KOMPENDIUM WIEDZY O PLATFORMIE ASP.NET MVC 4!

Apress

ASP.NET MVC 4

Zaawansowane programowanie

Adam Freeman

Helion

Tytuł oryginału: Pro ASP.NET MVC 4

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-7299-8

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/aspmv4.zip

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

Printed in Poland.

Spis treści

ROZDZIAŁ 13.

Routing URL

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

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

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

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

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

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

Tworzenie projektu routingu

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

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

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

Listing 13.1. *Kod kontrolera HomeController*

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

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

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

Listing 13.2. *Kod kontrolera CustomerController*

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

namespace UrlsAndRoutes.Controllers {
    public class CustomerController : Controller {

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

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

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

Listing 13.3. *Kod kontrolera AdminController*

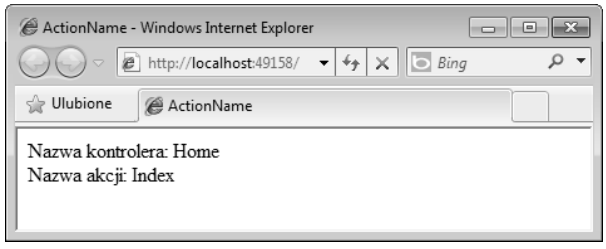
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace UrlsAndRoutes.Controllers {
    public class AdminController : Controller {

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

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

Listing 13.4. *Kod widoku ActionName.cshtml*

```
@{
    Layout = null;
}

<!DOCTYPE html>

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

Po uruchomieniu aplikacji otrzymasz komunikaty widoczne na rysunku 13.1.



Rysunek 13.1. *Efekt uruchomienia przykładowej aplikacji*

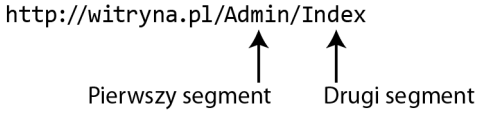
Wprowadzenie do wzorców URL

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

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

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

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



Rysunek 13.2. *Segmenty przykładowego adresu URL*

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

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

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

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

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

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

Tabela 13.1. *Dopasowanie adresów URL*

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

W tabeli 13.1 przedstawione są dwie kluczowe cechy wzorców URL:

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

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

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

Tworzenie i rejestrowanie prostej ścieżki

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 13.7. *Rejestrowanie ścieżki*

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

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

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

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

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

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

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

Listing 13.8. *Rejestrowanie ścieżki za pomocą metody MapRoute*

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

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

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

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

Test jednostkowy — testowanie przychodzących adresów URL

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

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

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

```
using System;

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

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

        private HttpContextBase CreateHttpContext(string targetUrl = null,
                                        string httpMethod = "GET") {
            // tworzenie imitacji żądania
            Mock<HttpRequestBase> mockRequest = new Mock<HttpRequestBase>();
            mockRequest.Setup(m => m.AppRelativeCurrentExecutionFilePath)
                .Returns(targetUrl);
            mockRequest.Setup(m => m.HttpMethod).Returns(httpMethod);

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

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

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

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

Nasza następna metoda pomocnicza pozwala testować ścieżkę:

```
...
private void TestRouteMatch(string url, string controller, string action,
    object routeProperties = null, string httpMethod = "GET") {

    // przygotowanie
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // działanie — przetwarzanie ścieżki
    RouteData result = routes.GetRouteData(CreateHttpContext(url, httpMethod));
    // asercje
    Assert.IsNotNull(result);
    Assert.IsTrue(TestIncomingRouteResult(result, controller,
        action, routeProperties));
}

...
```

