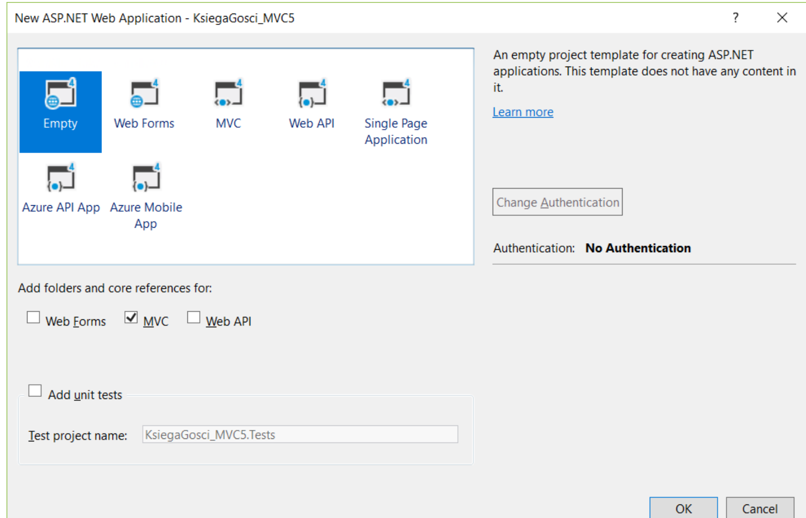
Księga gości w ASP.NET MVC 5 (Visual Studio 2017)

Jacek Matulewski

Tworzenie projektu

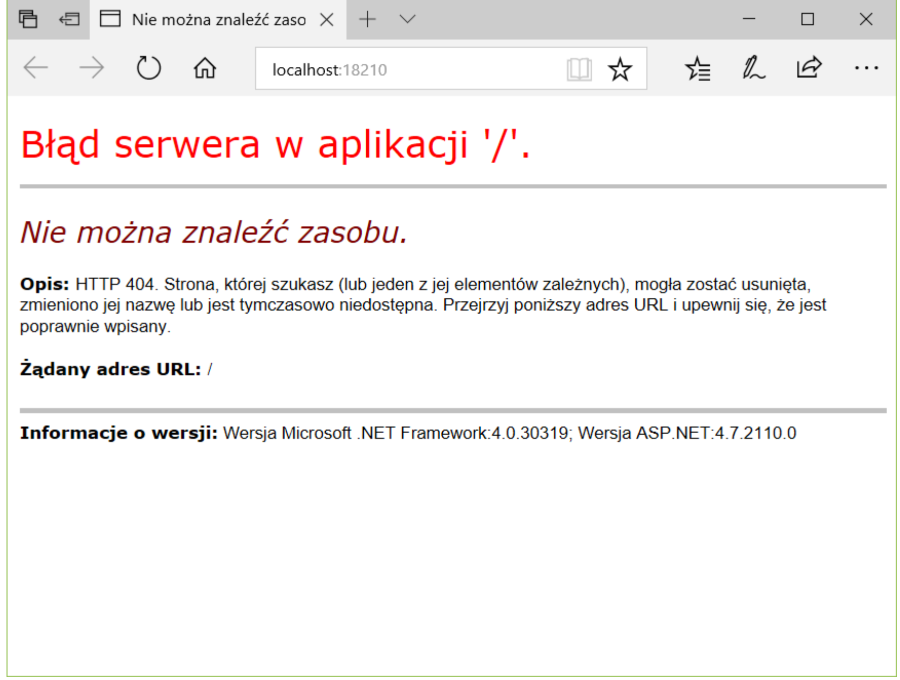
Będziemy budować internetową księgę gości w ASP.NET MVC 5 (Visual Studio 2017).

1. Tworzymy projekt typu *ASP.NET Web Application (.NET Framework)* o nazwie „KsiegaGosci_MVC5”.
2. Po kliknięciu *OK* pojawi się okno dialogowe *New ASP.NET Web Application – KsiegaGosci_MVC5*, w którym wybieramy szablon *Empty* oraz zaznaczamy pole opcji *MVC*. Wybór potwierdzamy, klikając *OK*.

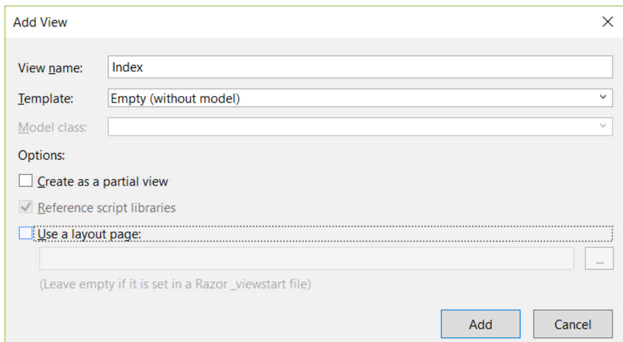


3. Powstanie projekt, który wbrew nazwie szablonu nie jest całkowicie pusty. Zawiera m.in. katalog ze skryptami (m.in. jQuery) oraz pliki *Web.config* i *Global.asax*. Ma również puste katalogi *Models* i *Controllers* oraz katalog *Views*, w którym są pliki domyślnego „layoutu” i strony wyświetlanej w razie błędu.

Próba uruchomienia projektu zaraz po jego utworzeniu powoduje pojawienie się błędu związanego z błędnym trasowaniem (rysunek).



