Projektowanie aplikacji ASP.NET

Wykład 06/15
ASP.NET MVC elementy zaawansowane

Wiktor Zychla 2024/2025

Spis treści

2 Własny routing z aktywacją handlera MVC 2
3 Filtry 7
4 Własny model binder 8
5 Własny atrybut walidacyjny 10
6 Własny HTML helper 11
7 WebGrid 13
8 Programowanie asynchroniczne 16
9 Własna fabryka kontrolerów 17
10 Widoki częściowe (Partial Views) i sekcje 19
11 Testy jednostkowe 21

2 Własny routing z aktywacją handlera MVC

Temat podsystemu MVC wprowadzaliśmy demonstrując zaawansowany mechanizm routingu – w przykładzie z wykładu pojawił się router parsujący ścieżki postaci /site/subsite/subsubsite/page.html. Omawiając tamten przykład obiecaliśmy sobie jeszcze do niego wrócić i pokazać jak przekierować taki routing nie do własnego handlera, ale do handlera MVC.

Przypomnijmy definicję routera:

```
public class CMSCustomRoute : Route
{
    public const string DEFAULTPAGEEXTENSION = ".html";

    public const string CMS = "CMS";

    public const string SITENAME = "siteName";
    public const string PAGENAME = "pageName";

    public CMSCustomRoute(
        RouteValueDictionary defaults,
        IRouteHandler routeHandler )
        : base( string.Empty, defaults, routeHandler )
    {
        this.Defaults = defaults;
        this.RouteHandler = routeHandler;
    }

    /// <summary>
    /// Metoda która dostaje Url i ma zwrócić segmenty routy
    /// </summary>
    public override RouteData GetRouteData( HttpContextBase httpContext )
    {
        RouteData routeData = new RouteData( this, this.RouteHandler );

        string virtualPath = httpContext.Request.AppRelativeCurrentExecutionFile
Path.Substring( 2 ) + ( httpContext.Request.PathInfo ?? string.Empty );

        string[] segments = virtualPath.ToLower().Split( new[] { '/' }, StringSp
litOptions.RemoveEmptyEntries );

        if ( segments.Length >= 1 && string.Equals( segments.First(), CMS, Strin
gComparison.InvariantCultureIgnoreCase ) )
        {
            if ( segments.Last().IndexOf( DEFAULTPAGEEXTENSION ) > 0 )
            {
                routeData.Values[SITENAME] = string.Join( "/", segments.Skip( 1
).Take( segments.Length - 2 ).ToArray() );
                routeData.Values[PAGENAME] = segments.Last().Substring( 0, segme
nts.Last().IndexOf( "." ) );
            }
            else if ( segments.Last().IndexOf( "." ) < 0 )
            {
                routeData.Values[SITENAME] = string.Join( "/", segments.Skip( 1
).ToArray() );
                routeData.Values[PAGENAME] = "index.html";
            }
            else
```

```
            {
                return null;
            }

            // add remaining default values
            foreach ( KeyValuePair<string, object> def in this.Defaults )
            {
                if ( !routeData.Values.ContainsKey( def.Key ) )
                {
                    routeData.Values.Add( def.Key, def.Value );
                }
            }

            return routeData;
        }
        else
            return null;
    }

    /// <summary>
    /// Metoda która dostaje segmenty routy a ma zwrócić URL
    /// </summary>
    /// <remarks>
    /// Wykorzystuje ją np. UrlHelper
    /// </remarks>
    public override VirtualPathData GetVirtualPath(
        RequestContext requestContext,
        RouteValueDictionary values )
    {
        List<string> baseSegments = new List<string>();
        List<string> queryString = new List<string>();

        if ( values[SITENAME] is string )
            baseSegments.Add( (string)values[SITENAME] );

        if ( values[PAGENAME] is string )
        {
            string pageName = (string)values[PAGENAME];
            if ( !string.IsNullOrEmpty( pageName ) &&
                 !pageName.EndsWith( DEFAULTPAGEEXTENSION ) )
                pageName += DEFAULTPAGEEXTENSION;

            baseSegments.Add( pageName );
        }

        string uri = string.Join( "/", baseSegments.Where( s => !string.IsNullOr
Empty( s ) ) );

        return new VirtualPathData( this, uri );
    }
}
```