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

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

```
...
private bool TestIncomingRouteResult(RouteData routeResult, string controller,
    string action, object propertySet = null) {

    Func<object, object, bool> valCompare = (v1, v2) => {
        return StringComparer.InvariantCultureIgnoreCase.Compare(v1, v2) == 0;
    };

    bool result = valCompare(routeResult.Values["controller"], controller)
        &&  valCompare(routeResult.Values["action"], action);

    if (propertySet != null) {
        PropertyInfo[] propInfo = propertySet.GetType().GetProperties();
        foreach (PropertyInfo pi in propInfo) {
            if (!(routeResult.Values.ContainsKey(pi.Name)
                    && valCompare(routeResult.Values[pi.Name],
                        pi.GetValue(propertySet, null)))) {

                result = false;
                break;
            }
        }
    }
    return result;
}

...
```

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

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

...
```

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

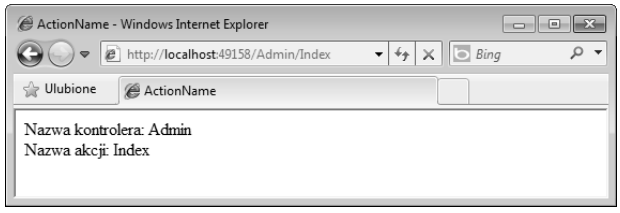
```
...
[TestMethod]
public void TestIncomingRoutes() {

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

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

...
```

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

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

Użycie prostej ścieżki

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



Rysunek 13.3. *Nawigacja za pomocą prostej ścieżki*

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

Definiowanie wartości domyślnych

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

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

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

Listing 13.9. *Określanie wartości domyślnej w ścieżce*

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

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

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

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

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

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

Listing 13.10. *Określanie domyślnych wartości dla kontrolera i akcji*

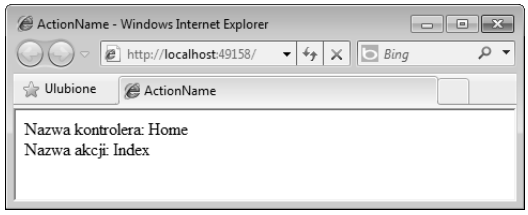
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

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

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

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

Tabela 13.2. *Dopasowanie adresów URL*

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

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



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

Testy jednostkowe — wartości domyślne

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

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

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

...
```

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

Użycie statycznych segmentów adresu URL

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

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

Możemy zrobić to przez użycie wzorca zamieszczonego na listingu 13.11.

Listing 13.11. *Wzorzec URL z segmentem statycznym*

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

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

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

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

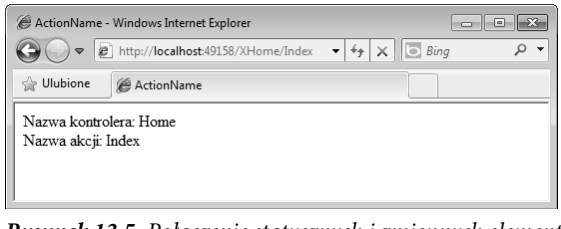
Wzorzec ten pasuje wyłącznie do adresów URL posiadających trzy segmenty, z których pierwszym *musi* być *Public*. Pozostałe dwa segmenty mogą zawierać dowolną wartość i będą używane dla zmiennych `controller` oraz `action`. Jeżeli dwa ostatnie segmenty zostaną pominięte, wtedy użyte będą wartości domyślne.

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

Listing 13.12. *Wzorzec URL z segmentem mieszanym*

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

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

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

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

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

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



Rysunek 13.5. *Połączenie statycznych i zmiennych elementów w pojedynczym segmencie*

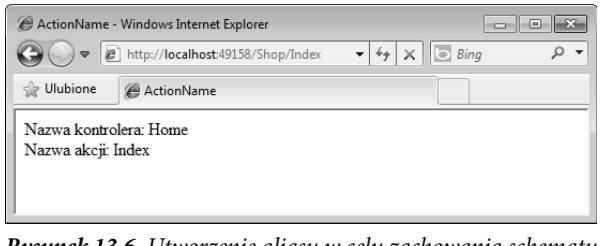
Możemy połączyć statyczne segmenty URL oraz wartości domyślne w celu utworzenia aliasów dla wybranych adresów URL. Jest to przydatne, jeżeli opublikowaliśmy schemat URL w postaci kontraktu dla użytkownika.

