieniu do wzorca URL, aby jeszcze bardziej zawęzić definicję dopasowanego adresu URL.

Domyślne szablony projektów ASP.NET MVC dodają ogólną trasę, która używa przedstawionej konwencji URL do podziału adresu URL danego żądania na trzy segmenty umieszczone w nawiasach klamrowych — `controller`, `action` i `id`.

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

Powyższy wzorzec trasy jest rejestrowany za pomocą metody `MapRoute()` wywoływanej podczas uruchamiania aplikacji (metoda znajduje się w pliku *App_Start/RouteConfig.cs*):

```
routes.MapRoute(
   "Default", // Nazwa trasy.
   "{controller}/{action}/{id}", // Adres URL wraz z parametrami.
   new { controller = "Home", action = "Index",
      id = UrlParameter.Optional } // Wartości domyślne parametrów.
);
```

Oprócz tego, że powyższa trasa dostarcza nazwę i adres URL wzorca, definiuje także dla parametrów zestaw wartości domyślnych. Będą one używane w przypadku, gdy adres URL zostanie dopasowany do wzorca trasy, ale nie będzie zawierał wartości dla wszystkich segmentów.

W tabeli 1.1 wymieniono listę adresów URL dopasowanych do powyższego wzorca trasy oraz odpowiednie wartości, które struktura routingu dostarczy dla poszczególnych segmentów.

Tabela 1.1. Wartości dostarczane dla adresów URL dopasowanych do omawianego wzorca trasy

Adres URL	Controller	Action	ID
/auctions/auction/1234	`AuctionController`	`Auction`	`1234`
/auctions/recent	`AuctionController`	`Recent`	
/auctions	`AuctionController`	`Index`	
/	`HomeController`	`Index`	

Pierwszy adres URL w tabeli (*/auctions/auction/1234*) to doskonałe dopasowanie do wzorca, ponieważ zawiera wartości dla wszystkich segmentów wzorca trasy. Ponieważ każdy następny adres w tabeli jest pozbawiony kolejnych segmentów na końcu adresu URL, to możesz zauważyć, że w miejscu wartości brakujących w adresie URL są używane wartości domyślne.

To jest bardzo ważny przykład, pokazujący, jak platforma ASP.NET MVC wykorzystuje koncepcję konwencji przed konfiguracją. Podczas uruchamiania aplikacji ASP.NET MVC wykrywa wszystkie kontrolery aplikacji poprzez wyszukanie dostępnych podzespołów dla klas implementujących interfejs `System.Web.Mvc.IController` (lub podzespołów wywodzących się z klas implementujących ten interfejs, np. `System.Web.Mvc.Controller`) *oraz* podzespołów, których nazwy klas zawierają przyrostek *Controller*. Kiedy struktura routingu używa tej listy w celu określenia kontrolerów, do których ma dostęp, to po prostu usuwa przyrostek *Controller* ze wszystkich nazw klas kontrolerów. Jeśli zatem chcesz się odwołać do kontrolera, to możesz

użyć jego skróconej nazwy; np. do kontrolera `AuctionsController` możesz się odwołać za pomocą słowa `Auctions`, do `HomeController` za pomocą słowa `Home`.

Co więcej, nazwy kontrolerów i wartości akcji w trasie nie rozróżniają wielkości liter. Z tego powodu wszystkie wymienione tutaj żądania (*/Auctions/Recent*, */auctions/Recent*, */auctions/recent*, a nawet */aucTionS/rEceNt*) zostaną odczytane prawidłowo — akcja `Recent` kontrolera `AuctionsController`.

Wzorce tras URL są określane względem katalogu głównego aplikacji i dlatego nie muszą rozpoczynać się od ukośnika (/) lub wskaźnika ścieżki wirtualnej (~/). Wzorce trasy zawierające wymienione znaki są nieprawidłowe i spowodują zgłoszenie wyjątku przez system routingu.

Jak mogłeś zauważyć, trasy URL mogą zawierać istotne informacje, które silnik routingu potrafi wyodrębnić. Jednak w celu przetworzenia żądania ASP.NET MVC silnik routingu musi mieć możliwość ustalenia dwóch informacji o znaczeniu krytycznym — *kontrolera* i *akcji*. Następnie silnik routingu będzie mógł przekazać wspomniane wartości do środowiska uruchomieniowego ASP.NET MVC w celu utworzenia i uruchomienia wskazanej akcji w odpowiednim kontrolerze.

Kontrolery

W kontekście wzorca architekturalnego MVC *kontroler* odpowiada na dane wejściowe dostarczane przez użytkownika (np. kliknięcie przycisku *Zapisz*) oraz współpracuje z warstwami modelu, widoku i (bardzo często) dostępu do danych. W aplikacji ASP.NET MVC kontroler to klasa zawierająca metody wywoływane przez strukturę routingu w celu przetworzenia żądania.

Aby zobaczyć przykład kontrolera ASP.NET MVC, spójrz na klasę `HomeController`, którą znajdziesz w pliku *Controllers/HomeController.cs*:

```
using System.Web.Mvc;

namespace Ebuy.Website.Controllers
{
   public class HomeController : Controller
   {
      public ActionResult Index()
      {
         ViewBag.Message = "Zmodyfikuj ten szablon, aby szybko uruchomić swoją aplikację
            platformy ASP.NET MVC.";
         return View();
      }

      public ActionResult About()
      {
         ViewBag.Message = "Twoja strona z opisem aplikacji.";
         return View();
      }

      public ActionResult Contact()
      {
         ViewBag.Message = "Strona z Twoimi danymi kontaktowymi.";
         return View();
      }
   }
}
```

Akcje kontrolera

Jak możesz się przekonać, klasa kontrolera sama w sobie nie jest niczym specjalnym, to znaczy nie odróżnia się zbytnio od innych klas .NET. W rzeczywistości *metody* w klasie kontrolera — nazywane również *akcjami kontrolera* — wykonują całą pracę związaną z przetworzeniem żądania.

Wyrażenia *kontroler* i *akcje kontrolera* są używane wymiennie, także w niniejszej książce. Powód jest prosty — architektura MVC nie rozróżnia ich. Jednak na platformie ASP.NET MVC najczęściej stosowane jest wyrażenie *akcje kontrolera*, ponieważ zawierają one rzeczywistą logikę odpowiedzialną za przetworzenie żądania.