Zobaczmy jak rejestrować taki router:

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

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

        // routy postaci CMS/site/subsite/page
        routes.Add(
               "customroute",
               new CMSCustomRoute(
                   new RouteValueDictionary(new { controller = "Page", action =
"Render" }),
                   new MvcRouteHandler())
                   );

        // domyślny routing MVC
        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParam
eter.Optional }
        );
    }
}
```

Zwróćmy uwagę na to że:

- W routerze zdecydowano się na pewną pomocniczą konwencję, w której ścieżka do strony CMS ma zawsze **/CMS** jako pierwszy segment – czyli router rozpoznaje tak naprawdę ścieżki postaci **/CMS/site/subsite/sububsite/page.html**. Ta konwencja, w połączeniu z faktem że żaden kontroler w aplikacji **nie będzie** się nazywał **CMSController**, spowoduje że nie będzie **niedjednoznaczności** przy odwołaniu do ścieżek – jeżeli ścieżka rozpoczyna się od /CMS to będzie rozpoznana przez router CMS, jeżeli ścieżka rozpoczyna się od czegokolwiek innego – to przez jakiś inny router (na przykład router MVC)
- Router specyficznych ścieżek skonfigurowany jest najpierw, router ogólnej ścieżki MVC (**/{controller}/{action}/{id}**) skonfigurowany jest później – wynika to z faktu że routy są dopasowywane w kolejności i jest to dobra, dodatkowa praktyka rozstrzygania niejednoznaczności – zgodnie z tą praktyką definicję routingu rozpoczyna się od ścieżek najbardziej specyficznych, a kończy na najogólniejszych
- Routing dla routera CMS jest określony tak że jako domyślne wartości dla handlera MVC pozostawia po sobie wskazanie kontrolera (**Page**) i akcji (**Render**) – dzięki temu do obsługi strony CMS zostanie uruchomiona akcja **Render** z kontrolera **Page** ale akcja ta będzie miała dostęp do pozostałych informacji pozostawionych w RouteData przez router (czyli do **SiteName** i **PageName**)

```
public class PageController : Controller
{
    public ActionResult Render()
    {
        var routeData = this.Request.RequestContext.RouteData.Values;

        string site = routeData[CMSCustomRoute.SITENAME] as string;
        string page = routeData[CMSCustomRoute.PAGENAME] as string;

        // odczyt z magazynu danych

        // renderowanie
        var model = new PageRenderModel()
        {
```

```
            Site = site,
            Page = page
        };

        return View(model);
    }
}

public class PageRenderModel
{
    public string Site { get; set; }
    public string Page { get; set; }
}
```

Żeby żądanie do zasobu statycznego (.html) przeszło na serwerze przez potok ASP.NET, należy wymusić to odpowiednim ustawieniem w **web.config**, w przeciwnym razie serwer zwróci od razu wynik 404 (dlaczego?)

```
        <system.webServer>
                <modules runAllManagedModulesForAllRequests="true" />
        </system.webServer>
```

Na platformie ASP.NET Core własny router to tzw. *transformator*

```
public class CMSCustomRouteTransformer : DynamicRouteValueTransformer
{
    public const string DEFAULTPAGEEXTENSION = ".html";

    public const string CMS = "CMS";

    public const string SITENAME = "siteName";
    public const string PAGENAME = "pageName";

    public override async ValueTask<RouteValueDictionary> TransformAsync( H
ttpContext httpContext, RouteValueDictionary values )
    {
        if ( !values.ContainsKey( "sitepage" ) ) return values;
        var virtualPath = values["sitepage"].ToString();

        if ( string.IsNullOrEmpty( virtualPath ) ) return values;

        string[] segments = virtualPath.ToLower().Split( new[] { '/' }, Str
ingSplitOptions.RemoveEmptyEntries );

        if ( segments.Length >= 0 )
        {
            if ( segments.Last().IndexOf( DEFAULTPAGEEXTENSION ) > 0 )
            {
                values["controller"] = "Page";
                values["action"]     = "Render";
                values[SITENAME]     = string.Join( "/", segments.Take( seg
ments.Length - 1 ).ToArray() );
```