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

Listing 13.13. *Łączenie statycznych segmentów URL oraz wartości domyślnych*

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

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

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

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

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

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

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



Rysunek 13.6. *Utworzenie aliasu w celu zachowania schematu URL*

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

Listing 13.14. *Aliasy dla kontrolera i akcji*

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

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

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

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

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

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

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

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

Kolejność ścieżek

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

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

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

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

...
```

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

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

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

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

Test jednostkowy — testowanie segmentów statycznych

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

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

Definiowanie własnych zmiennych segmentów

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

Listing 13.15. *Definiowanie nowych zmiennych we wzorcu URL*

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

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

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

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

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

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

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

Listing 13.16. *Dostęp do własnej zmiennej segmentu w metodzie akcji*

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

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

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

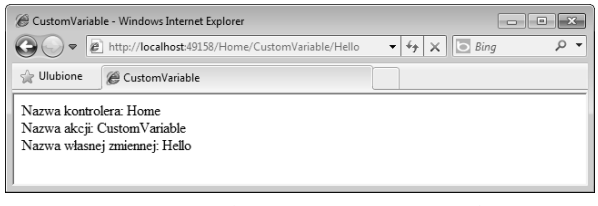
Metoda ta pozyskuje wartość zmiennej z wzorca ścieżki i przekazuje ją do widoku poprzez `ViewBag`. Kliknij prawym przyciskiem myszy nową metodę akcji w edytorze kodu i wybierz opcję *Dodaj widok...* z menu kontekstowego. Widokowi nadaj nazwę *CustomVariable* i kliknij przycisk *Dodaj*. Visual Studio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

Listing 13.17. *Wyświetlanie wartości zmiennej własnego segmentu*

```
@{
    Layout = null;
}
```

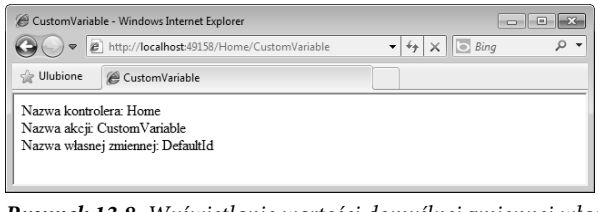
```
<!DOCTYPE html>