4. Do projektu witryny chcemy dodać stronę wyświetlającą wpisy internautów i pozwalającą na dodawanie nowych.
   Rozwój projektu rozpoczniemy od dodania do niego kontrolera. W tym celu:
   a. W podoknie *Solution Explorer* zaznacz katalog *Controllers* i z jego menu kontekstowego wybierz *Add/Controller...*.
   b. W nowym oknie dialogowym wybieramy szablon *MVC 5 Controller – Empty*. Klikamy *Add*.
   c. Pojawi się małe okno dialogowe do wpisania nazwy. Wpiszmy „KsiegaGosciController” i kliknijmy *Add*.
   d. Powstanie plik *Controllers\KsiegaGosciController.cs*, w którym znajduje się klasa kontrolera o nazwie `KsiegaGosciController` z jedną bezargumentową metodą `Index` zwracającą obiekt typu `ActionResult`. Metoda ta, nazywana akcją kontrolera, zwraca widok będący podstawą kodu HTML wysyłanego do przeglądarki.
5. Kolejnym krokiem będzie utworzenie tego widoku.
   a. Najprościej jest go utworzyć, klikając prawym przyciskiem myszy w obrębie metody `Index` i klikając *Add View...*.
   b. Użyjmy proponowanej przez kreator nazwy widoku (tj. „Index”). Usuwamy zaznaczenie przy opcji *Use a layout page* — nasz projekt będzie zawierał tylko jedną stronę, więc nie ma sensu używanie stron wzorcowych lub układów, których celem jest ujednolicenie stron i unikanie powtarzania tego samego kodu w różnych widokach.



   c. Następnie klikamy *Add*. Powstanie plik *Views\KsiegaGosci\Index.cshtml*, którego zawartość widoczna jest na listingu 1.

Listing 1. Poza pierwszymi liniami, kod pustego widoku to czysty HTML, zresztą niezbyt wyrafinowany

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

Znak `@` sygnalizuje instrukcję (`for`, `if`, itp.) lub grupę instrukcji w operatorze zakresu `@{ }` w języku C# interpretowane przez silnik Razor.

Widok — formularz dodawania wpisu

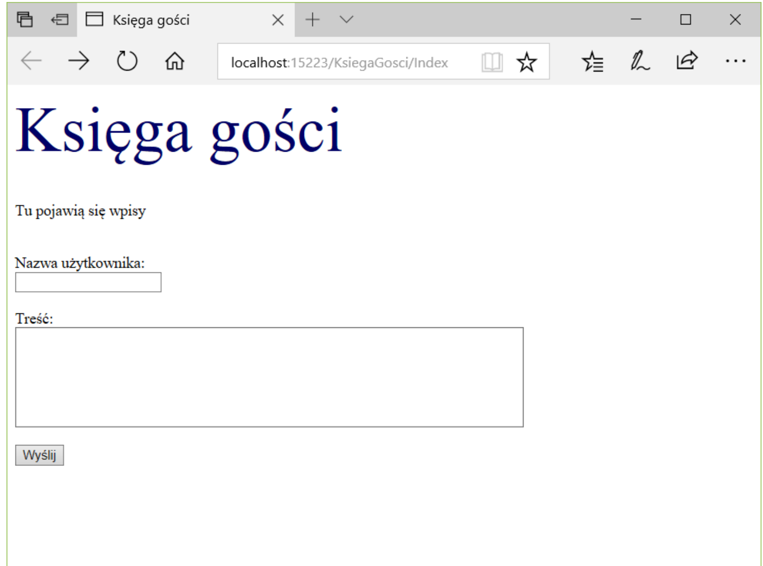
Zajmijmy się teraz tworzeniem interfejsu księgi gości. Będzie składał się z dwóch części. W górnej części strony wyświetlane będą wpisy. Natomiast pod nimi znajdować się będzie formularz umożliwiający dodanie nowego wpisu. Zacznijmy od utworzenia formularza, dodając do pliku *Index.cshtml* kod zaznaczony na listingu 2.

Listing 2. Zachowałem nazwy elementów formularza (TextBox1 i TextBox2), aby ułatwić porównanie z szablonem utworzonym w Web Forms

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Księga gości</title>
</head>
<body>
    <div>

        <span style="font-size:50pt;color:#000066;">Księga gości</span><br />
        <br />
        <br />
```

```
            Tu pojawią się wpisy
            <br />
            <br />
            <br />
            @using(Html.BeginForm()) {
                <div>
                    Nazwa użytkownika:<br />
                    @Html.TextBox("TextBox1")
                    <br />
                    <br />
                    Treść:<br />
                    @Html.TextArea("TextBox2","",new { style = "height:98px;width:525px;" })
                    <br />
                    <br />
                    <input type="submit" id="Button1" value="Wyślij"/>
                </div>
            }