Na przykład przedstawiona już wcześniej klasa `HomeController` zawiera trzy akcje — `Index`, `About` i `Contact`. Dlatego też w przypadku domyślnego wzorca trasy `{controller}/{action}/{id}` po wykonaniu żądania do adresu URL */Home/About* struktura routingu ustali, że żądanie powinno zostać przetworzone przez metodę `About()` klasy `HomeController`. Następnie platforma ASP.NET MVC utworzy nowy egzemplarz klasy `HomeController` i wywoła jej metodę `About()`.

W omawianym przypadku działanie metody `About()` jest bardzo proste — za pośrednictwem właściwości `ViewBag` (więcej na jej temat znajdziesz w dalszej części rozdziału) dane zostaną przekazane do widoku. Następnie platforma ASP.NET MVC otrzymuje polecenie wyświetlenia widoku o nazwie *About* poprzez wywołanie metody `View()`, której wartością zwrotną jest `ActionResult` typu `ViewResult`.

Obiekt ActionResult

Trzeba w tym miejscu powiedzieć o jednej ważnej kwestii — zadaniem kontrolera jest poinformowanie platformy ASP.NET MVC, *co* powinna zrobić dalej, a nie *jak* powinna to zrobić. Komunikacja odbywa się poprzez użycie obiektu `ActionResult`, czyli wartości zwrotnej oczekiwanej od każdej akcji kontrolera.

Gdy na przykład kontroler ma wyświetlić widok, to zwracając wartość typu `ViewResult`, nakazuje platformie ASP.NET MVC wyświetlenie widoku. Sam kontroler nie generuje widoku. Luźne powiązanie to kolejny doskonały przykład stosowanej w akcji zasady separacji zadań (*co* powinno zostać wykonane zamiast *jak* powinno zostać wykonane).

Wprawdzie każda akcja kontrolera powinna zwrócić obiekt `ActionResult`, ale rzadko będziesz go tworzył ręcznie. Zamiast tego wykorzystasz metody pomocnicze dostarczane przez klasę podstawową `System.Web.Mvc.Controller`. Poniżej wymieniono te metody pomocnicze.

`Content()`
: Metoda zwraca wartość `ContentResult`, która powoduje wygenerowanie wskazanego tekstu, np. *Witaj, świecie*.

`File()`
: Metoda zwraca wartość `FileResult`, która powoduje wyświetlenie zawartości wskazanego pliku, np. PDF.

`HttpNotFound()`
: Metoda zwraca wartość `HttpNotFoundResult`, która powoduje wygenerowanie odpowiedzi HTTP zawierającej kod 404.

`JavaScript()`
: Metoda zwraca wartość `JavaScripttResult`, która powoduje wygenerowanie skryptu JavaScript. np. `function hello() {alert(Witaj, świecie!);}`.

`Json()`
: Metoda zwraca wartość `JsonResult`, która przeprowadza serializację obiektu i generuje go w formacie JSON (*JavaScript Object Notation*), np. `{"Komunikat":Witaj, świecie!}`.

`PartialView()`
: Metoda zwraca wartość `PartialResult`, która generuje jedynie treść widoku, np. widok bez uwzględnienia jego układu.

`Redirect()`
: Metoda zwraca wartość `RedirectResult`, która powoduje wygenerowanie odpowiedzi HTTP zawierającej kod 302 (tymczasowe przekierowanie). Tego rodzaju odpowiedź powoduje przekierowanie użytkownika na podany adres URL, np. „302 *http://www.ebuy.com/auctions/recent*". Istnieje jeszcze metoda podobna do tej (`RedirectPermanent()`), która również zwraca wartość `RedirectResult`, ale używa kodu 301 odpowiedzi HTTP, wskazującego na trwałe przekierowanie zamiast tymczasowego.

`RedirectToAction()` i `RedirectToRoute()`
: Metody te działają podobnie jak `Redirect()`, ale to platforma dynamicznie ustala zewnętrzny adres URL poprzez wykonanie zapytania do silnika routingu. Podobnie jak w przypadku `Redirect()`, także te dwie metody pomocnicze posiadają wersje obsługujące trwałe przekierowanie: `RedirectToActionPermanent()` i `RedirectToRoutePermanent()`.

`View()`
: Metoda zwraca wartość `ViewResult`, która powoduje wygenerowanie widoku.

Jak możesz się przekonać, przeglądając powyższą listę, platforma dostarcza wartości `ActionResult` niemal w każdej sytuacji. Jeżeli nie wybierzesz żadnej z powyższych metod, zawsze możesz utworzyć własną!

Chociaż wszystkie akcje kontrolera muszą zapewniać obiekt `ActionResult` wskazujący następne kroki, które powinny zostać podjęte w celu przetworzenia żądania, to jednak nie wszystkie akcje kontrolera muszą używać obiektu typu `ActionResult`. Akcje kontrolera mogą stosować dowolny typ zwrotny wywodzący się z `ActionResult`, a nawet używać innego typu.

Kiedy platforma ASP.NET MVC wykonuje akcję kontrolera zwracającą typ inny niż `ActionResult`, automatycznie opakowuje wartość zwrotną typem `ContentResult`, a następnie wyświetla ją w niezmodyfikowanej postaci.

Parametry akcji

Akcje kontrolerów są podobne do pozostałych metod. Wykonywana akcja kontrolera może nawet określić liczbę parametrów uzupełnianych przez ASP.NET MVC na podstawie informacji znajdujących się w żądaniu. Taka funkcjonalność nosi nazwę *dołączania modelu* (ang. *model binding*) — to jedna z najużyteczniejszych funkcji platformy ASP.NET MVC, oferująca bardzo duże możliwości.

Zanim przeanalizujemy sposób działania dołączania modelu, w pierwszej kolejności warto wykonać krok wstecz i zapoznać się z przykładem „tradycyjnej" interakcji z wartościami żądania:

```
public ActionResult Create()
{
   var auction = new Auction() {
      Title = Request["title"],
      CurrentPrice = Decimal.Parse(Request["currentPrice"]),
      StartTime = DateTime.Parse(Request["startTime"]),
      EndTime = DateTime.Parse(Request["endTime"]),
   };
   //...
}
```

W powyższym fragmencie kodu kontroler tworzy i przypisuje wartości właściwościom nowego obiektu `Auction`. Przypisywane wartości są pobierane wprost z żądania. Ponieważ pewne właściwości obiektu `Auction` są zdefiniowane jako typy podstawowe inne niż `string`, akcja musi skonwertować każdą wartość żądania na odpowiedni typ.

Powyższy przykład może wydawać się oczywisty i prosty, ale tak naprawdę jest raczej zawodny — jeżeli konwersja dowolnej wartości zakończy się niepowodzeniem, to działanie całej akcji również zakończy się niepowodzeniem. Zastosowanie różnych metod `TryParse()` może pomóc w uniknięciu zgłoszenia większości wyjątków, ale użycie tych metod oznacza także konieczność utworzenia dodatkowego kodu.

