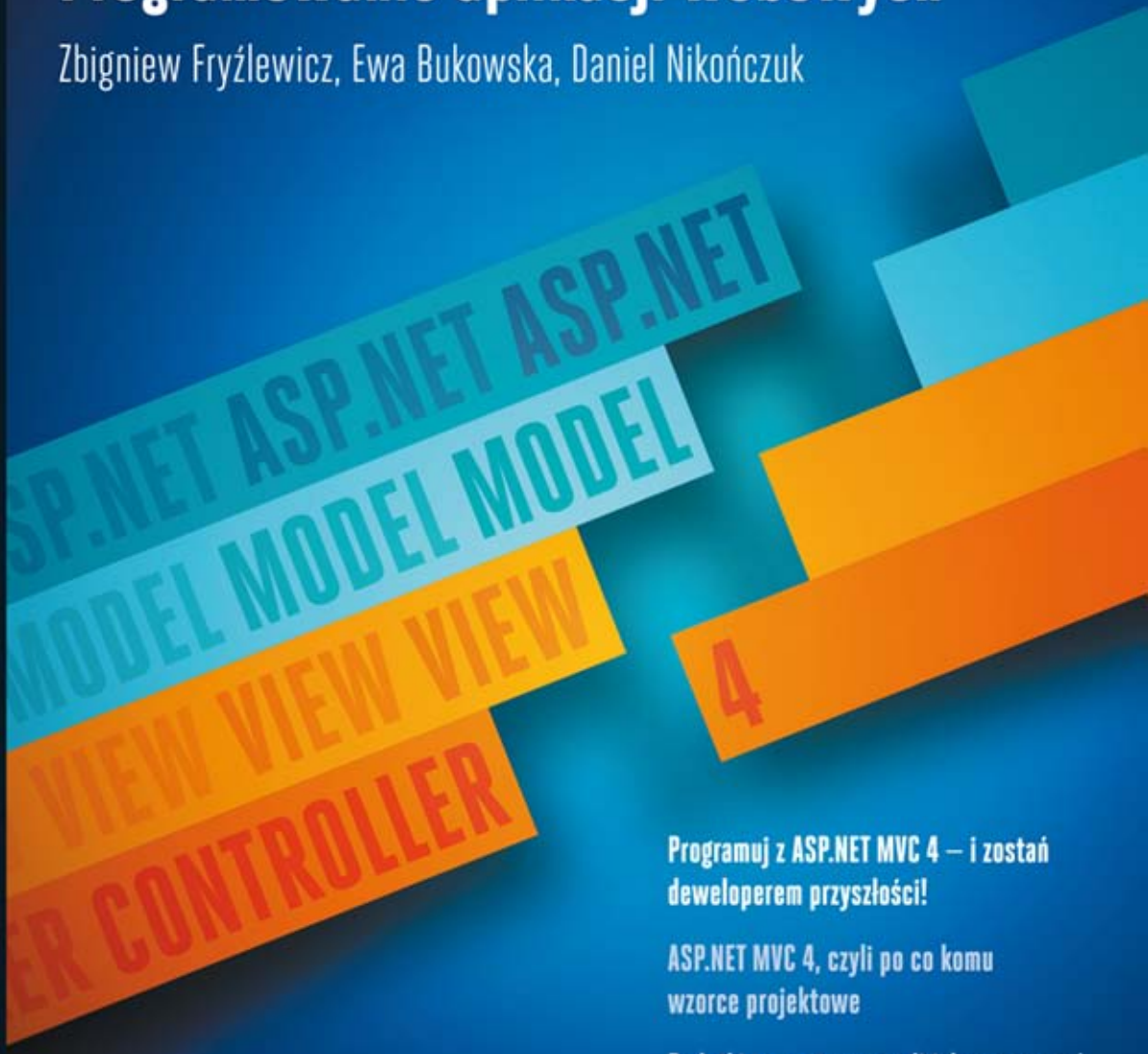
ASP.NET MVC 4

Programowanie aplikacji webowych

Zbigniew Fryźlewicz, Ewa Bukowska, Daniel Nikończuk



Programuj z ASP.NET MVC 4 – i zostań deweloperem przyszłości!

ASP.NET MVC 4, czyli po co komu wzorce projektowe

Dodatki zewnętrzne, czyli jak wspomagać aplikację innymi technologiami

Studia przypadku, czyli od czego zacząć i na czym skończyć w praktyce

Helion

Redaktor prowadzący: Ewelina Burska
Projekt okładki: ULABUKA
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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?aspm4w
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Materiały do książki można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/aspm4w.zip

ISBN: 978-83-246-6534-1

Printed in Poland.