```
                values[PAGENAME]     = segments.Last().Substring( 0, segmen
ts.Last().IndexOf( "." ) );
            }
            else if ( segments.Last().IndexOf( "." ) < 0 )
            {
                values["controller"] = "Page";
                values["action"]     = "Render";
                values[SITENAME]     = string.Join( "/", segments.ToArray()
 );
                values[PAGENAME]     = "index.html";
            }
        }

        return values;
    }
}
```

Jego rejestracja w potoku wymaga zarówno zarejestrowania go jako usługi (np. signeton) jak i wskazania jako parsera ścieżek (**MapDynamicControllerRoute**):

```
using WebApplication1.Controllers;

var builder  = WebApplication.CreateBuilder(args);
var services = builder.Services;

services.AddControllersWithViews();
services.AddSingleton<CMSCustomRouteTransformer>();

var app = builder.Build();

app.UseRouting();
app.UseEndpoints( endpoints =>
{
    endpoints.MapDynamicControllerRoute<CMSCustomRouteTransformer>( "CMS/{*
*sitepage}" );
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}" );
} );

app.Run();
```

3 Filtry

Mechanizm filtrów pozwala rozszerzyć potok przetwarzania o własną logikę w obszarach:

- Filtry akcji
- Filtry autentykacji
- Filtry wyniku
- Filtry wyjątków

Przykład filtra akcji:

```
public class CustomActionFilter : ActionFilterAttribute, IActionFilter
 {
     #region IActionFilter Members

     void IActionFilter.OnActionExecuted(ActionExecutedContext filterContext)
     {
         //throw new NotImplementedException();
     }

     void IActionFilter.OnActionExecuting(ActionExecutingContext filterContext)
     {
         string controllerName = filterContext.ActionDescriptor.ControllerDescri
ptor.ControllerName;
         string actionName     = filterContext.ActionDescriptor.ActionName;

         // np. logowanie
         // lub wręcz nadpisanie odpowiedzi
         // filterContext.Result = ...
     }

     #endregion
 }
```

oraz jego użycie

```
[CustomActionFilter]
public ActionResult Index()
{
    return Content("foo");
}
```

4 Własny model binder

Własny binder modelu ma zastosowanie w przypadkach nietypowych konstrukcji formularzy, które miałyby się niestandardowo mapować na model. Przykładowy binder zadziała tak że wartość w polu tekstowym wpisana z przecinkami jako „a,b,c” zostanie zbindowana nie do zmiennej typu **string,** standardowo, tylko do zmiennej typu **string[]**, gdzie poszczególne elementy tablicy będą kolejnymi wartościami z napisu: a b i c.

```
public class CustomBinder : IModelBinder
 {
     #region IModelBinder Members

     public object BindModel(
         ControllerContext controllerContext,
         ModelBindingContext bindingContext)
     {
         // nazwa zmiennej modelu (parametr metody akcji)
         string name = bindingContext.ModelName;

         var result = bindingContext.ValueProvider.GetValue(name);
         if (result != null)
         {
             string value = result.AttemptedValue;

             return value.Split(new[] { ',' });
         }
         else
             return Enumerable.Empty<string>();
     }

     #endregion
 }
```

Bindowanie musi być wskazane, na przykład atrybutem

```
[HttpPost]
public ActionResult Index(
    [ModelBinder(typeof(CustomBinder))] IEnumerable<string> foo
    )
{
    IndexModel model = new IndexModel();
    model.Foo = string.Join(",", foo);
    return View(model);
}
```

W przykładzie tym jest standardowy model i widok