Efektem ubocznym przedstawionego podejścia jest to, że każda akcja pozostaje ściśle określona. Wadą tworzenia ściśle określonego kodu akcji jest obciążenie dla programisty związane z tworzeniem dodatkowego kodu i konieczność pamiętania o jego przygotowaniu za każdym razem, gdy istnieje taka potrzeba. Duża ilość kodu przesłania rzeczywisty cel, którym w omawianym przykładzie jest dodanie nowej aukcji (`Auction`) do systemu.

Podstawy dołączania modelu

Mechanizm dołączania modelu nie tylko pozwala na uniknięcie konieczności tworzenia ściśle określonego kodu, ale również jest bardzo łatwy do zastosowania — nawet nie musisz o nim myśleć.

Poniżej przedstawiono ten sam kontroler, który widziałeś już wcześniej, ale tym razem został utworzony z użyciem techniki dołączania modelu:

```
public ActionResult Create(
   string title, decimal currentPrice,
   DateTime startTime, DateTime endTime
)
{
   var auction = new Auction() {
      Title = title,
      CurrentPrice = currentPrice,
      StartTime = startTime,
      EndTime = endTime,
   };
   //...
}
```

Zamiast pobierać wartości wprost z `Request`, akcja zadeklarowała je jako parametry. Kiedy platforma ASP.NET MVC wykonuje metodę, próbuje zdefiniować parametry akcji, przypisując im wartości pobrane z żądania, jak to zostało pokazane w poprzednim przykładzie.

Zwróć uwagę na jedno — wprawdzie nie uzyskujemy bezpośredniego dostępu do słownika `Request`, ale nazwy parametrów są bardzo ważne, ponieważ nadal odpowiadają wartościom słownika `Request`.

Jednak obiekt `Request` to nie tylko miejsce, z którego mechanizm dołączania modelu na platformie ASP.NET MVC pobiera wartości. Standardowo platforma szuka wartości w kilku miejscach, między innymi w danych trasy, parametrach ciągu tekstowego zapytania, w wartościach zapytania typu `POST`, a nawet w serializowanych obiektach JSON. Na przykład fragment kodu przedstawiony na listingu 1.1 pobiera wartość `id` z adresu URL poprzez po prostu zdefiniowanie parametru o takiej samej nazwie.

Listing 1.1. Pobranie wartości id z adresu URL (np. /auctions/auction/123)

```
public ActionResult Auction(long id)
{
   var context = new EBuyContext();
   var auction = context.Auctions.FirstOrDefault(x => x.Id == id);
   return View("Auction", auction);
}
```

To, gdzie i jak mechanizm dołączania modelu na platformie ASP.NET MVC znajduje wspomniane wartości, można skonfigurować i nawet rozszerzyć. Więcej informacji na temat mechanizmu dołączania modelu na platformie ASP.NET MVC znajdziesz w rozdziale 8.

Jak pokazano w powyższych przykładach, mechanizm dołączania modelu pozwala platformie ASP.NET MVC na obsługę większości nieciekawego, standardowego kodu. W ten sposób logika umieszczana w akcji może koncentrować się na dostarczaniu rzeczywistej funkcjonalności, a sam kod akcji pozostanie konkretny i bez wątpienia będzie znacznie czytelniejszy.

Dołączanie modelu w przypadku skomplikowanych obiektów

Zastosowanie mechanizmu dołączania modelu nawet w przypadku prostych, podstawowych typów może mieć całkiem duży wpływ na znaczne zwiększenie czytelności kodu. Jednak rzeczywistość bywa dużo bardziej skomplikowana — jedynie w najprostszych scenariuszach wykorzystuje się tylko kilka parametrów. Na szczęście platforma ASP.NET MVC obsługuje mechanizm dołączania modelu zarówno w przypadku prostych, jak i złożonych typów.

Przedstawiony poniżej przykład pokazuje jeszcze jedno podejście do akcji `Create`. Tym razem pomijamy pośrednie typy proste i przeprowadzamy bezpośrednie łączenie z egzemplarzem `Auction`:

```
public ActionResult Create(Auction auction)
{
   // ...
}
```

Powyższa akcja jest odpowiednikiem tej, którą widziałeś już w poprzednim przykładzie. To prawda, mechanizm dołączania modelu w ASP.NET MVC wyeliminował cały kod wymagany do utworzenia i przypisania wartości nowemu egzemplarzowi `Auction`! Dzięki przedstawionemu przykładowi masz okazję przekonać się o prawdziwej potędze mechanizmu dołączania modelu.

Filtry akcji

Filtry akcji zapewniają prostą, choć jednocześnie oferującą bardzo duże możliwości technikę modyfikacji bądź usprawnienia potoku ASP.NET MVC poprzez „wstawienie" logiki w określonych miejscach. W ten sposób można ułatwić rozwiązanie pewnych problemów, które mogą pojawić się w wielu (lub wszystkich) komponentach aplikacji. Rejestrowanie zdarzeń w aplikacji to klasyczny przykład — ma zastosowanie w odniesieniu do wszystkich komponentów aplikacji, niezależnie od ich podstawowego przeznaczenia.

Logika filtru akcji jest wprowadzana poprzez zastosowanie klasy `ActionFilterAttribute` w odniesieniu do akcji kontrolera, aby zmienić sposób wykonania danej akcji. Przykład takiego rozwiązania przedstawiono w poniższym fragmencie kodu, który stosując atrybut `Authorize-Attribute`, chroni akcję kontrolera przed uzyskaniem do niej nieautoryzowanego dostępu.

```
[Authorize]
public ActionResult Profile()
{
   // Pobranie informacji o profilu bieżącego użytkownika.
   return View();
}
```

Platforma ASP.NET MVC zawiera całkiem sporą liczbę filtrów akcji przeznaczonych do stosowania w najczęściej spotykanych scenariuszach. Z filtrami akcji będziesz spotykał się w trakcie lektury niniejszej książki, ponieważ pomagają one wykonywać różne zadania w schludny, luźny sposób.

Filtr akcji to doskonały sposób zastosowania własnej logiki w całej witrynie internetowej. Pamiętaj o możliwości samodzielnego tworzenia własnych filtrów akcji poprzez rozbudowę klasy podstawowej `ActionFilterAttribute` lub innego dowolnego filtru akcji platformy ASP.NET MVC.