Spis treści

Rozdział 3.
Kontrolery

Kontroler to zasadniczo pośrednik pomiędzy działaniami użytkownika a danymi zawartymi w modelu. Pośrednik, który obsługując żądania HTTP, może dokonywać zmian w modelu i uruchamiać odpowiedni widok z prezentacją danego fragmentu modelu.

Zgodnie z konwencją obowiązującą w ASP.NET MVC klasa, która jest kontrolerem, musi mieć nazwę zakończoną przyrostkiem `Controller` oraz implementować przynajmniej interfejs `IController` — listing 3.1.

Listing 3.1. *Interfejs IController*

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Interfejs `IController` jest bardzo prosty i zawiera tylko jedną metodę `Execute`. Wewnątrz tej metody można dostać się do obiektów takich jak `HttpContext`, `Request` czy `Response`. Taki sposób definiowania kontrolera jest rzadko użyteczny, bo w ten sposób pomija się wszystkie udogodnienia frameworka związane np. z bezpieczeństwem, wiązaniem modelu (ang. *model binding*) czy też wynikami akcji. Traci się także możliwość definiowania metod akcji — wszystkie żądania przychodzące do tego kontrolera są obsługiwane centralnie przez metodę `Execute`. Z tych powodów w praktyce nigdy nie implementuje się bezpośrednio interfejsu `IController`. Podejście praktyczne to dziedziczenie z jednej z dwóch klas bazowych — `ControllerBase` i `Controller`.

Framework ASP.NET MVC 4 posiada abstrakcyjną klasę, która jest klasą bazową dla wszystkich kontrolerów MVC. Klasa `ControllerBase` bezpośrednio implementuje interfejs `IController` oraz dostarcza dodatkowe funkcjonalności, np. odpowiada za tworzenie obiektu `ControllerContext`, dostarcza właściwość `ViewData` etc. `ControllerBase` nie dostarcza jednak innych wysoce pożądanych funkcjonalności, takich jak np. metody akcji. Te i inne funkcjonalności zawiera dopiero abstrakcyjna klasa `Controller`. Dostarcza ona trzy kluczowe funkcje: **akcje** (ang. *Action Methods*), **wyniki akcji** (ang. *Action Results*) oraz **filtry** (ang. *Filters*).

Akcje umożliwiają rozszerzenie zachowania kontrolera. Stosowanie pojedynczej akcji `Execute`, którą oferuje interfejs `IController`, jest mało wygodne. Przy zaimplementowaniu klasy kontrolera dziedziczącego po `Controller` każda akcja jest ściśle powiązana z adresem URL oraz parametrami przekazywanymi do niej, a wydobytymi z przychodzącego żądania.

Wyniki akcji umożliwiają zwracanie różnych bytów z poziomu wykonywanej akcji, np. wygenerowanie widoku, przekierowanie do innej akcji czy zwrócenie jakiegoś pliku. Taka jawna separacja wyników zwracanych przez akcję upraszcza przeprowadzanie testów jednostkowych.

W praktyce właśnie na podstawie klasy `Controller` tworzone są wszystkie klasy kontrolerów używanych w aplikacjach opartych na frameworku ASP.NET MVC 4.

Podczas projektowania aplikacji zachodzi zwykle konieczność ustalenia ziarnistości kontrolerów. W tym procesie wszystko zależy od projektanta. Jeżeli projektant przewidzi w aplikacji jeden kontroler, odpowiednio organizując w nim akcje, to aplikacja może działać prawidłowo. Wadą takiego podejścia jest jednak zwykle duży przyrost akcji oraz metod, które będą musiały obsługiwać żądania z szerokiej grupy klas biznesowych.

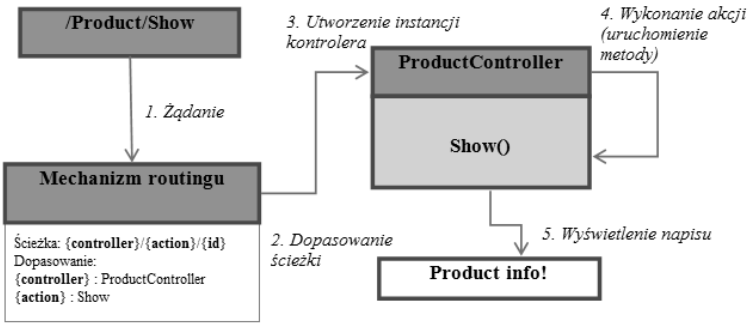
Bardziej wskazanym podejściem jest utworzenie kontrolerów w liczbie występujących w systemie klas biznesowych, które wymagają bezpośrednich operacji na danych. Wtedy w każdym z tych kontrolerów tworzy się zestaw akcji, które umożliwiają utworzenie, odczyt, modyfikację oraz usunięcie danych dla konkretnej klasy biznesowej.