```
public class IndexModel
{
    public string Foo { get; set; }
}
```

```
@model WebApplication2.Models.IndexModel
@{
```

```
    ViewBag.Title = "Index";
}

@using (var form = Html.BeginForm())
{
    @Html.TextBoxFor( m => m.Foo )
    <button>Zapisz</button>
}
```

5 Własny atrybut walidacyjny

Własny atrybut walidacyjny pozwala na rozszerzenie dostarczonego zestawu atrybutów walidacyjnych. Poniższy, przykładowy, pozwala na wskazanie oczekiwanej wartości pola tekstowego:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class OurCustomValidationAttribute : ValidationAttribute
{
    private string _value { get; set; }
    public OurCustomValidationAttribute(string value)
    {
        this._value = value;
    }

    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        object property = validationContext.ObjectInstance.GetType().InvokeMembe
r(validationContext.DisplayName, System.Reflection.BindingFlags.GetProperty | Sy
stem.Reflection.BindingFlags.Public | System.Reflection.BindingFlags.Instance, n
ull, validationContext.ObjectInstance, null);

        if (property is string && ((string)property) == _value)
            return ValidationResult.Success;
        else
            return new ValidationResult("wrong value");
    }
}
```

Model udekorowany walidatorem

```
public class IndexModel
{
    [OurCustomValidation("foo")]
    public string Foo { get; set; }
}
```

6 Własny HTML helper

Ten mechanizm rozszerzający pozwala na tworzenie własnych helperów, w tym rozbudowanych jak WebGrid

```
public static class CustomHtmlHelper
{
    /// <summary>
    /// Usage Html.CustomTextBox( "foo" )
    /// </summary>
    /// <param name="htmlHelper"></param>
    /// <param name="name"></param>
    /// <returns></returns>
    public static HtmlString CustomTextBox(this HtmlHelper htmlHelper, string na
me, object value)
    {
        TagBuilder tb = new TagBuilder("input");

        tb.MergeAttribute("type", "text");
        tb.MergeAttribute("name", name);
        if (value != null)
        {
            tb.MergeAttribute("value", value.ToString());
        }

        return new HtmlString(tb.ToString());
    }

    public static HtmlString CustomTextBoxFor<TModel, TProperty>(
        this HtmlHelper<TModel> htmlHelper,
        Expression<Func<TModel, TProperty>> Property)
    {
        TagBuilder tb = new TagBuilder("input");

        tb.MergeAttribute("type", "text");

        string name = ExpressionHelper.GetExpressionText(Property);

        tb.MergeAttribute("name", name);

        object value = ModelMetadata.FromLambdaExpression(Property, htmlHelper.V
iewData).Model;

        if (value != null)
        {
            tb.MergeAttribute("value", value.ToString());
        }

        return new HtmlString(tb.ToString());
    }
}
```

Użycie:

```
@model WebApplication2.Models.IndexModel
@using WebApplication2.Models
@{
```

```
    ViewBag.Title = "Index";
}

@using (var form = Html.BeginForm())
{
    @Html.CustomTextBoxFor( m => m.Foo )
    <button>Zapisz</button>
}
```

7 WebGrid

W implementacji stronicowania w MVC przydaje się pomocniczy model, opisujący stronicowany zbiór danych.

```
public class PagedEnumerable<T>
{
    public int CurrentPage { get; set; }
    public int PageSize { get; set; }
    public int TotalCount { get; set; }

    public IEnumerable<T> Items { get; set; }
}
```

Taki pomocniczy model może być następnie składową modelu głównego:

```
public class IndexModel
{
    public PagedEnumerable<User> Users { get; set; }
}
```

Helper **WebGrid** funkcjonalnie jest tylko namiastką GridView/ListView z **WebForms,** brakuje tu zwłaszcza rozbudowanych mechanizmów edycji ale obsługuje poprawnie stronicowanie i sortowanie.

Niestety, nie dzieje się to automatycznie. Kontroler musi wspierać odczytywanie porządku sortowania oraz numeru strony, jak również przygotowywać dane w modelu dla grida. Grid potrafi poprawnie wyrenderować stronę danych oraz pager. Inaczej niż w przypadku WebForms, tu kolumna sortowania i porządek sortowania są przez grid obsługiwane na dwóch rozłącznych parametrach:

```
public ActionResult Index(
    int page = 1,
    string sort = "GivenName", string sortdir = "ASC")
{
    var model     = new IndexModel();
    var dataLayer = new DataLayer();

    model.Users = new PagedEnumerable<DataLibrary.User>()
    {
        Items      = dataLayer.GetUsers(string.Format( "{0} {1}", sort, sortdir
), (page - 1) * 10, 10),
        TotalCount = dataLayer.TotalUsers()
    };

    return View(model);
}
```