        </div>
    </body>
    </html>
```

Warto sprawdzić, jak powyższy kod formularza będzie wyglądał w przeglądarce. Po naciśnięciu klawisza F5, aplikacja zostanie uruchomiona na lokalnym serwerze i zobaczymy w przeglądarce formularz (adres *http://localhost:15523/KsiegaGosci/Index*).



Model — obsługa plików XML

Wysłanie danych z formularza pokazywanego użytkownikowi w przeglądarce do aplikacji powinno spowodować dodanie nowego rekordu do danych, a następnie uwzględnienie go w liście wpisów wyświetlanych na tej samej stronie po jej przeładowaniu. Wpis ten powinien być również widoczny przez innych użytkowników strony. Oznacza to, że po stronie serwera konieczny jest mechanizm trwałego przechowywania danych. Naturalne byłoby użycie do tego bazy danych, lecz my użyjemy pliku XML.

Do katalogu *Models* dodajmy plik o nazwie *Wpisy.cs* z klasą `Wpisy`. To będzie główna klasa modelu. Model jest niezależną od pozostałych modułów częścią aplikacji przechowującą dane, które przetwarza aplikacja – w naszym przypadku jest to kolekcja wpisów do księgi gości.

1. Z menu kontekstowego katalogu Models wybieramy *Add*, *Class…*.
2. W oknie dialogowym w polu *Name* wpiszmy *Wpisy.cs*. Zdefiniowana w tym pliku klasa zostanie automatycznie nazwana `Wpisy`.
3. Kliknij przycisk *Add*.
4. Powstanie plik, w którym należy umieścić kod widoczny na listingu 3

Listing 3. Separację modelu wymusza wzorzec MVC

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

namespace KsiegaGosci_MVC5.Models
{
    //Struktura opisująca elementarny typ danych (klasa encji)
    public struct Wpis
    {
        private string nazwaUżytkownika;
        private string treść;
        private DateTime data;
        private string adresIP;

        public string NazwaUżytkownika { get { return nazwaUżytkownika; } }
        public string Treść { get { return treść; } }
        public DateTime Data { get { return data; } }
        public string AdresIP { get { return adresIP; } }

        public Wpis(string nazwaUżytkownika, string treść, DateTime data, string 
adresIP)
        {
            this.nazwaUżytkownika = nazwaUżytkownika;
            this.treść = treść;
            this.data = data;
            this.adresIP = adresIP;
        }
    }

    //Model księgi gości. Tu zapisana jest logika biznesowa aplikacji
    public class Wpisy
    {
        List<Wpis> wpisy = new List<Wpis>();

        public bool ZapiszDoPlikuPoKażdymDodaniuWpisu = true;
```

```
public void Dodaj(Wpis nowyWpis)
{
    wpisy.Add(nowyWpis);
    if (ZapiszDoPlikuPoKażdymDodaniuWpisu) ZapiszDoPlikuXml();
}

public int Liczba
{
    get
    {
        return wpisy.Count;
    }
}

public Wpis this[int i]
{
    get
    {
        return wpisy[i];
    }
}

public bool ZapiszDoPlikuXml(string nazwaPlikuXml = null)
{
    if (nazwaPlikuXml == null) nazwaPlikuXml = this.nazwaPlikuXml;

    try
    {
        if (System.IO.File.Exists(nazwaPlikuXml))
            System.IO.File.Copy(nazwaPlikuXml, nazwaPlikuXml + ".bak", true);

        XDocument xml = new XDocument(
            new XDeclaration("1.0", "utf-8", "yes"),
            new XElement("Wpisy",
                new XElement("DataZapisu", DateTime.Now.ToString()),
                from wpis in wpisy
                select new XElement("Wpis",
                    new XElement("NazwaUżytkownika", wpis.NazwaUżytkownika),
                    new XElement("Treść", wpis.Treść),
                    new XElement("Data", wpis.Data),
                    new XElement("AdresIP", wpis.AdresIP)
                )
            )
        );

        xml.Save(nazwaPlikuXml);
```

```
                //MvcApplication.DopiszDoPlikuLog("Zapis prawidłowy");
                return true;
            }
            catch (Exception exc)
            {
                //MvcApplication.DopiszDoPlikuLog(exc.Message);
                return false;
            }
        }

        private bool CzytajZPlikuXml(string nazwaPlikuXml)
        {
            try
            {
                XDocument xml = XDocument.Load(nazwaPlikuXml);

                IEnumerable<Wpis> odczytaneWpisy =
                    from wpis in xml.Descendants("Wpis")
                    select
                       new Wpis(
                           wpis.Element("NazwaUżytkownika").Value,
                           wpis.Element("Treść").Value,
                           DateTime.Parse(wpis.Element("Data").Value),
                           wpis.Element("AdresIP").Value);

                wpisy.Clear();
                wpisy.AddRange(odczytaneWpisy);

                return true;
            }
            catch
            {
                return false;
            }
        }

        string nazwaPlikuXml;

        public Wpisy(string nazwaPlikuXml)
        {
            this.nazwaPlikuXml = nazwaPlikuXml;
            CzytajZPlikuXml(nazwaPlikuXml);
        }
    }
}
```

Aplikacja — rejestrowanie zdarzeń (log)

W listingu 1 znajdują się dwa zakomentowane wywołania metody zapisującej informacje do pliku tekstowego, który pozwoli administratorowi aplikacji na śledzenie obsługi żądań i reakcji na nie. Teraz należy tą funkcję zdefiniować.

W pliku *Global.asax* obecnym w projekcie znajduje się klasa `KsiegaGosci_MVC5.MvcApplication`, która podobnie jak klasa `Global` z projektów Web Forms, dziedziczy z klasy `HttpApplication`. W klasie tej znajdziemy definicję metody `Application_Start` uruchamianej przy uruchomieniu aplikacji na serwerze.

Do tej klasy dopiszmy prostą metodę statyczną o nazwie `DopiszDoPlikuLog`, dopisującą do pliku tekstowego, którego nazwa przechowywana jest w stałej `nazwaPliku_Log`, łańcuch podany w argumencie (listing 4). Ta właśnie metoda powinna być wywoływana w razie zgłoszenia wyjątku podczas zapisywania danych do pliku przez metodę `Wpisy.ZapiszDoPlikuXml` (zob. listing 3) – należy jej wywołanie „odkomentować". Aby zadziałała, należy zadeklarować użycie przestrzeni nazw `System.IO`.