Widoki

Na platformie ASP.NET MVC akcja kontrolera, która jest odpowiedzialna za wyświetlenie treści HTML użytkownikowi, zwraca egzemplarz klasy `ActionResult` typu `ViewResult`, który potrafi wygenerować treść w odpowiedzi. Kiedy trzeba wygenerować widok, platforma ASP.NET MVC będzie szukała widoku, opierając się na nazwie podanej przez kontroler.

Przeanalizuj akcję `Index` w kontrolerze `HomeController`:

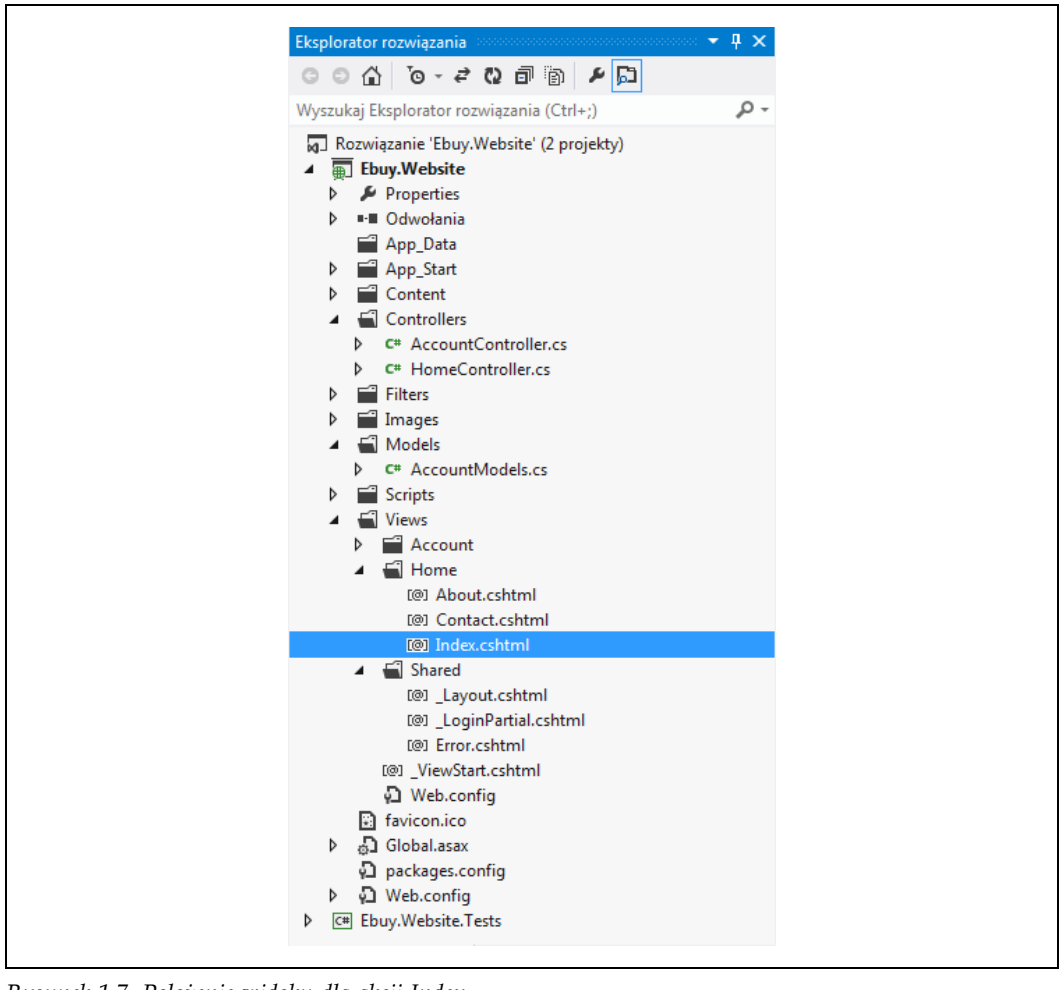
```
public ActionResult Index()
{
   ViewBag.Message = "Zmodyfikuj ten szablon, aby szybko uruchomić swoją aplikację
      platformy ASP.NET MVC.";
   return View();
}
```

Do utworzenia `ViewResult` powyższa akcja wykorzystuje zalety metody pomocniczej `View()`. Wywołanie metody `View()` bez żadnych parametrów — jak w powyższym przykładzie — nakazuje platformie ASP.NET MVC odszukanie widoku o nazwie takiej samej jak nazwa bieżącej akcji kontrolera. W omawianym przykładzie platforma będzie szukała widoku o nazwie *Index*. Powstaje jednak pytanie, gdzie będzie szukała tego widoku.

Wyszukiwanie widoków

Platforma ASP.NET MVC stosuje się do konwencji, według której wszystkie widoki aplikacji znajdują się w podkatalogu *Views* katalogu głównego witryny internetowej. Ujmując rzecz dokładniej, platforma oczekuje znalezienia widoku w podkatalogach o nazwach takich samych jak nazwy kontrolerów obsługujących dane widoki.

Jeśli zatem platforma chce wyświetlić widok dla akcji `Index` kontrolera `HomeController`, to w katalogu */Views/Home* będzie szukała pliku o nazwie *Index*. Na rysunku 1.7 możesz zobaczyć, że szablon projektu automatycznie utworzył widok i zapisał go w pliku *Index.cshtml*.



Rysunek 1.7. Położenie widoku dla akcji Index

Kiedy w wymienionym podkatalogu platforma nie znajdzie widoku o szukanej nazwie, to przystąpi do przeszukiwania katalogu */Views/Shared*.

Katalog */Views/Shared* to doskonałe miejsce na umieszczenie widoków wykorzystywanych przez wiele kontrolerów.

Po znalezieniu widoku należącego do danej akcji otwórz plik i przekonaj się, co zawiera — znaczniki HTML i kod. To jednak nie są *dowolne* znaczniki HTML i dowolny kod — to *Razor*!

Poznaj Razor

Razor to składnia pozwalająca na połączenie kodu i treści w płynny i czytelny sposób. Wprawdzie Razor wprowadza kilka symboli i słów kluczowych, ale na pewno nie jest nowym językiem. Składnia Razor pozwala na tworzenie kodu w prawdopodobnie znanych Ci już językach, takich jak C# lub Visual Basic.NET.