Warstwa danych może tu być całkowicie przykładowa:

```
public class DataLayer
 {
     public IEnumerable<User> GetUsers( string OrderBy, int StartRow, int RowCou
nt )
     {
         IEnumerable<User> model = StaticModel.Users;

         model = model.AsQueryable().OrderBy(OrderBy).ToList();
```

```
        return model.Skip(StartRow).Take(RowCount);
    }

    public int TotalUsers()
    {
        return StaticModel.Users.Count();
    }
}

public class StaticModel
{
    static StaticModel()
    {
        Users = new List<User>();
        Enumerable.Range(1, 100).ToList().ForEach(i =>
        {
            Users.Add(new User()
            {
                ID        = i,
                GivenName = "GivenName " + i.ToString(),
                Surname   = "Surname " + (100-i).ToString()
            });
        });
    }

    public static List<User> Users { get; private set; }
}

public class User
{
    public int ID { get; set; }
    public string GivenName { get; set; }
    public string Surname { get; set; }
}
```

Renderowanie WebGrida polega na wskazaniu mu źródła danych oraz na użyciu metod renderujących tabelę oraz pager. Na uwagę zasługuje możliwość użycia funkcji typu lambda do tworzenia zawartości kolumn:

```
@{
    var grid = new WebGrid(canPage: true, rowsPerPage: 10);
    grid.Bind(source: Model.Users.Items,
              rowCount: Model.Users.TotalCount, autoSortAndPage: false);
}

<div>
    @grid.Table(
            columns: grid.Columns(
                grid.Column("GivenName", "Given name"),
                grid.Column("Surname", "Surname"),
                grid.Column(format:
                    item =>
                        Html.ActionLink("Edycja", "Edit",
                        new { id = item.ID }), canSort: false)
            )
        )
```

```
</div>
<div>
    @grid.Pager(WebGridPagerModes.All, "<<", "<", ">", ">>")
</div>
```

8 Programowanie asynchroniczne

Serwer aplikacji przydziela przychodzącym żądaniom wątki według ściśle określonych reguł. Jeśli żądania blokują wykonanie na oczekiwaniu na zewnętrzne I/O (pliki, baza danych), to możemy myśleć o nieoptymalnym wykorzystaniu zasobów serwera.

Dlatego MVC wprowadza możliwość tworzenia kodu asynchronicznego na serwerze, co w praktyce oznacza, że z punktu widzenia klienta (protokół http) żądanie nadal jest synchroniczne, ale na serwerze wątek roboczy można było uwolnić dla innych obliczeń:

```
[HttpGet]
public async Task<ActionResult> Index()
{
    IndexModel model = new IndexModel();
    await Task.Delay(1000);
    return View(model);
}
```

9 Własna fabryka kontrolerów

MVC pozwala na przejęcie kontroli nad fabryką kontrolerów. Dzięki temu możliwe jest np. zaimplementowanie wstrzykiwania zależności do klas kontrolerów przez kontenener IoC:

```
public class CustomControllerFactory :
    DefaultControllerFactory,
    IControllerFactory
{
    private IUnityContainer _container { get; set; }

    public CustomControllerFactory( IUnityContainer container )
    {
        this._container = container;
    }

    public override IController CreateController(RequestContext requestContext,
string controllerName)
    {
        var controllerType = this.GetControllerType(requestContext, controllerNa
me);

        if (controllerType != null)
        {
            IController controller = (IController)_container.Resolve(controllerT
ype);

            if (controller != null)
                return controller;
        }

        throw new HttpException(404, string.Format("Nie odnaleziono zasobu dla ż
ądania '{0}'", new object[]
        {
            requestContext.HttpContext.Request.Path
        }));
    }
}
```

Fabrykę należy wskazać i skonfigurować w potoku:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        IUnityContainer container = new UnityContainer();
        container.RegisterType<IService, ServiceImpl>();

        ControllerBuilder.Current.SetControllerFactory(new CustomControllerFacto
ry(container));
    }
}

public interface IService
{
```

```
    string DoWork();
}

public class ServiceImpl : IService
{
    public string DoWork()
    {
        return string.Format("serviceimpl: {0}", DateTime.Now);
    }
}
```

dzięki czemu możliwe jest wstrzykiwanie zależności do kontrolera, np. przez konstruktor (czyli tak jak wspiera to wykorzystany kontener: tu Unity):