Listing 4. Metoda pomocnicza umożliwiająca rejestrowanie zdarzeń aplikacji

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

namespace KsiegaGosci_MVC5
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }

        private const string nazwaPliku_Log = @"d:\KsiegaGosci.log";

        public static void DopiszDoPlikuLog(string informacja)
        {
            if (File.Exists(nazwaPliku_Log))
                File.Copy(nazwaPliku_Log, nazwaPliku_Log + ".bak", true);
            using (StreamWriter sw = new StreamWriter(nazwaPliku_Log, true))
            {
                sw.WriteLine(DateTime.Now.ToString() + ": " + informacja);
            }
        }
    }
}
```

Aplikacja – trasowanie

W metodzie `MvcApplication.Application_Start` znajduje się wywołanie metody `RouteConfig.RegisterRoutes`, której definicję znajdziemy w pliku *App_Start\RouteConfig.cs*. Przyglądając się jej kodowi, zobaczymy, że rejestruje ona jedno odwzorowanie trasy: adres domyślny aplikacji (tj. bez podkatalogów lub nazw plików, np. *http://localhost:*numer_portu) „mapowany” jest na akcję `Index` w kontrolerze *Home*. Takiego kontrolera pusty szablon ASP.NET MVC jednak nie zawiera i stąd komunikat o błędzie pojawiający się przy próbie uruchomienia aplikacji zaraz po jego utworzeniu. Jednak jeżeli zastąpimy ten kontroler dodanym wcześniej kontrolerem `KsiegaGosci` (listing 5), to wywoływana będzie obecna w nim akcja `Index` zwracająca widok z formularzem. W nazwie kontrolera można pominąć „..Controller”.

Listing 5. Wiązanie adresu URL z akcją kontrolera, czyli trasowanie lub routing

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default", //nazwa routingu
        "{controller}/{action}/{id}", //adres URL z parametrami
        new { controller = "KsiegaGosci", action = "Index", id = UrlParameter.Optional }
    );
}
```

GET i POST w protokole HTTP

Są dwa sposoby przekazywania danych między kolejnymi odsłonami stron WWW w protokole HTTP: metoda GET i POST. W metodzie GET dane (wartości parametrów) doczepiane są do URL np. *http://domena.serwer/index.php?lang=pl&cat=2&sidebar=yes*. Nie należy tak przesyłać loginów, haseł i innych danych wrażliwych. W metodzie POST dane przesyłane są w pakiecie np.

```
POST /login.jsp HTTP/1.1
Host: www.strona.com
User-Agent: Mozilla/5.0
Content-Length: 30
Content-Type: application/x-www-form-urlencoded

login=jacek&password=hasło
```

Metodę GET obsługuje metoda kontrolera, która ma atrybut `[HttpGet]` lub nie ma żadnego atrybutu. Metodę POST obsługuje metoda z atrybutem `[HttpPost]`. Dostęp do nagłówka POST można uzyskać poprzez `Request.Headers`, a do przechowywanych w nim parametrów przez metodę `Request.Headers.GetValues`.

Kontroler — obsługa POST

Użyjmy klasy `Wpisy` jako modelu. Aby to zrobić, musimy „osadzić” go w kontrolerze, tworząc jego instancję. W tym celu do klasy `KsiegaGosciController` dopiszmy dwa pola:

```
public const string nazwaPliku_WpisyXml = @"d:\wpisy.xml";
Wpisy wpisy = new Wpisy(nazwaPliku_WpisyXml);
```

Aby klasa `Wpisy` była widoczna w kontrolerze, do jego nagłówka należy dodać deklarację użycia przestrzeni nazw, w której jest zdefiniowana:

```
using KsiegaGosci_MVC5.Models;
```

Następnie przeciążmy metodę `Index`, tworząc jej wersję wywoływaną, gdy dane formularza przekazywane są z przeglądarki do aplikacji ASP.NET. Wykorzystywana jest wówczas metoda POST, która dane te ukrywa w pakiecie, nie dodając ich do adresu (jak to ma miejsce w metodzie GET). Drugą metodę `Index` pokazuje listing 6.

Listing 6. Metoda obsługująca zapytania z danymi przesyłanymi metodą POST

```
[HttpPost]
public ActionResult Index(FormCollection formularz)
{
    string nazwaUzytkownika = formularz["TextBox1"];
    string trescWpisu = formularz["TextBox2"];

    if (nazwaUzytkownika != "" && trescWpisu != "") //walidacja po stronie serwera
    {
        //uzupełnianie danych modelu
        Wpis nowyWpis = new Wpis(nazwaUzytkownika.Trim(), trescWpisu.Trim(),
                                 DateTime.Now, Request.UserHostAddress);
        wpisy.Dodaj(nowyWpis);

        ZapamietajNazwęUżytkownika(nazwaUzytkownika); //zapis do cookie
        ViewData.Add("nazwaOstatniegoUżytkownika", nazwaUzytkownika);
    }

    return View(wpisy);
}

private void ZapamietajNazwęUżytkownika(string nazwaUżytkownika)
{
    HttpCookie ciasteczko = new HttpCookie("nazwaUzytkownika", nazwaUżytkownika);
    ciasteczko.Expires = DateTime.Today.AddDays(3);
    Response.Cookies.Add(ciasteczko);
}
```

W nowej metodzie nie ma elementów związanych w wyświetlaniem wpisów. Za wyświetlanie wpisów odpowiedzialny będzie wyłącznie widok, do którego przekażemy jedynie model zawierający zaktualizowane dane. To widok na podstawie modelu przygotuje kod HTML.