Na listingu 3.2 przedstawiono fragment kodu, za pomocą którego utworzono kontroler `Product`.

Listing 3.2. *Przykładowy kontroler*

```
public class ProductController : Controller
{
   public void Show()
   {
      Response.Write("Product info!");
   }
}
```

W klasie kontrolera `Product` dodano metodę `Show`, której wynikiem wykonania jest wyświetlenie napisu *Product info*. Na rysunku 3.1 przedstawiono sposób, w jaki odbywa się wykonanie akcji `Show` w kontrolerze `Product`. W pierwszym kroku użytkownik aplikacji — poprzez interakcję z widokiem — przekazuje do aplikacji żądanie */Product/Show*. System dopasowuje otrzymane żądanie do wzorca ścieżki przy użyciu mechanizmu routingu. W trzecim kroku, kiedy ścieżka została dopasowana, system tworzy instancję kontrolera `ProductController` (odczytanego z żądania), po czym wykonuje akcję (poprzez uruchomienie metody) `Show`, która również odczytana została z żądania HTTP. Ostatnim krokiem jest wyświetlenie tekstu.



Rysunek 3.1. *Tworzenie kontrolera i uruchamianie akcji*

3.1. Akcje i parametry

Domyślnie w aplikacji ASP.NET MVC ścieżka URL rozbijana jest — przy użyciu mechanizmu routingu — na trzy główne elementy: `/{controller}/{action}/{id}`, gdzie `controller` to nazwa kontrolera, `action` oznacza nazwę akcji, a `id` to parametr dla akcji.

W kontrolerze może znajdować się wiele akcji (metod publicznych), z których każda może (ale nie musi) przyjmować parametry.

Listing 3.3 przedstawia zmodyfikowaną wersję kontrolera z poprzedniego przykładu (listing 3.2), która zawiera jeden parametr `id`.

Listing 3.3. *Akcja i parametr*

```
public class ProductController : Controller
{
   public void Show(string id)
      {
      Response.Write("Product" + id + "info!");
   }
}
```

Teraz gdy nastąpi żądanie: */Product/Show/TVsets*, w odpowiedzi wyświetlony zostanie napis: `Product TVsets info!`. Taki sam rezultat zostanie osiągnięty, jeśli żądanie będzie wyglądało tak: */Product/Show?id=TVsets*. Łatwo zauważyć, że pierwszy sposób jest wygodniejszy dla użytkownika. Takie przekazywanie parametru możliwe jest dzięki odpowiednio skonfigurowanemu mechanizmowi routingu, który sam zajmuje się przekazywaniem metodzie parametru odczytanego z adresu.

Oczywiście jest możliwość przekazywania większej liczby parametrów. Ilustracją jest zmodyfikowany kontroler z poprzedniego przykładu — listing 3.4.

Listing 3.4. *Akcja i większa liczba parametrów*

```
public class ProductController : Controller
{
```

```
    public void Show(String p1, String p2, int p3)
       {
       Response.Write("[id:" + p3 + "] name:" + p1 + " info " + p2);
       }
}
```

Jeśli teraz do aplikacji trafi żądanie: */Product/Show?p1=TVsets&p2=test&p3=1*, wówczas wyświetlony zostanie tekst: `[id:1] name: TVsets info test`. Aby adres mógł być bardziej czytelny dla użytkownika, należy zdefiniować odpowiednią ścieżkę w mechanizmie routingu — szczegóły przedstawia listing 3.5.

Listing 3.5. *Przykładowa ścieżka z kilkoma parametrami*

```
routes.MapRoute(
"productInfo", "Product/Show/{p1}/{p2},{p3}",
  new { Controller = "Product", action = "Show"}
)
```

Przy tak zdefiniowanej ścieżce jak na listingu 3.5 można użyć bardziej przyjaznej dla użytkownika wersji adresu URL, w postaci: */Product/Show/TVsets/test,1*.

W prezentowanych wcześniej przykładach zastosowano tymczasowo metodę `Response.Write`, która umożliwiała wyświetlenie tekstu. Użycie tej metody w znacznym stopniu naruszyło jednak zasadę rozdziału kompetencji ról poszczególnych warstw wzorca MVC. Kontrolery nie mogą zajmować się widokiem danych; dane prezentowane są w bytach zwanych widokami.

Aby zapewnić zgodność z konwencją, akcje w kontrolerach zwracają różnego rodzaju wyniki. W ASP.NET MVC 4 realizacja takiego podejścia następuje przy wykorzystaniu obiektów klasy `ActionResult` — listing 3.6.