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

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



Rysunek 13.7. *Wyświetlanie wartości zmiennej własnego segmentu*

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



Rysunek 13.8. *Wyświetlanie wartości domyślnej zmiennej własnego segmentu*

Test jednostkowy — testowanie zmiennych własnych segmentów

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

```
...

[TestMethod]
public void TestIncomingRoutes() {

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

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

Użycie własnych zmiennych jako parametrów metod akcji

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

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

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

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

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

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

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

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

Definiowanie opcjonalnych segmentów URL

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

Listing 13.19. *Określanie opcjonalnego segmentu URL*

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

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

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

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

Tabela 13.3. *Dopasowanie adresów URL z opcjonalną zmienną segmentu*

Liczba segmentów	Przykładowy URL	Odwzorowany na
0	*witryna.pl*	`controller = Home` `action = Index`
1	*witryna.pl/Customer*	`controller = Customer` `action = Index`
2	*witryna.pl/Customer/List*	`controller = Customer` `action = List`

3	*witryna.pl/Customer/List/All*	`controller = Customer`
		`action = List`
		`id = All`

4	*witryna.pl/Customer/List/All/Delete*	Brak dopasowania — za dużo segmentów

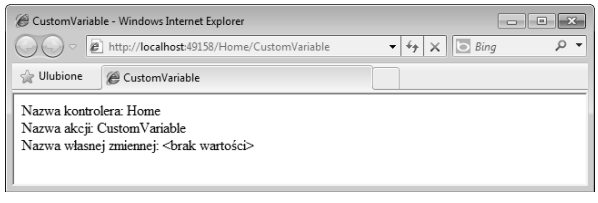
Jak można zauważyć w tabeli, zmienna `id` jest dodawana do zbioru zmiennych tylko w przypadku, gdy odpowiedni segment znajduje się w przychodzącym adresie URL. Funkcja ta jest przydatna, gdy musimy wiedzieć, czy użytkownik podał wartość. Jeżeli dla opcjonalnej zmiennej segmentu nie zostanie podana żadna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

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

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

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

namespace UrlsAndRoutes.Controllers {
    public class HomeController : Controller {

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

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

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



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

Użycie opcjonalnych segmentów URL w celu wymuszenia separacji zadań

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

Listing 13.21. *Definiowanie wartości domyślnej dla parametru metody akcji*

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

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

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

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

Definiowanie ścieżek o zmiennej długości

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

Listing 13.22. *Wyznaczanie zmiennej przechwytującej*

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

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

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

Testy jednostkowe — opcjonalne segmenty URL

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

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

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

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

Tabela 13.4. *Dopasowanie adresów URL ze zmienną przechwytującą segmentu*

Liczba segmentów	Przykładowy URL	Odwzorowany na
0	*witryna.pl*	`controller = Home` `action = Index`
1	*witryna.pl/Customer*	`controller = Customer` `action = Index`
2	*witryna.pl/Customer/List*	`controller = Customer` `action = List`

3	*witryna.pl/Customer/List/All*	`controller = Customer`
		`action = List`
		`id = All`

4	*witryna.pl/Customer/List/All/Delete*	`controller = Customer`
		`action = List`
		`id = All`
		`catchall = Delete`

5	*witryna.pl/Customer/List/All/Delete/Perm*	`controller = Customer`
		`action = List`
		`id = All`
		`catchall = Delete/Perm`

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

Test jednostkowy — testowanie zmiennych segmentów przechwytujących

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

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

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

Definiowanie priorytetów kontrolerów na podstawie przestrzeni nazw

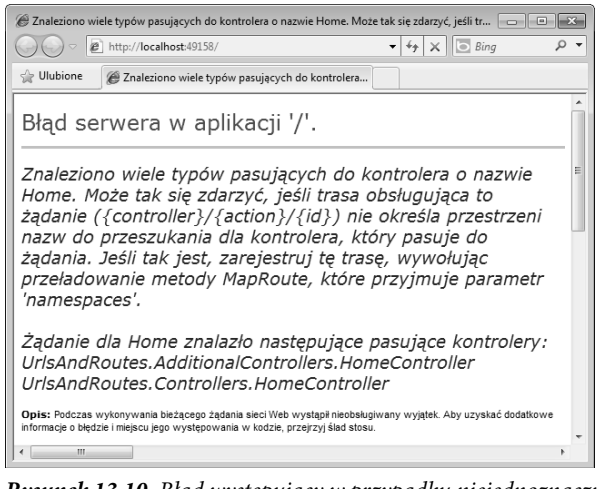
Gdy przychodzący adres URL zostanie dopasowany do ścieżki, platforma MVC odczytuje nazwę zmiennej `controller` i szuka klasy o odpowiedniej nazwie. Jeżeli na przykład wartością zmiennej `controller` jest *Home*, platforma MVC poszukuje klasy o nazwie `HomeController`. Jest to *niekwalifikowana* nazwa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

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

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

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

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

Po uruchomieniu aplikacji zobaczysz błąd pokazany na rysunku 13.10.



Rysunek 13.10. *Błąd występujący w przypadku niejednoznaczności klas kontrolerów*

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

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

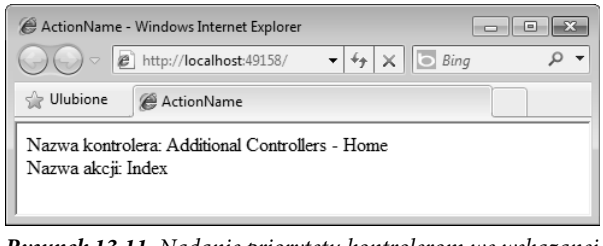
Aby rozwiązać ten problem, możemy określić przestrzenie nazw, które powinny mieć wyższy priorytet przy wyborze nazwy klasy kontrolera (listing 13.24).

Listing 13.24. *Określanie kolejności wykorzystania przestrzeni nazw*

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

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

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

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

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



Rysunek 13.11. *Nadanie priorytetu kontrolerom we wskazanej przestrzeni nazw*

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

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

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

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

Listing 13.25. *Użycie wielu ścieżek do sterowania przeszukiwaniem przestrzeni nazw*

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

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

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

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

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

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

Listing 13.26. *Wyłączenie domyślnego przeszukiwania przestrzeni nazw*

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

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

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

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

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

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

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

Ograniczenia ścieżek

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

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

Ograniczanie ścieżki z użyciem wyrażeń regularnych

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

Listing 13.27. *Użycie wyrażeń regularnych do ograniczania ścieżki*

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

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

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

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

Ograniczanie ścieżki do zbioru wartości

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

Listing 13.28. *Ograniczanie ścieżki do zbioru wartości zmiennej segmentu*

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

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

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

Ograniczanie ścieżek z użyciem metod HTTP

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

Listing 13.29. *Ograniczanie ścieżek bazujące na metodzie HTTP*

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

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

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

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

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

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

Testy jednostkowe — ograniczenia ścieżek

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

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

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


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

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

...
```