Na powyższym listingu znajduje się również metoda pomocnicza `ZapamiętajNazwęUżytkownika`. Zwróćmy uwagę, że zawiera odwołania do obiektu `Response` reprezentującego odpowiedź aplikacji ASP.NET odsyłaną do przeglądarki. Dostępny jest również obiekt `Request` reprezentujący zapytanie przesłane z przeglądarki do aplikacji. W metodzie `Index` używamy go do odczytania adresu IP klienta.

Widok — wyświetlanie wpisów

Nowa metoda `Index` zwraca widok, przekazując do niego obiekt modelu uzyskany dzięki wywołaniu metody `View` zdefiniowanego w klasie bazowej kontrolera (`return View(wpisy)`). Dane o wpisach do księgi gości są w ten sposób dostępne z poziomu widoku, gdzie można ich użyć do wyświetlenia zawartości księgi gości (listing 7). Do widoku przekazywana jest również nazwa użytkownika, który dodał ostatni wpis. Używam do tego obiektu `ViewData`. Wszystkie zmiany, jakie należy wprowadzić do pliku widoku (plik *Views\KsiegaGosci\Index.cshtml*), zostały na listingu wyróżnione pogrubieniem.

Listing 7. Model to dane przesyłane z kontrolera do widoku

```
@model KsiegaGosci_MVC5.Models.Wpisy

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <title>Księga gości</title>
</head>
<body>
    <div>

        <span style="font-size:50pt;color:#000066;">Księga gości</span><br />
        <br />
        <br />
        @for (int i = Model.Liczba - 1; i >= 0; --i)
        {
            string kolorParzysty = "navy";
            string kolorNieparzysty = "blue";
            string kolorNagłówka = (i % 2 == 0) ? kolorParzysty : kolorNieparzysty;

            KsiegaGosci_MVC5.Models.Wpis wpis = Model[i];

            <div>
            <font color="@kolorNagłówka"><b>Dodano dnia
@Html.Encode(wpis.Data.ToString())</b></font>
            <br />
            <div style="white-space:pre;">@Html.Encode(wpis.Treść.Trim())</div>
            <i>@Html.Encode(wpis.NazwaUżytkownika)</i> (@Html.Encode(wpis.AdresIP))
            <p /><hr width='30%' align='left' />
            </div>
        }
        <br />
        <br />
        <br />
        @using(Html.BeginForm())
        {
            string nazwaUżytkownika;

            <div>
                Nazwa użytkownika:<br />
                @if (ViewData["nazwaOstatniegoUżytkownika"] == null ||
                     ViewData["nazwaOstatniegoUżytkownika"] as string == "")
```

```
                {
                    nazwaUżytkownika = Request.Cookies["nazwaUzytkownika"] == null ? ""
: Request.Cookies["nazwaUzytkownika"].Value;
                }
                else
                {
                    nazwaUżytkownika=ViewData["nazwaOstatniegoUżytkownika"] as string;
                }
                @Html.TextBox("TextBox1", nazwaUżytkownika)
                <br />
                <br />
                Treść:<br />
                @Html.TextArea("TextBox2","",new { style = "height:98px;width:525px;" })
                <br />
                <br />
                <input type="submit" id="Button1" value="Wyślij"/>
            </div>
        }

    </div>
</body>
</html>
```

Przede wszystkim zauważymy, że do kodu widoku dodaliśmy pętlę `for`, podobną do tej, jakiej można użyć w języku C#. Pętla ta oznacza, że kod HTML w niej zawarty zostanie powtórzony dla każdego rekordu tabeli (tj. dla każdego wpisu w księdze gości). Do określenia formatu wyświetlania wpisu używamy znaczników HTML (używam znacznika `font`, który jest przestarzały, ale wygodny – powinno go zastąpić formatowanie za pomocą kaskadowych arkuszy stylu).

W pierwszej chwili może oczywiście razić mieszanie kodu HTML i C# w plikach *.cshtml*, szczególnie jeżeli mamy nawyk wyrobiony w projektach Web Forms, w których stara się je rozdzielać. Pamiętajmy jednak, że nie jest to zanurzanie „skryptów" w kodzie wysyłanym do przeglądarki. W plikach *.cshtml* składnia języka C# używana jest do przygotowywania szablonu, na podstawie którego silnik Razor generuje kod HTML wysyłany do przeglądarki. W nim już tej pętli nie będzie, a w zamian pojawi się seria znaczników, które ta pętla opisuje. Instrukcja `for` jest tu więc częścią metajęzyka ułatwiającego prezentowanie danych w HTML. To wygodne, a zarazem potężne narzędzie. Należy jednak liczyć się z tym, że instrukcje takie, jak pętle `for` lub instrukcje warunkowe `if` nie są kompilowane przez Visual Studio w trakcie kompilacji rozwiązania, a tym samym ich poprawność jest weryfikowana dopiero w trakcie działania aplikacji, gdy generowana jest odpowiedź wysyłana do przeglądarki. To zwiększa podatność projektu ASP.NET MVC na błędy i utrudnia jej debugowanie. Kompensuje to jednak łatwość, z jaką te projekty, w odróżnieniu od ASP.NET Web Forms, poddają się weryfikacji poprawności za pomocą testów jednostkowych. Stosowanie metodyki *test-driven developement* pozwala rozwiązać problem zarządzania błędami w projekcie w sposób konsekwentny.

Druga zmiana w kodzie widoku dotyczy domyślnej zawartości pola tekstowego. Jest ona ustalana na podstawie pliku *cookie*, który tworzymy w akcji kontrolera (por. metoda `ZapamietajNazwęUżytkownika` w klasie kontrolera). W ten sposób, nawet jeżeli wrócimy do strony dopiero następnego dnia, nie będziemy musieli ponownie wpisywać nazwy użytkownika. Jednak jeżeli z naszego komputera skorzysta ktoś inny, wpisując inną nazwę użytkownika, akcja wprawdzie zaktualizuje plik *cookie*, ale zmiana nie będzie widoczna w odpowiedzi odesłanej tuż po tym zdarzeniu. Dlatego przesyłam ją do widoku osobno za pomocą klasy `ViewData`.