Listing 3.6. *Klasa ActionResult*

```
public abstract class ActionResult
{
protected ActionResult()
   {
   }
public abstract void ExecuteResult(ControllerContext context);
}
```

Akcja w kontrolerze zwraca obiekt typu `ActionResult`, jednak nie służy on jako kontener dla danych. `ActionResult` jest abstrakcyjną klasą oferującą wspólny interfejs, który wykonuje pewne dalsze działania w imieniu akcji. Poprzez nadpisanie metody `ExecuteResult` pochodna klasy uzyskuje dostęp do wszelkich danych dostarczonych przez wykonanie akcji i wyzwala jakąś późniejszą akcję. Są to kolejne działania związane z wytworzeniem odpowiedzi dla przeglądarki.

Ponieważ klasa `ActionResult` jest abstrakcyjna, każda akcja zdefiniowana w kontrolerze może zwrócić jeden z predefiniowanych typów `ActionResult` — tabela 3.1.

Tabela 3.1. *Typy ActionResult*

Typ	Opis
`ContentResult`	Zwraca bezpośrednio podaną treść (niekoniecznie HTML) do przeglądarki. Metoda `ExecuteResult` serializuje przekazywaną treść.
`EmptyResult`	Nie zwraca niczego do przeglądarki. Metoda `ExecuteResult` nic nie robi.
`FileContentResult`	Zwraca zawartość pliku do przeglądarki. Zawartość pliku wyrażona jest jako tablica bajtów. Metoda `ExecuteResult` przekazuje zawartość do strumienia wyjściowego.
`FilePathResult`	Zwraca zawartość pliku do przeglądarki. Plik identyfikowany jest poprzez ścieżkę. Metoda `ExecuteResult` przekierowuje przeglądarkę bezpośrednio do pliku.
`FileStreamResult`	Zwraca zawartość pliku do przeglądarki. Zawartość pliku reprezentowana jest przez obiekt `Stream`. Metoda `ExecuteResult` kopiuje zawartość pliku z dostarczonego strumienia do strumienia wyjściowego.
`HttpNotFoundResult`	Zwraca kod HTTP 404 do przeglądarki. Podany kod identyfikuje żądanie jako niepowodzenie — żądany zasób nie został odnaleziony.
`HttpUnauthorizedResult`	Zwraca kod HTTP 401 do przeglądarki. Podany kod wskazuje nieuprawnione żądanie. W efekcie mechanizm uwierzytelniania prosi o zalogowanie.
`JavaScriptResult`	Zwraca kod JavaScript do przeglądarki. Metoda `ExecuteResult` zwraca skrypt i ustawia odpowiedni typ. Skrypt będzie wykonany po stronie klienta.
`JsonResult`	Serializuje obiekty .NET do formatu JSON i wysyła jako odpowiedź do przeglądarki. Metoda `ExecuteResult` określa typ obiektu, który ma zostać zwrócony, oraz wywołuje klasę `JavaScriptSerializer` w celu dokonania serializacji dostarczonego obiektu do formatu JSON.
`PartialViewResult`	Zwraca zawartość HTML do przeglądarki; zwracana zawartość reprezentuje fragment widoku strony.
`RedirectResult`	Zwraca kod HTTP 302 do przeglądarki w celu jej przekierowania na inny adres URL. Metoda `ExecuteResult` wywołuje po prostu `Response.Redirect`.
`RedirectToRouteResult`	Działa podobnie jak `RedirectResult`; w tym przypadku przekierowuje użytkownika na adres URL określony w parametrach routingu.
`ViewResult`	Zwraca zawartość HTML do przeglądarki; zawartość reprezentuje pełny widok strony.

W tabeli 3.1 można odnaleźć typy do obsługi kodów HTTP, możliwości zwracania plików, przekierowania i wsparcia dla JavaScriptu. Można też uzyskać asynchroniczne przekazywanie danych w formie serializowanej do formatu JSON.

3.2. Model binding

Model binding (wiązanie modelu) to mechanizm odwzorowywania danych przychodzących w żądaniu HTTP bezpośrednio na parametry akcji i obiekty *.NET* (model). Mechanizm jest wykorzystywany w tzw. modelu wejściowym danych, czyli takim, który opisuje dane przychodzące do aplikacji. Każde przychodzące żądanie jest rozpatrywane pod kątem kontrolera i akcji, której rezultat jest odpowiedzią na żądanie. Najpierw rozpoznawany jest kontroler, później akcja. Jeśli oba elementy zostaną odnalezione, to odszukiwane są dane oczekiwane przez akcję, a zdefiniowane jako parametry tej akcji.

Działanie mechanizmu *model binding* stanie się jasne, gdy najpierw przeanalizowany zostanie sposób pobierania danych z przychodzącego żądania HTTP. Na listingu 3.7 przedstawiono przykładowy model wraz z akcją `Create`. Akcja ma za zadanie utworzyć nową usługę i dodać ją do systemu.