Nauka składni Razor jest prosta, ponieważ tak naprawdę wykorzystujesz posiadane już umiejętności, a nie uczysz się od początku zupełnie nowego języka. Jeśli wiesz, jak tworzyć kod HTML i .NET w językach C# lub Visual Basic.NET, to bardzo łatwo możesz stosować znaczniki, jak pokazano poniżej:

```
<div>Ta strona została wygenerowana dnia @DateTime.Now</div>
```

Powyższy wiersz kodu powoduje wygenerowanie następujących danych:

```
<div>Ta strona została wygenerowana dnia 2013-01-10 12:09:32</div>
```

Przykład rozpoczyna się od standardowego znacznika HTML (`<div>`). Następnie znajduje się na stałe zdefiniowany tekst i dalej tekst generowany dynamicznie jako wynik odwołania się do właściwości .NET o nazwie `System.DateTime.Now`. Na końcu mamy zamykający znacznik HTML (`</div>`).

Wbudowany w Razor inteligentny analizator składni pozwala programistom na większą ekspresję podczas tworzenia logiki oraz znacznie ułatwia połączenie kodu i znaczników. Wprawdzie składnia Razor może wydawać się odmienna od innych składni znaczników (np. od składni Web Forms), to jednak prowadzi do tego samego celu, jakim jest wygenerowanie kodu HTML.

Aby to zilustrować, przeanalizujemy poniższe fragmenty kodu przedstawiające przykłady najczęstszych sytuacji, w których implementowana jest składnia zarówno Razor, jak i Web Forms.

Poniżej przedstawiono polecenie `if-else` zaimplementowane z użyciem składni Web Forms:

```
<% if(User.IsAuthenticated) { %>
    <span>Witaj, <%: User.Username %>!</span>
<% } %>
<% else { %>
    <span>Proszę <%: Html.ActionLink("zaloguj się") %></span>
<% } %>
```

Ten sam fragment kodu, ale zaimplementowany z użyciem składni Razor:

```
@if(User.IsAuthenticated) {
    <span>Witaj, @User.Username!</span>
} else {
    <span>Proszę @Html.ActionLink("zaloguj się")</span>
}
```

Kolejny fragment kodu przedstawia pętlę `foreach` zaimplementowaną z użyciem składni Web Forms:

```
<ul>
<% foreach(var auction in auctions) { %>
    <li><a href="<%: auction.Href %>"><%: auction.Title %></a></li>
<% } %>
</ul>
```

Ten sam fragment kodu, ale zaimplementowany z użyciem składni Razor:

```
<ul>
@foreach( var auction in auctions) {
    <li><a href="@auction.Href">@auction.Title</a></li>
}
</ul>
```

Wprawdzie powyższe fragmenty kodu używają odmiennej składni, ale ostatecznie powodują wygenerowanie tego samego kodu HTML.

Odróżnianie kodu od znaczników

Składnia Razor zapewnia dwa sposoby odróżnienia kodu od znaczników — pakiety kodu i bloki kodu.

Pakiety kodu

Pakiety kodu to proste wyrażenia wykonywane w miejscu ich zdefiniowania. Mogą być połączone z tekstem i przedstawiają się następująco:

```
Niezalogowany: @Html.ActionLink("Login", "Zaloguj się")
```

Wyrażenie rozpoczyna się tuż po symbolu `@`. Składnia Razor jest wystarczająco inteligentna, aby wiedzieć, że nawias zamykający oznacza koniec danego wyrażenia.

Przedstawiony powyżej wiersz kodu powoduje wygenerowanie następujących danych:

```
Niezalogowany: <a href="/Login">Zaloguj się</a>
```

Zauważ, że pakiet kodu zawsze musi zwrócić widokowi kod znaczników przeznaczony do wygenerowania. Jeżeli przygotujesz pakiet kodu, którego wartością zwrotną będzie `void`, to w trakcie generowania widoku pojawi się komunikat o błędzie.

Blok kodu

Blok kodu to sekcja widoku zawierająca tylko i wyłącznie kod zamiast połączenia kodu i znaczników. Razor definiuje blok kodu jako dowolną sekcję szablonu Razor umieszczoną pomiędzy znakami `@{` a znakiem `}`. Znaki `@{` oznaczają początek bloku. Następnie można umieścić dowolną liczbę linii poprawnego kodu. Koniec bloku kodu jest oznaczony znakiem `}`.

Pamiętaj, że kod umieszczony w bloku kodu nie działa na takiej samej zasadzie jak pakiet kodu. To jest po prostu zwykły kod, który musi spełniać reguły narzucane przez użyty język programowania. Na przykład każdy wiersz kodu w języku C# musi być zakończony średnikiem (`;`), podobnie jak w przypadku umieszczenia danego wiersza w pliku klasy (*.cs*).

Poniżej przedstawiono przykład typowego bloku kodu:

```
@{
   LayoutPage = "~/Views/Shared/_Layout.cshtml";
   View.Title = "Aukcja " + Model.Title;
}
```

Blok kodu nie generuje żadnej treści w widoku. Zamiast tego pozwala Ci na tworzenie dowolnego kodu niewymagającego użycia wartości zwrotnej.

Ponadto zmienne zdefiniowane w blokach kodu mogą być stosowane w pakietach kodu znajdujących się w tym samym zasięgu. Oznacza to, że zmienna zdefiniowana w zasięgu pętli `foreach` lub podobnego kontenera będzie dostępna jedynie w tym kontenerze. Natomiast zmienna zdefiniowana na najwyższym poziomie widoku (nie w żadnym kontenerze) będzie dostępna dla dowolnego bloku kodu lub pakietu kodu znajdujących się w tym samym widoku.

Aby lepiej zrozumieć tę koncepcję, spójrz na poniższy fragment kodu, w którym zdefiniowano zmienne o różnych zasięgach:

```
@{
   // Zmienne title i bids są dostępne dla całego widoku.
   var title = Model.Title;
   var bids = Model.Bids;
}

<h1>@title<h1>
<div class="items">
<!-- Iteracja przez obiekty w zmiennej bids. -->
@foreach(var bid in bids) {
   <!-- Zmienna bid jest dostępna jedynie w pętli foreach. -->
   <div class="bid">
      <span class="bidder">@bid.Username</span>
      <span class="amount">@bid.Amount</span>
   </div>
}

<!-- Poniższy wiersz spowoduje wystąpienie błędu - zmienna bid nie istnieje w tym zasięgu! -->
<div>Last Bid Amount: @bid.Amount</div>
</div>
```

Bloki kodu są przeznaczone do wykonywania kodu w szablonie, a nie do generowania czegokolwiek w widoku. W przeciwieństwie do pakietu kodu, który musi dostarczyć widokowi wartość zwrotną, ewentualna wartość zwrotna bloku kodu jest całkowicie ignorowana przez widok.

Układy graficzne

Razor oferuje możliwość zachowania spójnego wyglądu i działania całej witryny internetowej dzięki zastosowaniu *układu graficznego*. Układ graficzny to jeden widok działający w charakterze szablonu dla wszystkich pozostałych widoków i definiujący układ oraz styl strony stosowany w całej witrynie.