Kontroler – obsługa GET

Ponieważ zamiast tworzyć osobny widok dla nowej akcji, zmodyfikowaliśmy istniejący widok, dodając do niego model, powinniśmy także zmodyfikować akcję `Index` reagującą na zapytanie z przeglądarki przesyłające dane

metodą GET. Także w tym przypadku do widoku powinien być przesyłany model. Mówiąc prościej, należy zmodyfikować także bezargumentową metodę `Index` kontrolera zgodnie ze wzorem z listingu 8.

Listing 8. Przesyłanie modelu z kontrolera do widoku

```
public ActionResult Index()
{
    return View(wpisy);
}
```

Widok — walidacja

Na koniec zostawiłem problem weryfikacji danych wpisywanych do formularza. Dane te są wprawdzie sprawdzane w kontrolerze przed dodaniem wpisu do danych przechowywanych w modelu (tj. po stronie serwera), ale oczywiście warto również walidować je po stronie klienta.

Sercem walidacji w MVC jest odpowiednie „udekorowanie” klasy encji atrybutami określającymi wymagania względem wartości przypisywanych poszczególnym jej właściwościom. Atrybuty, które dodałem do struktury `Wpis` (plik *Models\Wpisy.cs*), prezentuje listing 9. Wymagają one zaimportowania dwóch przestrzeni nazw: `System.ComponentModel` i `System.ComponentModel.DataAnnotations`.

Listing 9. Własności klasy encji z atrybutami określającymi warunki nałożone na przypisywane im wartości

```
public struct Wpis
{
    private string nazwaUżytkownika;
    private string treść;
    private DateTime data;
    private string adresIP;

    [Required(ErrorMessage="Wymagane jest wpisanie nazwy użytkownika")]
    [StringLength(256, MinimumLength = 1, ErrorMessage = "Niepoprawna długość nazwy 
użytkownika")]
    [RegularExpression(@"[a-zA-Z0-9'.ąćęłńóśżźĄĆĘŁŃÓŚŻŹ\s]{1,256}", ErrorMessage="Nazwa 
użytkownika zawiera niepoprawne znaki")]
    [DisplayName("Nazwa użytkownika")]
    public string NazwaUżytkownika { get { return nazwaUżytkownika; } }

    [Required(ErrorMessage = "Wymagane jest wpisanie treści wpisu")]
    [StringLength(65535, MinimumLength = 1, ErrorMessage = "Niepoprawna długość wpisu")]
    [RegularExpression(@"[a-zA-Z0-9'.,ąćęłńóśżźĄĆĘŁŃÓŚŻŹ\s]{1,65535}", ErrorMessage = 
"Wpis zawiera niepoprawne znaki")]
    public string Treść { get { return treść; } }

    [Required]
    public DateTime Data { get { return data; } }

    [Required]
    public string AdresIP { get { return adresIP; } }

    public Wpis(string nazwaUżytkownika, string treść, DateTime data, string adresIP)
    {
        this.nazwaUżytkownika = nazwaUżytkownika;
```

```
            this.treść = treść;
            this.data = data;
            this.adresIP = adresIP;
        }
    }
```

Zwróćmy uwagę, że warunki opisywane przez część z powyższych atrybutów powtarzają się. W szczególności w ogóle niepotrzebny jest atrybut `StringLength`, bo warunek dotyczący długości i tak jest zawarty w wyrażeniu regularnym. Chodziło mi jednak o to, żeby przejrzeć najczęściej używane atrybuty.

Pośród atrybutów dodanych do własności `NazwaUżytkownika` struktury `Wpis` jest też taki, który nie jest bezpośrednio związany z walidacją — mam na myśli atrybut `DisplayName`. Definiuje on przyjazną nazwę własności. Użyjemy jej do stworzenia etykiet formularza.

Po uzupełnieniu modelu możemy zająć się korzystającymi z niego kontrolerem i widokiem. Wiązanie pól z elementami HTML następuje poprzez ich identyfikatory. Dlatego zamiast identyfikatorów `TextBox1` i `TextBox2` w przypadku pól tekstowych powinniśmy teraz użyć identyfikatorów `NazwaUżytkownika` i `Treść`, tj. takich samych, jak nazwy dwóch właściwości struktury `Wpis` — nowej klasy modelu, której będzie używał formularz z walidatorami po kolejnych zmianach. I tu właśnie pojawia się problem. Przecież nasz widok już ma model — jest nim klasa `Wpisy`, której instancji używamy w części widoku odpowiedzialnej za wyświetlanie wpisów z księgi gości. Niestety, jeżeli formularz i wyświetlanie widoków chcemy umieścić na jednej stronie, a tak zaplanowałem, przewagę ma formularz. Instancję klasy `Wpisy`, której wartości są używane biernie, tj. tylko odczytywane, bez ich modyfikacji, możemy przekazać z kontrolera do widoku w inny sposób, choćby korzystając z `ViewData` lub `ViewBag`.

Wróćmy zatem do widoku (plik *Views\KsiegaGosci\Index.cshtml*) i zmieńmy pierwszą linię określającą model:

```
@model KsiegaGosci_MVC5.Models.Wpis
```

lub lepiej

```
@using KsiegaGosci_MVC5.Models;
@model Wpis
```

W tym drugim przypadku klasy `Wpis` i `Wpisy` nie będą wymagały wskazywania ich przestrzeni nazw.