Listing 3.7. *Pobieranie danych z żądania*

```
public class Service
{
    public int ID { get; set; }
    public String Name { get; set; }
    public String Content { get; set; }
    public DateTime PostedDate { get; set; }
}

public ActionResult Create()
{
    var service = new Service(){
    Name = Request["name"],
    Content = Request["content"],
    PostedDate = DateTime.Parse(Request["postedDate"])
    };
// ...
}
```

W przykładzie przedstawionym na listingu 3.7 akcja `Create` tworzy nowy obiekt `Service` i wypełnia go danymi bezpośrednio pobranymi z żądania. Ponieważ dane odczytywane z żądania są ciągiem znaków, to dla każdego pola z modelu `Service`, który nie jest typu `string`, należy pobraną wartość przekształcić na odpowiedni typ (jak w przypadku pola `PostedDate`).

Tworzenie akcji w taki sposób jest bardzo kłopotliwe i nieefektywne. W przypadku wykorzystania w akcji obiektu, który będzie miał wiele pól różnych typów, samo tworzenie obiektu będzie bardzo skomplikowane i przesłoni główny cel akcji, jakim jest — w powyższym przykładzie — dodanie nowej usługi do systemu.

W celu uniknięcia takich mało efektywnych metod można tworzyć akcje inaczej. Zamiast pobierać dane bezpośrednio z żądania można deklarować je jako parametry akcji — listing 3.8.

Listing 3.8. *Pobieranie danych z parametrów akcji — typy proste*

```
public ActionResult Create(string title, string content, DateTime postedDate)
{
    var service = new Service(){
    Title = title,
    Content = content
    PostedDate = postedDate
    };
// ...
}
```

W przypadku akcji `Create` z listingu 3.8 zadziała mechanizm *model binding*. Przed uruchomieniem akcji `Create` dwa ważne elementy, `ValueProviderDictionary` oraz `DefaultModelBinder`, wykonają pewną pracę. `ValueProviderDictionary` pobierze wartości z żądania HTTP i zapisze je jako ciągi znaków. Pobieranie odbywa się kolejno z następujących miejsc (na przykładzie parametru `string title`):

- `Request.Form["title"]`; jeśli element nie istnieje, to:
- `RouteData.Values["title"]`; jeśli element nie istnieje, to:
- `Request.Querystring["title"]`; jeśli element nie istnieje, to zwraca `null`.

Po pobraniu z żądania odpowiednich wartości przez `ValueProviderDictionary` klasa `DefaultModelBinder` bierze na siebie odpowiedzialność przekształcenia tych ciągów znaków na odpowiednie obiekty *.NET*.

Tak się dzieje, jeśli parametry oparte są o typy proste. Sytuacja się nieco komplikuje, gdy parametrem jest obiekt złożony (model), tak jak na listingu 3.9. Wtedy z pomocą przychodzi mechanizm **refleksji** (ang. *reflection*). Przy udziale refleksji wpisywane są — do wszystkich publicznych własności typu złożonego — wartości pobrane przez klasę `ValueProviderDictionary`. Warto wspomnieć również o tym, że wszystkie publiczne własności typu złożonego, które również są typem złożonym, w ten sam sposób otrzymają wartość.

Listing 3.9. *Pobieranie danych z parametrów akcji — typ złożony*

```
public ActionResult Create(Service service)
{
   // ...
}
```

Czasami zachodzi potrzeba, aby niektóre pola złożonego typu (modelu) wyłączyć z mechanizmu *model binding*. Do tego celu służy atrybut `Bind`. Można go wykorzystać na dwa sposoby: lokalnie w akcji — listing 3.10 lub globalnie w klasie modelu — listing 3.11. Przy wykorzystaniu atrybutu `Bind` można użyć dwóch parametrów `Include` lub `Exclude`, które odpowiednio włączają wymienione właściwości modelu do mechanizmu *model binding* lub wyłączają właściwości modelu z tego mechanizmu.

Listing 3.10. *Lokalny sposób użycia atrybutu Bind*

```
public ActionResult Create([Bind(Include = "Title,Content,PostedDate")] Service service)
{
    // Mechanizm model binding zadziała tylko dla właściwości Title, Content i PostedDate
    // ...
}

// lub
public ActionResult Create([Bind(Exclude = "ID, PostedDate")] Service service)
{
    // mechanizm model binding nie zadziała dla właściwości ID.
    // ...
}
```

Listing 3.11. *Globalny sposób użycia atrybutu Bind*

```
[Bind(Include = "Name,Content,PostedDate")]
public class Service
{
    public int ID { get; set; }
    public String Name { get; set; }
    public String Content { get; set; }
    public DateTime PostedDate { get; set; }
}

// lub
[Bind(Exclude = "ID")]
public class Service
{
    public int ID { get; set; }
    public String Name { get; set; }
    public String Content { get; set; }
    public DateTime PostedDate { get; set; }
}
```

Podane przykłady użycia mechanizmu *model binding* pokazują, że wykorzystanie tego narzędzia pozwala na tworzenie bardziej czytelnego kodu aplikacji, a także na jego znaczne skrócenie.