Szablon układu graficznego najczęściej zawiera podstawowy kod znaczników (skrypty, arkusze stylów CSS i strukturalne elementy HTML, takie jak nawigacja oraz kontenery przeznaczone na treść) wraz z określonymi miejscami, w których widok może zdefiniować treść. Następnie każdy widok w witrynie odwołuje się do tak przygotowanego układu graficznego, dołączając jedynie odpowiednią treść we wskazanych miejscach.

Spójrz na podstawowy plik układu graficznego Razor (*_Layout.cshtml*):

```
<!DOCTYPE html>

<html lang="en">
   <head>
      <meta charset="utf-8" />
      <title>@View.Title</title>
   </head>
   <body>
      <div class="header">
         @RenderSection("Header")
      </div>

      @RenderBody()

      <div class="footer">
         @RenderSection("Footer")
      </div>
   </body>
</html>
```

Plik układu graficznego zawiera podstawową strukturę HTML dla całej witryny internetowej. Układ opiera się na zmiennych (np. `@View.Title`) oraz funkcjach pomocniczych, takich jak `@RenderSection([Nazwa sekcji])` i `@RenderBody()`, przeznaczonych do interakcji z poszczególnymi widokami.

Po zdefiniowaniu układu graficznego Razor widoki odwołują się do tak przygotowanego układu i po prostu dostarczają treść, która jest umieszczona w miejscach wskazanych w układzie.

Poniżej przedstawiono podstawową stronę dla treści, odwołującą się do poprzednio zdefiniowanego pliku *_Layout.cshtml*:

```
@{ Layout = "~/_Layout.cshtml"; }

@section Header {
<h1>EBuy - aukcje internetowe<h1>
}

@section Footer {
    Copyright @DateTime.Now.Year
}

<div class="main">
   To jest podstawowa treść strony.
</div>
```

Układy graficzne Razor i widoki z treścią budowane na podstawie tych układów są ze sobą łączone podobnie jak puzzle, każdy z nich może definiować jedną część lub więcej części całej strony. Po zebraniu wszystkich wymaganych fragmentów następuje utworzenie pełnej strony internetowej.

Widoki częściowe

Chociaż układy graficzne są użytecznym rozwiązaniem pozwalającym na ponowne używanie fragmentów kodu w celu zapewnienia spójnego wyglądu i działania wielu stron witryny internetowej, to jednak pewne scenariusze mogą wymagać nieco innego podejścia.

Bardzo często spotykanym scenariuszem jest potrzeba wyświetlenia w wielu miejscach witryny pewnych informacji wysokiego poziomu, ale jedynie na kilku konkretnych stronach i jednocześnie w różnych miejscach tych stron.

W wielu miejscach (np. na stronie wyników wyszukiwania lub na stronie głównej) witryna aukcji internetowych EBuy może wygenerować skróconą listę informacji szczegółowych o aukcjach, to znaczy zawierającą jedynie tytuł aukcji, aktualną cenę i prawdopodobnie miniaturkę przedmiotu.

ASP.NET MVC obsługuje tego rodzaju scenariusze dzięki widokom częściowym.

Widok częściowy to widok zawierający docelowy kod znaczników zaprojektowany w taki sposób, aby był generowany jako część większego widoku. Przedstawiony poniżej fragment kodu demonstruje widok częściowy przeznaczony do wyświetlenia wspomnianej wcześniej, skróconej listy szczegółowych informacji o aukcjach:

```
@model Auction

<div class="auction">
   <a href="@Model.Url">
      <img src="@Model.ImageUrl" />
   </a>
   <h4><a href="@Model.Url">@Model.Title</a></h4>
   <p>Aktualna cena: @Model.CurrentPrice</p>
</div>
```

Aby wygenerować powyższy fragment kodu jako widok częściowy, należy go po prostu zapisać w oddzielnym pliku widoku (np. */Views/Shared/Auction.cshtml*) i użyć jednej z metod pomocniczych HTML oferowanych przez ASP.NET MVC — `Html.Partial()` — do wygenerowania widoku jako części innego widoku.

Gotowe rozwiązanie w działaniu przedstawiono w poniższym fragmencie kodu, który przeprowadza iterację przez obiekty aukcji oraz używa omówionego wcześniej widoku częściowego do wygenerowania kodu HTML dla każdej aukcji.

```
@model IEnumerable<Auction>

<h2>Wyniki wyszukiwania</h2>

@foreach(var auction in Model) {
   @Html.Partial("Auction", auction)
}
```

Zwróć uwagę, że pierwszym parametrem metody pomocniczej `Html.Partial()` jest ciąg tekstowy zawierający nazwę widoku podaną bez rozszerzenia.

Powód jest następujący — metoda pomocnicza `Html.Partial()` to jedynie prosta warstwa umieszczona na górze potężnego silnika widoku platformy ASP.NET MVC, generującego widok w sposób bardzo podobny jak w przypadku, gdy akcja kontrolera wywołuje metodę `View()`, której wartością zwrotną ma być `ViewResult`. Silnik używa nazwy widoku do odszukania i uruchomienia odpowiedniego widoku.

Dzięki temu widoki częściowe są tworzone i uruchamiane niemal dokładnie tak samo jak pozostałe rodzaje widoków. Jedyna różnica polega na tym, że widok częściowy został przeznaczony do wygenerowania jako część większego widoku.

Drugi parametr (w omawianym przykładzie to `auction`) funkcji pomocniczej `Html.Partial()` akceptuje model widoku częściowego, podobnie jak parametr modelu w metodzie `View(Nazwa_widoku, [Model])`. Drugi parametr jest opcjonalny. Kiedy nie zostanie podany, domyślnie będzie użyty model widoku, dla którego nastąpiło wywołanie metody pomocniczej `Html.Partial()`. Jeżeli na przykład w omawianym przykładzie zostanie pominięty parametr `auction`, platforma ASP.NET MVC umieści w jego miejscu właściwość `Model` widoku (która będzie typu `IEnumerable<Auction>`).

Przedstawione powyżej przykłady pokazują, jak widoki częściowe mogą dostarczać możliwych do ponownego używania fragmentów kodu znaczników, co pomaga w zmniejszeniu stopnia powielania kodu i skomplikowania widoków.

To jedna z użytecznych zalet widoków częściowych. W rozdziale 6. dowiesz się, jak wykorzystać widoki częściowe w celu dostarczenia prostego i efektywnego sposobu usprawnienia witryny internetowej poprzez użycie technologii AJAX.