```
public class HomeController : Controller
{
    private IService _service;
    public HomeController( IService service )
    {
        this._service = service;
    }
```

Taka architektura wspiera właściwą modularność aplikacji i jest chętnie wykorzystywania w praktyce. Chciałbym zwrócić uwagę na ten przykład – ponieważ wstrzykiwanie zależności do kontrolerów jako mechanizm wbudowany we framework będzie jedną z cech charakterystycznych **ASP.NET Core MVC**.

10 Widoki częściowe (Partial Views) i sekcje

Widok może zlecić renderowanie części zawartości do innego widoku, wywołanie jednego widoku z innego widoku wymaga użycia helpera **@Html.Partial**. Ta technika bywa przydatna w sytuacji gdy widok jest na tyle złożony, że warto wydzielić z niego fragment lub kiedy jakiś fragment widoku może być wielokrotnie użyty. Widok częściowy może mieć opcjonalnie przekazany model lub jego fragment.

```
@model WebApplication1.Models.PartialViewDemo.PartialViewFromViewModel
@{
    ViewBag.Title = "PartialViewFromView";
}

<h2>PartialViewFromView</h2>

Zawartość strony PartialViewFromLayout. Strona sama sięga do modelu i wywołuje w
idok częściowy, przekazując do niego fragment modelu.

@Html.Partial("PartialViewFromViewPartial", Model.Model2)
```

Pewnym problemem który należy umieć rozwiązać jest sytuacja, w której widok częściowy jest renderowany nie tyle z konkretnego widoku (np. **Index**), co z widoku szablonowego (**Layout**).

Czyli zamiast

Layout -> (zawiera) -> **Index** -> (renderuje widok częściowy) -> **Partial**

Byłoby

Layout -> (renderuje widok częściowy) -> **Partial**

też -> (zawiera) -> **Index**

Jak w takiej sytuacji przekazać model z widoku do widoku częściowego?

Jedna z możliwości w tej sytuacji to przygotowanie interfejsu, implementacja tego interfejsu na modelu, który otrzymuje strona oraz zadeklarowanie że model widoku częściowego implementuje ten interfejs.

```
@* Przykładowy layout który używa widoku częściowego (UserInfoPartial) *@
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title</title>
</head>
<body>
    @Html.Partial("UserInfoPartial")
    @RenderBody()
</body>
</html>
```

```
@* Widok częściowy (Partial View) - nagłówek.
   Jest wywoływany bezpośrednio z widoku Layoutu, dlatego nie ma jak otrzymać in
stancji modelu z konkretnego widoku.
```

```
   Zamiast tego deklaruje typ modelu jako interfejs (IUserInfo), który następnie
 implementują modele wszystkich widoków
   renderowanych z tego samego layoutu
*@
@model WebApplication1.Models.IUserInfo
@{}

<header>
    Zalogowany użytkownik: @Model.UserName
</header>
```

Sekcje to mechanizm w którym fragment strony może być „zadeklarowany” w widoku layoutu, a konkretna strona może (lub nie) dostarczyć jego zawartość. Jest to przydatne na przykład w sytuacji gdy konkretne strony dodają (lub nie) swoje własne referencje do zewnętrznych skryptów JS – w takiej sytuacji to konkretna strona dostarcza sekcji z referencjami do konkretnych skryptów

```
@RenderSection("sekcjaSkryptow", false)
```

(a inne strony mogą tej sekcji nie mieć)

```
@section SekcjaSkryptow {
    <div>
    Ta sekcja jest opcjonalna w stronie layoutowej, strony mogą ją dostarczać lu
b nie
    </div>
}
```

11 Testy jednostkowe

Z uwagi na możliwość wykonywania metod kontrolerów poza kontekstem aktualnych żądań HTTP, framework MVC jest lepiej przystosowany do pisania testów jednostkowych. W przyszłości pokażemy jednak alternatywę do testów jednostkowych na poziomie kontrolerów/metod.