3.3. Filtry

Mechanizm filtrów w ASP.NET MVC 4 umożliwia wstrzyknięcie dodatkowej logiki do potoku przetwarzania żądania. Umożliwia to prosty i elegancki sposób realizacji przekrojowych problemów dotyczących logowania, autoryzacji czy buforowania. Na listingu 3.12 przedstawiono fragment definicji kontrolera zawierającego filtry.

Listing 3.12. *Przykładowy kontroler z filtrami*

```
[Authorize(Roles = "role")]
public class ExampleController : Controller
{
    [ShowMessage]
    [OutputCache(Duration = 60)]
```

```
    public ActionResult Index()
    {
        // ... ... ...
    }
}
```

Filtry zapisuje się w formie atrybutów (pochodzą z przestrzeni nazw `System.Attribute`) i mogą być dołączane do innych elementów kodu, w tym klas, metod, właściwości i pól. Celem ich użycia jest dostarczenie dodatkowych informacji do skompilowanego kodu. W języku C# elementy te umieszcza się w nawiasach kwadratowych, które dodatkowo mogą zawierać parametry, np. `[MyAttribute(SomeProperty = value)]`.

Przykład z listingu 3.12 zawiera kilka filtrów. Pierwszy w postaci `[Authorize(Roles = "role")]` oznacza, że użytkownik, który może korzystać z tego kontrolera, musi być autoryzowany i dodatkowo musi występować w roli `role`. Ten filtr obejmuje cały kontroler wraz ze wszystkimi akcjami.

Filtr zapisany w postaci `[OutputCache(Duration = 60)]` dotyczy tylko akcji `Index` i oznacza uruchomienie buforowania wyników tej akcji przez okres czasu wyrażony w sekundach i podany w parametrze `Duration`. Domyślnie dane przechowywane są w trzech miejscach: na serwerze WWW, na wszystkich serwerach proxy oraz w przeglądarce internetowej.

Framework ASP.NET MVC 4 obsługuje różne rodzaje filtrów:

- **Filtry zezwoleń** (`Authorization`) — filtr implementuje interfejs `IAuthorizationFilter` i służy do ograniczania dostępu do kontrolerów i akcji kontrolerów. Przykładami tego filtra są klasy `AuthorizeAttribute` oraz `RequireHttpsAttribute`. Filtr `Authorization` działa przed wszystkimi innymi filtrami.
- **Filtry akcji** (`Action`) — filtr implementuje interfejs `IActionFilter` i „opakowuje" działanie akcji kontrolera. Interfejs `IActionFilter` deklaruje dwie metody: `OnActionExecuting` i `OnActionExecuted`. `OnActionExecuting` uruchamiana jest przed wykonaniem akcji i może dostarczać akcji dodatkowych danych lub sprawdzać poprawność wprowadzanych danych. Metoda `OnActionExecuted` uruchamiana jest natomiast po wykonaniu akcji i może służyć do kontroli danych wynikowych lub anulowania wykonywania rezultatu akcji. Przykładem takiego filtru jest klasa `ActionFilterAttribute`.

  Filtry rezultatu (`Result`) — filtr implementuje interfejs `IResultFilter` i „opakowuje" obiekt wyniku działania akcji — ActionResult. Interfejs `IResultFilter` deklaruje dwie metody: `OnResultExecuting` i `OnResultExecuted`. Metoda `OnResultExecuting` uruchamiana jest przed zwróceniem rezultatu akcji (np. przed wyświetleniem widoku lub przekierowaniem do innej akcji), a `OnResultExecuted` działa po zwróceniu wyniku akcji. Obie metody mogą wprowadzać dodatkowe przetwarzanie rezultatu akcji, np. mogą modyfikować odpowiedź HTTP. Przykładem takiego filtru jest klasa `OutputCacheAttribute`.

- **Filtry wyjątku** (`Exception`) — filtr implementuje interfejs `IExceptionFilter` i wykonywany jest, gdy podczas przetwarzania akcji wystąpi nieobsługiwany wyjątek. Filtr ten może być wykorzystywany do rejestrowania lub wyświetlania strony błędu. Przykładem implementacji filtru wyjątku jest klasa `HandleErrorAttribute`.