Wyświetlanie danych

Architektura MVC jest oparta na oddzielnych komponentach — model, widok i kontroler — które razem współdziałają w celu wykonania określonego zadania. W tej koncepcji zadanie kontrolera przypomina działalność policjanta kierującego ruchem na drodze; kontroler po prostu koordynuje pracę różnych komponentów systemu podczas wykonywania logiki aplikacji. Przetwarzanie zwykle prowadzi do wygenerowania pewnego rodzaju danych, które powinny być wyświetlone użytkownikowi. Niestety, wyświetlanie danych użytkownikowi nie jest zadaniem kontrolera — do tego są przeznaczone widoki! Powstaje więc pytanie, w jaki sposób kontroler przekazuje widokowi dane przeznaczone do wyświetlenia.

Platforma ASP.NET MVC oferuje dwa sposoby przekazywania danych pomiędzy modelem, widokiem i kontrolerem — `ViewData` i `TempData`. Wymienione obiekty są słownikami dostępnymi jako właściwości zarówno kontrolerów, jak i widoków. Dlatego też przekazanie danych z kontrolera do widoku jest bardzo proste i sprowadza się do ustawienia odpowiedniej wartości w kontrolerze, jak w poniższym fragmencie kodu, pochodzącym z pliku *HomeController.cs*:

```
public ActionResult About()
{
   ViewData["Username"] = User.Identity.Username;

   ViewData["CompanyName"] = "EBuy: Witryna demonstracyjna ASP.NET MVC";
   ViewData["CompanyDescription"] =
      "EBuy to światowy lider w demonstrowaniu platformy ASP.NET MVC!";

   return View("About");
}
```

Następnie do wspomnianej wartości można odnieść się w widoku, jak w poniższym fragmencie kodu, pochodzącym z pliku *About.cshtml*:

```
<h1>@ViewData["CompanyName"]</h1>
<div>@ViewData["CompanyDescription"]</div>
```

Dostęp do wartości ViewData poprzez ViewBag

Kontrolery ASP.NET MVC i widoki udostępniające właściwość `ViewData` oferują także podobną właściwość, o nazwie `ViewBag`. Właściwość `ViewBag` to po prostu opakowanie dla `ViewData` udostępniające słownik `ViewData` jako obiekt typu `dynamic`.

Na przykład wszystkie odniesienia do wartości słownika `ViewData` przedstawione w poprzednim fragmencie kodu mogą być zastąpione odniesieniami do właściwości `dynamic` w obiekcie `ViewBag`:

```
public ActionResult About()
{
   ViewBag.Username = User.Identity.Username;

   ViewBag.CompanyName = "EBuy: Witryna demonstracyjna ASP.NET MVC";
   ViewBag.CompanyDescription = "EBuy to światowy lider w demonstrowaniu platformy
      ASP.NET MVC!";

   return View("About");
}
```

i:

```
<h1>@ViewBag.CompanyName</h1>
<div>@ViewBag.CompanyDescription</div>
```

Modele widoku

Oprócz zwykłego zachowania słownika obiekt `ViewData` oferuje także właściwość `Model` przedstawiającą podstawowy obiekt będący celem żądania. Wprawdzie właściwość `ViewData.Model` pod względem koncepcyjnym nie różni się od `ViewData["Model"]`, ale promuje model — staje się w ten sposób komponentem pierwszej kategorii, ważniejszym niż inne dane, które mogą znajdować się w żądaniu.

Poprzednie dwa przykłady kodu pokazały, że wartości `CompanyName` i `CompanyDescription` słownika są ze sobą powiązane. To tworzy doskonałą możliwość opakowania ich w modelu.

Spójrz na plik *CompanyInfo.cs*:

```
public class CompanyInfo
{
   public string Name { get; set; }
   public string Description { get; set; }
}
```

Oto definicja akcji `About` w pliku *HomeController.cs*:

```
public ActionResult About()
{
   ViewBag.Username = User.Identity.Username;

   var company = new CompanyInfo {
      Name = "EBuy: Witryna demonstracyjna ASP.NET MVC",
      Description = "EBuy to światowy lider w demonstrowaniu platformy ASP.NET MVC!",
   };

   return View("About", company);
}
```

Poniższy fragment kodu pochodzi z pliku *About.cshtml*:

```
@{ var company = (CompanyInfo)ViewData.Model; }

<h1>@company.Name</h1>
<div>@company.Description</div>
```

W powyższych fragmentach kodu odniesienia do wartości `CompanyName` i `CompanyDescription` słownika zostały połączone w egzemplarz nowej klasy o nazwie `CompanyInfo` (`company`). Uaktualniony fragment kodu w pliku *HomeController.cs* pokazuje również w działaniu przeciążoną metodę pomocniczą `View()`. Przeciążona wersja metody akceptuje nazwę żądanego widoku jako jej pierwszy parametr. Z kolei drugi parametr przedstawia obiekt, który będzie przypisany właściwości `ViewData.Model`.

Teraz nie trzeba bezpośrednio przypisywać wartości słownika, obiekt `company` jest przekazywany metodzie pomocniczej `View()` jako parametr `model`, a widok (*About.cshtml*) może pobrać lokalne odniesienie do obiektu `company` i uzyskać dostęp do jego wartości.

Ściśle określone widoki

Domyślnie właściwość `Model` dostępna dla widoków w składni Razor jest typu `dynamic`, co oznacza możliwość uzyskania dostępu do jej wartości bez potrzeby znajomości dokładnego typu właściwości.

Jednak biorąc pod uwagę statyczną naturę języka C# i oferowaną przez Visual Studio doskonałą obsługę listy IntelliSense dla widoków w składni Razor, bardzo często korzystne będzie wyraźne określenie typu modelu strony.

Na szczęście dzięki składni Razor to jest całkiem łatwe zadanie — wystarczy po prostu użyć słowa kluczowego `@model`, które wskazuje typ modelu:

```
@model CompanyInfo

<h1>@Model.Name</h1>
<div>@Model.Description</div>
```

Powyższy fragment kodu modyfikuje poprzedni przykład pliku *Auction.cshtml* i pozwala nam uniknąć konieczności dodawania zmiennej przejściowej w celu rzutowania na `ViewData.Model`. Zamiast tego użyte w wierszu pierwszym słowo kluczowe `@model` wskazuje, że typ modelu to `CompanyInfo`. W ten sposób wszystkie odniesienia do `ViewData.Model` mają ściśle określony typ i są dostępne bezpośrednio.