Zmiana modelu powoduje nieco komplikacji, którymi po kolei się zajmiemy. Przede wszystkim, przywróćmy sprawność tej części kodu widoku, która jest odpowiedzialna za wyświetlanie wpisów. Załóżmy, że instancja klasy `Wpisy` zostanie przekazana za pomocą `ViewData` (identyfikujący ją łańcuch to „wpisy”). Wówczas w kodzie tym wystarczą zmiany zaznaczone na listingu 10.

Listing 10. Zmiana sposobu przekazywania danych o wpisach z kontrolera do widoku

```
@{
    Wpisy wpisy=ViewData["wpisy"] as Wpisy;
    for (int i = wpisy.Liczba - 1; i >= 0; --i)
    {
        string kolorParzysty = "navy";
        string kolorNieparzysty = "blue";
        string kolorNagłówka = (i % 2 == 0) ? kolorParzysty : kolorNieparzysty;

        Wpis wpis = wpisy[i];

        <div>
            <font color="@kolorNagłówka"><b>Dodano dnia
                                              @Html.Encode(wpis.Data.ToString())</b></font>
            <br />
            <div style="white-space:pre;">@Html.Encode(wpis.Treść.Trim())</div>
            <i>@Html.Encode(wpis.NazwaUżytkownika)</i> (@Html.Encode(wpis.AdresIP))
```

```
            <p /><hr width='30%' align='left' />
        </div>
    }
}
```

Kolejnym krokiem będzie zmodyfikowanie formularza. Listing 11 pokazuje, jakie zmiany należy w nim wprowadzić. Jest to więc zapowiedziana zmiana identyfikatorów pól tekstowych. Tych samych identyfikatorów używamy, korzystając z metody `ValidationMessage` klasy `HtmlHelper`, aby określić weryfikowane pola. Ponadto, aby zebrać wszystkie komunikaty w jednym miejscu, można użyć metody `ValidationSummary` (na listingu w komentarzu). Ostatnia zmiana dotyczy sposobu wyświetlania etykiet – użyłem metody `HtmlHelper.Label`, która wykorzystuje nazwy zdefiniowane w atrybucie `DisplayName`.

Listing 11. Wiązanie widoku z modelem Wpis jest dość „ciasne”

```
@using (Html.BeginForm())
{
    string nazwaUżytkownika;

    <div>
        @Html.Label("NazwaUżytkownika"):<br />
        @if (ViewData["nazwaOstatniegoUżytkownika"] == null ||
ViewData["nazwaOstatniegoUżytkownika"] as string == "")
        {
            nazwaUżytkownika =
                Request.Cookies["nazwaUzytkownika"] == null ?"":
Request.Cookies["nazwaUzytkownika"].Value;
         }
         else
         {
             nazwaUżytkownika = ViewData["nazwaOstatniegoUżytkownika"] as string;
         }
         @Html.TextBox("NazwaUżytkownika", nazwaUżytkownika)
         @Html.ValidationMessage("NazwaUżytkownika")
         <br />
         <br />
         @Html.Label("Treść"):<br />
         @Html.TextArea("Treść", "", new { style = "height:98px;width:525px;" })
         @Html.ValidationMessage("Treść")
         <br />
         <br />

         @*
         <p>
         @Html.ValidationSummary("Popraw następujące błędy:",new { style="color:red" })
         </p>
         *@

         <input type="submit" id="Button1" value="Wyślij"/>
     </div>
 }
```

`*@ … @*` to komentarz wieloliniowy w plikach *.cshtml*.

Walidacja działająca po stronie klienta nie może być zrealizowana inaczej niż za pomocą JavaScript. W przypadku ASP.NET MVC skrypty te bazują na jQuery. A zatem aby działały, konieczne jest dodanie szeregu referencji widocznych na listingu 12. Przy okazji można zdefiniować używane w tych skryptach style CSS. Również one widoczne są na listingu 12 (skopiowałem je zresztą z dodawanego do projektu pliku *Content\Site.css*).

Listing 12. Skrypty i style wykorzystywane przez walidatory

```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Księga gości</title>

    @* Referencje skryptów używanych do walidacji *@
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/MicrosoftAjax.js")"
type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.validate.js")"
type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")"
type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/MicrosoftMvcValidation.js")"
type="text/javascript"></script>

    <style type="text/css">
        .field-validation-error
        {
            color: #ff0000;
        }

        .field-validation-valid
        {
            display: none;
        }

        .input-validation-error
        {
            border: 1px solid #ff0000;
            background-color: #ffeeee;
        }
    </style>
</head>
```

To już wszystkie zmiany w widoku. Przejdźmy zatem do kontrolera (plik *Controllers\KsiegaGosciController.cs*). Tu zmiany będą na szczęście mniejsze. Przede wszystkim konieczne jest przesłanie do widoku listy wpisów za pomocą klasy `ViewData` i jednocześnie podanie instancji struktury `Wpis` jako nowego modelu. W przypadku akcji `Index` obsługującej zapytania GET będzie to wyglądać następująco:

Listing 13. Zmiana sposobu przesyłania danych do widoku widziana od strony kontrolera

```
public ActionResult Index()
{
```

```
        ViewData.Add("wpisy", wpisy);
        return View(new Wpis());
    }
```

W drugiej akcji, która obsługuje zapytania POST, należy uwzględnić także zmienione identyfikatory pól tekstowych formularza. Nowa wersja tej akcji może wyglądać tak, jak przedstawiona na listingu 14.