W celu zdefiniowania filtru wystarczy utworzyć klasę, która będzie implementowała jeden z wymienionych wyżej interfejsów, a w metodach tego interfejsu zdefiniować dodatkowe działania. Przykładem takiej implementacji może być atrybut `CaptchaValidator` ↪`Attribute` wykorzystany w aplikacji *Portal Usług* (rozdział 8.).

Definicję `CaptchaValidatorAttribute` przedstawiono na listingu 3.13. Klasa atrybutu dziedziczy po klasie `ActionFilterAttribute`. Nazwa klasy musi być — według przyjętej we frameworku ASP.NET MVC konwencji — zakończona przyrostkiem *Attribute*. Filtr nadpisuje metodę `OnActionExecuting`, która uruchamiana jest przed wykonaniem akcji. Metoda sprawdza wartość przekazywaną w modelu (`model.Captcha`) i na jej podstawie dodaje do akcji nowy parametr (`captchaValid`), który można odczytać podczas wykonywania akcji — listing 3.14.

Listing 3.13. *Przykładowa implementacja filtra akcji*

```
public class CaptchaValidatorAttribute : ActionFilterAttribute
    {
        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
         RegisterModel model = filterContext.ActionParameters["model"] as RegisterModel;

            if (filterContext.HttpContext.Session["Captcha"] == null ||
            ↪filterContext.HttpContext.Session["Captcha"].ToString() != model.Captcha)
            {
                filterContext.ActionParameters["captchaValid"] = false;
            }
            else
            {
                filterContext.ActionParameters["captchaValid"] = true;
            }

            base.OnActionExecuting(filterContext);
        }
    }
```

Listing 3.14. *Wykorzystanie atrybutu CaptchaValidator*

```
      [CaptchaValidator]
        [AllowAnonymous]
        [HttpPost]
        public ActionResult Register(RegisterModel model, bool captchaValid)
        {
            if (captchaValid)
           {
    ...
    }
}
```

Podczas wytwarzania prostych aplikacji nie trzeba definiować własnych filtrów. Framework ASP.NET MVC 4 dostarcza cztery podstawowe filtry zaimplementowane w postaci atrybutów. Filtry te mogą być stosowane na poziomie akcji kontrolerów, kontrolerów lub na poziomie aplikacji:

- `AuthorizeAttribute` — ogranicza dostęp.
- `HandleErrorAttribute` — atrybut określający, jak obsługiwać wyjątek, który wystąpi podczas przetwarzania (filtr wymaga, aby w pliku konfiguracyjnym *Web.config* element `CustomErrors` miał wartość `true`).
- `OutputCacheAttribute` — zapewnia buforowanie wyjścia.
- `RequireHttpsAttribute` — wymusza ponowne przesłanie żądania przez HTTPS, gdy żądanie przyszło z HTTP.

3.4. Wbudowane atrybuty

Framework ASP.NET MVC udostępnia wbudowane atrybuty, takie jak `NonAction` ➥`Attribute`, `ActionNameAttribute` oraz `AcceptVerbsAttribute`.

3.4.1. NonAction

Wszystkie publiczne metody definiowane w kontrolerach to akcje, które użytkownik może wywołać, wpisując odpowiedni adres URL w przeglądarce. Czasami jednak zachodzi potrzeba zdefiniowania publicznej metody, która nie będzie dostępna dla użytkownika, a widzialność tej metody nie może zostać zmieniona. Aby ograniczyć bezpośredni dostęp do metody z przeglądarki, wystarczy opatrzyć ją atrybutem `NonActionAttribute` — listing 3.15.

Listing 3.15. *Zastosowanie atrybutu NonActionAttribute*

```
[NonAction]
public void Test()
{
    // logika metody
}
```

Podczas próby wywołania z przeglądarki metody opatrzonej atrybutem `NonAction` framework nie uruchomi metody, tylko w odpowiedzi zwróci kod błędu 404.

3.4.2. Atrybut ActionNameAttribute

Zastosowanie atrybutu `ActionNameAttribute` pozwala zmienić nazwę akcji. W przykładzie przedstawionym na listingu 3.16 zdefiniowane są dwie metody: `Delete` oraz `DeleteConfirmed`. Poprzez opatrzenie atrybutem `ActionName("Delete")` metody `Delete` ➥`Confirmed` zostanie ona wywołana przy żądaniu `POST` akcji `Delete`.