Definiowanie własnych ograniczeń

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

Listing 13.30. *Tworzenie własnego ograniczenia ścieżki*

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

namespace UrlsAndRoutes.Infrastructure {

    public class UserAgentConstraint : IRouteConstraint {
        private string requiredUserAgent;

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

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

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

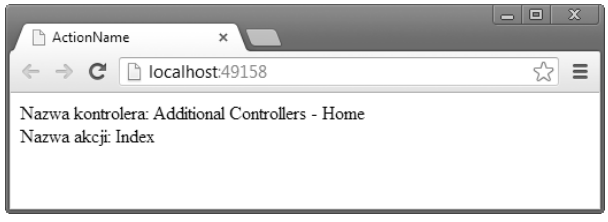
Interfejs `IRouteConstraint` definiuje metodę `Match`, której implementacja wskazuje systemowi routingu, czy ograniczenie jest spełnione. Parametry metody `Match` zapewniają dostęp do żądania od klienta, do kontrolowanej ścieżki, do zmiennych segmentów pobranych z adresu URL oraz do informacji, czy żądanie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

Listing 13.31. *Użycie niestandardowego ograniczenia w ścieżce*

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

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

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

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

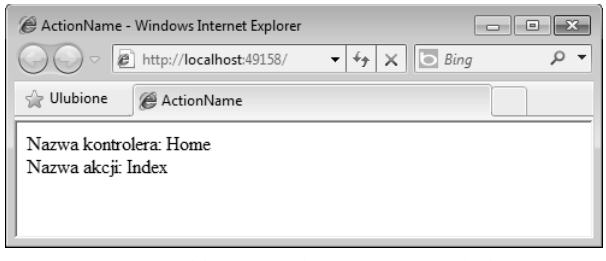
Na listingu mamy zdefiniowane ograniczenie ścieżki pozwalające na dopasowanie wyłącznie żądań wykonywanych z przeglądarek, których nagłówek `user-agent` zawiera `Chrome`. Jeżeli ścieżka zostanie dopasowana, wówczas żądanie będzie skierowane do metody akcji `Index` w kontrolerze `HomeController` zdefiniowanym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