Listing 14. Zmiany w kontrolerze wynikające ze zmiany sposobu przekazywania danych do widoku

```
    [HttpPost]
    public ActionResult Index(FormCollection formularz)
    {
        string nazwaUzytkownika = formularz["NazwaUżytkownika"]; //dawniej "TextBox1"
        string trescWpisu = formularz["Treść"]; //dawniej "TextBox2"

        if(ModelState.IsValid) //walidacja po stronie serwera
        {
            //uzupełnianie danych modelu
            Wpis nowyWpis = new Wpis(
                nazwaUzytkownika.Trim(),
                trescWpisu.Trim(),
                DateTime.Now,
                Request.UserHostAddress);
            wpisy.Dodaj(nowyWpis);

            ZapamietajNazwęUżytkownika(nazwaUzytkownika); //zapis do cookie
            ViewData.Add("nazwaOstatniegoUżytkownika", nazwaUzytkownika);
        }

        ViewData.Add("wpisy", wpisy);
        return View(new Wpis());
    }
```

Kontroler — użycie modelu odsyłanego przez widok

Zmiana modelu (jest nim teraz instancja struktury `Wpis`), poza ułatwieniem walidacji, ma również tę ważną zaletę, że możliwe staje się wówczas takie zdefiniowanie akcji kontrolera, aby jej argumentem był ów obiekt, ale już zainicjowany wartościami wpisanymi w pola formularza. Jest jednak pewien dość trywialny warunek — właściwości związane z polami formularza nie mogą być zdefiniowane jako tylko do odczytu. A właśnie tak jest w naszym przypadku. Proponuję zatem dopisać sekcje `set` do **każdej z tych własności**, np.

```
    public string NazwaUżytkownika
    {
        get { return nazwaUżytkownika; }
        set { nazwaUżytkownika = value; }
    }
```

Nawet wówczas musimy jednak pamiętać, że formularz pozwala na ustalenie wartości tylko dwóch z czterech pól modelu. A wszystkie cztery są przecież wymagane (atrybut `Required`). To oznacza, że również wartość pozostałych dwóch pól powinna być ustalana albo już w formularzu, albo w kontrolerze przed sprawdzeniem

wartości `ModelState.IsValidate`, aby walidacja była pomyślna. W pierwszym przypadku możemy do tego użyć dość typowej funkcjonalności formularzy HTML, a mianowicie ukrytych pól. Do kodu z pliku *Index.cshtml* w obrębie formularza dopiszmy zatem:

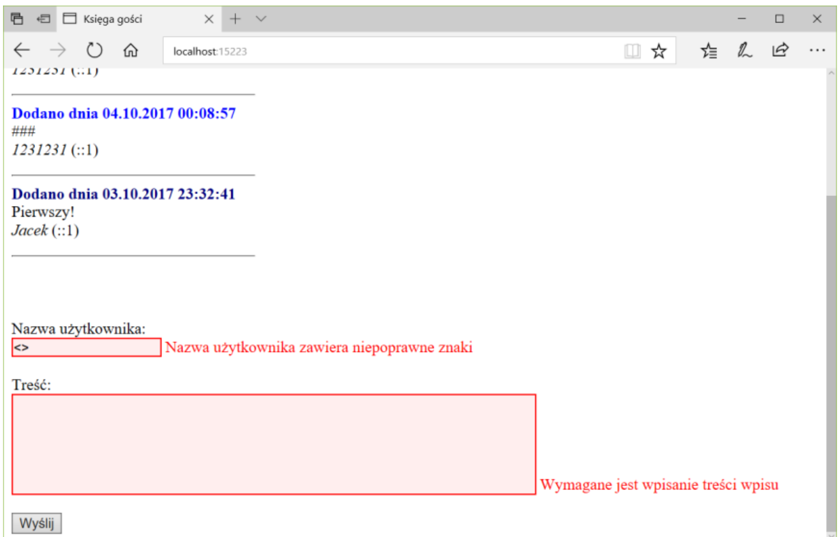
```
@Html.Hidden("AdresIP",Request.UserHostAddress)
@Html.Hidden("Data",DateTime.Now)
```

Wówczas akcję `Index` związaną z metodą POST przesyłania danych możemy uprościć do postaci widocznej na listingu 15.

Listing 15. Optymalna wersja akcji kontrolera reagującej na przesłany z widoku formularz zawierający model

```
[HttpPost]
public ActionResult Index(Wpis nowyWpis)
{
    if (ModelState.IsValid)
    {
        wpisy.Dodaj(nowyWpis);

        ZapamietajNazwęUżytkownika(nowyWpis.NazwaUżytkownika);
        ViewData.Add("nazwaOstatniegoUżytkownika", nowyWpis.NazwaUżytkownika);
    }

    ViewData.Add("wpisy", wpisy);
    return View(new Wpis());
}
```

Możemy teraz uruchomić program i sprawdzić, czy walidatory spełniają swoją funkcję. Próba kliknięcia przycisku „Wyślij” bez wypełnienia formularza lub gdy formularz wypełniony jest nieprawidłowo (niewłaściwe znaki lub niewłaściwa długość), powinna zakończyć się wyświetleniem komunikatów, których przykłady widoczne są na rysunku.



Zwracając widok, przesyłamy do niego pusty obiekt typu `Wpis`. A co by się stało, gdybyśmy przesłali do niego jeden z istniejących obiektów tego typu przechowywany w klasie `Wpisy`? Jego zawartość znalazłaby się po prostu w polach edycyjnych. Zatem jeżeli chcemy do aplikacji dodać możliwość edytowania wpisów — to najprostszy sposób.