Metody pomocnicze HTML i URL

Podstawowym celem większości żądań sieciowych jest dostarczenie kodu HTML przeglądarce internetowej użytkownika. Platforma ASP.NET MVC oferuje więc rozwiązania pomagające w tworzeniu tego kodu. Oprócz składni Razor ASP.NET MVC oferuje wiele komponentów pomocniczych, odpowiedzialnych za proste i efektywne generowanie kodu HTML. Dwa najważniejsze komponenty pomocnicze to klasy `HtmlHelper` i `UrlHelper` udostępniane w kontrolerach i widokach w postaci właściwości odpowiednio `Html` i `Url`.

Poniżej przedstawiono przykład użycia tych dwóch komponentów pomocniczych:

```
<img src='@Url.Content("~/Content/images/header.jpg")' />
@Html.ActionLink("Strona główna", "Index", "Home")
```

Wygenerowany przez nie kod znaczników przedstawia się następująco:

```
<img src='/vdir/Content/images/header.jpg' />
<a href="/vdir/Home/Index">Strona główna</a>
```

W większości przypadków typy `HtmlHelper` i `UrlHelper` nie mają zbyt wielu własnych metod i są zaledwie szkieletami, które działają poprzez metody rozszerzające. W ten sposób oferują bardzo istotną możliwość ich rozbudowy. Odniesienia do dwóch wymienionych typów będziesz często spotykał w tej książce.

Wprawdzie istnieje znacznie więcej metod, które można by tu wymienić, ale powinieneś pamiętać o jednym — klasa `HtmlHelper` pomaga w wygenerowaniu kodu znaczników HTML, natomiast klasa `UrlHelper` pomaga w wygenerowaniu adresów URL. Z wymienionych komponentów pomocniczych korzystaj za każdym razem, gdy musisz wygenerować kod HTML lub adresy URL.

Modele

Po przedstawieniu kontrolerów i widoków omawianie definicji architektury MVC należy zakończyć prezentacją *modeli*, które zwykle są uznawane za najważniejszy komponent architektury MVC. Mógłbyś zapytać, dlaczego będą omówione jako ostatnie, skoro są takie ważne. Otóż warstwa modelu jest najtrudniejsza do wyjaśnienia, ponieważ zawiera całą logikę biznesową aplikacji, a sama logika jest inna w poszczególnych aplikacjach.

Z technicznego punktu widzenia model zazwyczaj składa się ze zwykłych klas, które udostępniają dane w postaci właściwości, a logikę w postaci metod. Wspomniane klasy są dostarczane w różnych postaciach, ale najczęściej stosowane są „modele danych" lub „modele domeny", których podstawowym zadaniem jest zarządzanie danymi.

Spójrz na poniższy fragment kodu przedstawiający klasę `Auction` — model obsługujący całą aplikację EBuy:

```
public class Auction
{
   public long Id { get; set; }
   public string Title { get; set; }
   public string Description { get; set; }
   public decimal StartPrice { get; set; }
   public decimal CurrentPrice { get; set; }
   public DateTime StartTime { get; set; }
   public DateTime EndTime { get; set; }
}
```

Wprawdzie na dalszych etapach prac dodamy do powyższej klasy `Auction` kolejne funkcje, np. uwierzytelnianie, ale omawiany fragment kodu nadal przedstawia typowy model — definiuje dane tworzące aukcję.

Rozbudowując klasę `Auction`, w dalszej części książki będziemy mieć na oku więcej rodzajów klas (np. usługi i klasy pomocnicze), które razem utworzą warstwę modelu architektury MVC.

Zebranie wszystkich komponentów w całość

Omówiliśmy już wszystkie komponenty tworzące aplikację ASP.NET MVC, ale koncentrowaliśmy się na kodzie wygenerowanym przez Visual Studio jako część szablonu projektu. Innymi słowy, tak naprawdę niczego *nie zrobiliśmy*. Pora więc to zmienić!

W tym podrozdziale dowiesz się, jak zupełnie od początku zaimplementować funkcję. Utworzymy wszystko, co będzie potrzebne do obsługi tej funkcji — wyświetlenia aukcji. Warto w tym miejscu przypomnieć, że każde żądanie ASP.NET MVC wymaga przynajmniej trzech rzeczy — trasy, akcji kontrolera oraz widoku (i opcjonalnie modelu).

Trasa

Aby określić wzorzec trasy używanej dla danej funkcji, w pierwszej kolejności trzeba ustalić dla niej wygląd adresu URL. W omawianym przykładzie decydujemy się na użycie względnie standardowego adresu URL w postaci *Auctions/Details/[Auction ID]*, czyli np. *http://www.ebuy.biz/Auctions/Details/1234*.

I tutaj mamy miłą niespodziankę — domyślna konfiguracja trasy już obsługuje przedstawiony adres URL!

Kontroler

Następnym krokiem jest utworzenie kontrolera przeznaczonego do przechowywania wszystkich akcji odpowiedzialnych za przetwarzanie żądania.

Ponieważ kontroler jest klasą implementującą interfejs kontrolera ASP.NET MVC, to *mógłbyś* do katalogu *Controllers* ręcznie dodać nową klasę wywodzącą się z `System.Web.Mvc.Controller` i rozpocząć definiowanie akcji kontrolera w tej klasie. Jednak Visual Studio oferuje pewne narzędzia wykonujące za programistę większość pracy podczas tworzenia nowego kontrolera. Po prostu kliknij prawym przyciskiem myszy katalog *Controllers*, a następnie z menu kontekstowego wybierz opcję *Dodaj/Kontroler…*. Na ekranie wyświetli się okno dialogowe (rysunek 1.8).

W pokazanym na rysunku 1.8 oknie dialogowym podajesz nazwę dla nowej klasy kontrolera (w omawianym przykładzie to `AuctionsController`), a następnie wybierasz szablon używany podczas tworzenia kontrolera. Możesz również podać opcje dodatkowe, dzięki którym zyskasz nieco większą kontrolę nad sposobem wygenerowania nowej klasy kontrolera przez platformę ASP.NET MVC.

Szablony kontrolerów

Omawiane okno dialogowe oferuje kilka różnych szablonów kontrolerów pomagających Ci w szybszym rozpoczęciu pracy nad tworzonym kontrolerem.

Pusty kontroler MVC
: To jest szablon domyślny i jednocześnie najprostszy z dostępnych. Nie oferuje żadnych opcji pozwalających na dostosowanie kontrolera do własnych potrzeb, ponieważ jest zbyt prosty, aby posiadać jakiekolwiek opcje. Tworzy zaledwie nowy kontroler o podanej nazwie i generuje w nim pojedynczą akcję o nazwie `Index`.