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



Rysunek 13.12. *Aplikacja uruchomiona w przeglądarce Google Chrome*

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



Rysunek 13.13. *Aplikacja uruchomiona w przeglądarce Internet Explorer*

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

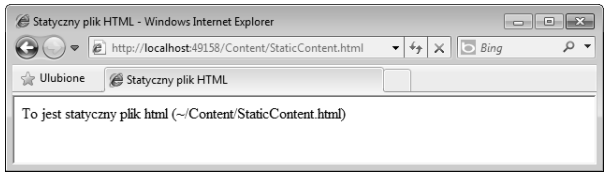
Routing żądań dla plików dyskowych

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

Listing 13.32. *Plik StaticContent.html*

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

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



Rysunek 13.14. *Żądanie przesłania pliku statycznego*

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

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

Listing 13.33. *Aktywowanie przetwarzania ścieżek przed kontrolą plików*

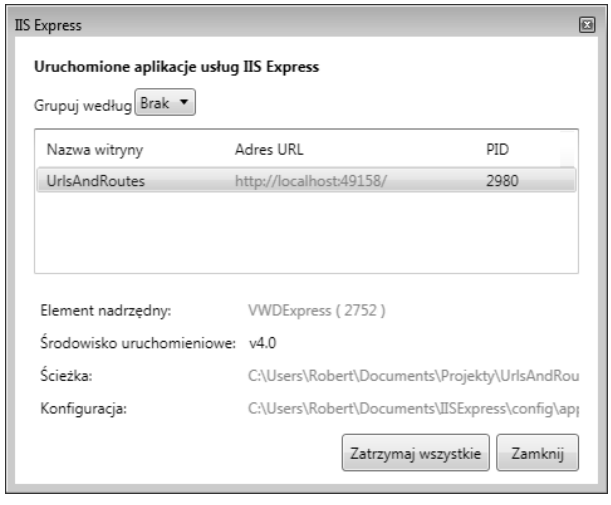
```
public static void RegisterRoutes(RouteCollection routes) {

    routes.RouteExistingFiles = true;

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

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

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

Konfiguracja serwera aplikacji

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

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

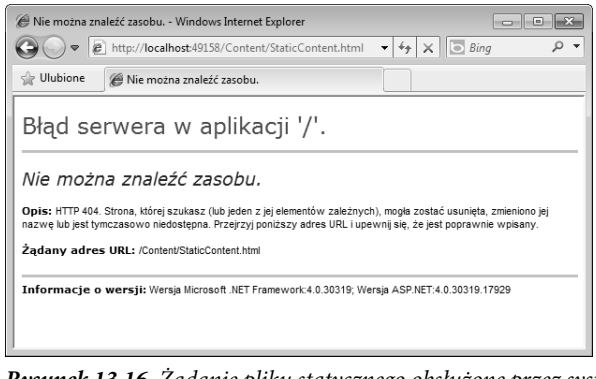


Rysunek 13.15. *Informacje konfiguracyjne serwera IIS Express*

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

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

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



Rysunek 13.16. *Żądanie pliku statycznego obsłużone przez system routingu*

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

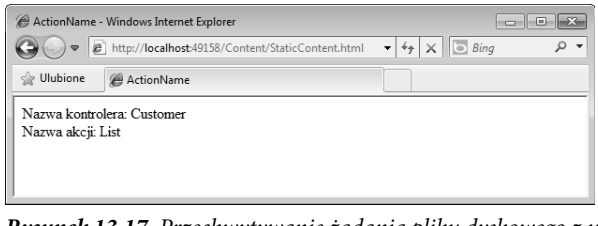
Definiowanie ścieżek dla plików na dysku

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

Listing 13.34. *Ścieżka, której wzorzec URL odpowiada plikowi na dysku*

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

    routes.RouteExistingFiles = true;

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

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

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

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



Rysunek 13.17. *Przechwytywanie żądania pliku dyskowego z użyciem ścieżki*

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

Pomijanie systemu routingu

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

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

Listing 13.35. *Użycie metody IgnoreRoute*

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

    routes.RouteExistingFiles = true;

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

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

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

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

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

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

Podsumowanie

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

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

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

Ż

LEKTURA OBOWIĄZKOWA KAŻDEGO PROGRAMISTY!

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

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

Sięgnij po książkę i sprawdź:

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

Apress

cena 99,00 zł

Nr katalogowy: 14573

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

helion.pl księgarnia internetowa

Sprawdź najnowsze promocje: http://helion.pl/promocje
Książki 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
http://helion.pl

ISBN 978-83-246-7299-8

9 788324 672998

Informatyka w najlepszym wydaniu