Listing 3.16. *Zastosowanie atrybutu ActionNameAttribute*

```
[HttpGet]
public ActionResult Delete(int id)
{
    var model = repository.Find(id);

    // Wyświetla widok potwierdzający usunięcie rekordu.
    return View(model);
}

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    repository.Delete(id);

    return RedirectToAction("Index");
}
```

3.4.3. Atrybut AcceptVerbsAttribute

Atrybut `AcceptVerbsAttribute` służy do związania akcji z metodą protokołu HTTP. Metody protokołu HTTP, które można związać tym atrybutem z akcjami, to `GET`, `POST`, `PUT` i `DELETE`. Przykładowo aby akcja `Test()` przekazywała wynik metodą `POST`, należy opatrzyć ją atrybutem jak na listingu 3.17.

Listing 3.17. *Zastosowanie atrybutu AcceptVerbsAttribute*

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Test()
{
   return ViewA()
}
```

Framework dostarcza również cztery skrócone formy tego atrybutu w postaci identyfikatorów: `HttpDeleteAttribute`, `HttpGetAttribute`, `HttpPostAttribute`, `HttpPutAttribute`.

3.5. Podsumowanie

We frameworku ASP.NET MVC kontroler to klasa, która w odpowiedzi na żądanie użytkownika zwraca odpowiedź. Kontroler może wydawać dowolny zbiór poleceń do modelu, a użytkownikowi (przeglądarce) zwrócić dowolny widok. Każdy kontroler zawiera jedną lub więcej akcji, które w istocie są specjalnymi metodami służącymi do przekazywania różnego rodzaju wyników. Predefiniowane typy `ActionResult` zostały zestawione w tabeli 3.1.

Z kontrolerami związany jest *model binding*, czyli mechanizm odwzorowania danych przychodzących w żądaniu HTTP bezpośrednio na parametry akcji i obiekty *.NET* (model). Każde przychodzące żądanie jest rozpatrywane pod kątem kontrolera i akcji, której

rezultat jest odpowiedzią na żądanie. Framework najpierw poszukuje odpowiedniego kontrolera, później akcji. Jeśli oba elementy zostaną odnalezione, to odszukiwane są dane oczekiwane przez akcję, a zdefiniowane jako parametry tej akcji.

Akcje są metodami publicznymi klasy kontrolera. Jeśli zachodzi konieczność wykluczenia publicznej metody kontrolera jako akcji, to należy ją opatrzyć atrybutem `NonAction ↪Attribute`, co powoduje skutki opisane w punkcie 3.4. To jest jeden z trzech wbudowanych atrybutów, nazywanych również filtrami akcji. Dwa pozostałe to `ActionName ↪Attribute` i `AcceptVerbsAttribute`. `ActionNameAttribute` służy do prostej zmiany nazwy akcji. Atrybut drugi służy do związania akcji z metodą protokołu HTTP.

Filtry mogą być stosowane do indywidualnych akcji lub do całego kontrolera. Służą do rozszerzania logiki, która jest stosowana w aplikacji podczas przetwarzania żądania, ale bez konieczności umieszczania tej logiki w sposobie działania. W efekcie zapis działań jest elegancki i czytelny.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W

Z

ASP.NET MVC to technologia, której poznanie pozwala na szybkie tworzenie eleganckich, prostych w utrzymaniu i wyrafinowanych w działaniu średnich i dużych aplikacji webowych. Współpraca z najważniejszymi standardami internetowymi, takimi jak HTML5, CSS, jQuery czy chmura Windows Azure, możliwość projektowania aplikacji dla urządzeń mobilnych i uproszczenie budowy modelu domenowego to tylko próbka najnowszych funkcjonalności, jakie zapewnia wersja 4 tego znakomitego frameworka.

Jeśli chcesz zostać rozchwytywanym specjalistą pracującym z użyciem ASP.NET MVC 4, nie znajdziesz lepszej książki. W części pierwszej opisano powstanie i warianty wzorca MVC, a także warstwy kontrolerów, widoków i modeli. Jej lektura pozwoli Ci poznać zasady działania frameworka, zorientować się w jego strukturze i opanować korzystanie z komponentów zewnętrznych, takich jak jQuery czy AJAX. Natomiast część druga zawiera cztery studia przypadku, czyli szczegółowe omówienie procesu budowania czterech różnych aplikacji korzystających z różnych technologii pomocniczych i osadzonych w różnych środowiskach. Dzięki temu uda Ci się sprawdzić, jak to wszystko działa w praktyce.

ASP.NET MVC 4 + ta książka = sukces w programowaniu!

helion.pl księgarnia internetowa

Nr katalogowy: 13405

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 0 601 339900

Helion

Sprawdź najnowsze promocje: http://helion.pl/promocje
Książki najchętniej czytane: http://helion.pl/bestsellery
Zamów informacje o nowościach: http://helion.pl/nowosci

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

Informatyka w najlepszym wydaniu

ISBN 978-83-246-6534-1

9 788324 665341

Cena: 59,00 zł