Projektowanie aplikacji ASP.NET

Wykład 08/15
ASP.NET MVC, routing

Wiktor Zychla 2019/2020

Spis treści

1 Routing ASP.NET

1.1 Omówienie

Jednym z ograniczeń klasycznego ASP.NET jest przywiązanie ścieżek (url) do fizycznych zasobów – w WebForms żądanie postaci /foo/bar/qux.aspx musi posiadać w systemie plików aplikacji na serwerze zasób w dokładnie takiej lokalizacji.

Co jednak zrobić w sytuacji w której, z różnych powodów, żądania powinny być adresowane inaczej niż zasoby fizyczne? Jednym z typowych scenariuszy jest budowanie *dynamicznego* adresowania. W klasycznym WebForms jest ono możliwe wyłącznie w obszarze *parametrów* żądań

http:/……./foo.aspx?parametry&dalszeparametry

nie jest natomiast łatwo możliwe na segmentach właściwego żądania

http:/……./parametry/dalszeparametry/foo.aspx

bo w przypadku gdy parametry żądania mogą przybierać wiele różnych wartości oznacza to konieczność posiadania wielu kopii tego samego zasobu w fizycznych lokalizacjach odpowiadających wszystkim możliwym wariantom parametrów.

Wczesne podejścia do tzw. dynamicznego adresowania szły w dwóch kierunkach

- Użycie metody **RewritePath** obiektu **HttpContext** pozwala na wczesnym etapie potoku przetwarzania (np. BeginRequest) „przepisać" adres żądania na inny zasób (czyli działa jak Server.Transfer tylko w przeciwieństwie do tamtego – nie wykonuje w ogóle próby przetwarzania oryginalnego żądania)
- Użycie modułów przepisujących adresy – jest to bardziej rozbudowana wersja poprzedniego podejścia, włącznie z tym że istnieją podejścia umożliwiające oddzielenie przepisywania adresów od samej aplikacji (np. rozszerzenie UrlRewriting do serwera IIS)

Ogólne rozwiązanie problemu pojawiło się dopiero w ASP.NET 4 i polega na rozbudowaniu potoku przetwarzania o wykorzystanie tzw. *routowania* (ASP.NET Routing).

Mechanizm routowania polega na dodaniu do potoku przetwarzania obiektu dziedziczącego z klasy Route, który dostarcza dwóch metod

- GetRouteData – zadaniem tej metody jest zbudowanie na podstawie adresu żądania tzw. tablicy routingu, która zawiera pary klucz-wartość, z której mogą korzystać dalsze elementy potoku przetwarzania. Tablica dostępna jest w HttpContext w

```
HttpContext.Current.Request.RequestContext.RouteData;
```

  i jest typu RouteData. Niepusta tablica routingu dla zadanego adresu oznacza, że nastąpiło dopasowanie ścieżki
- GetVirtualPath – zadaniem tej metody jest odbudowanie adresu na podstawie tablicy routingu (czyli działanie odwrotne do GetRouteData) i korzysta się z tej metody rzadziej (korzysta z niej m.in. klasa UrlHelper z MVC)

1.2 Przykład routing i multitenancy

Przykład jaki omówimy na wykładzie dotyczy scenariusza, w którym celem jest zbudowanie aplikacji typu CMS. CMS składa się z **witryn**, witryny składają się ze **stron**. Domyślną stroną witryny jest /index.html.

Oznacza to że poprawne są adresy

http://cms.local/site/page.html - oznacza żądanie do strony **page** w witrynie **site**

http://cms.local/site - oznacza żądanie do strony **index** w witrynie **site**

http://cms.local/site/subsite/subsubsite/anotherpage.html

- oznacza żądanie do strony **anotherpage** w witrynie **site/subsite/subsubsite**

W typowej implementacji struktura drzewiasta witryn i stron jest reprezentowana np. w bazie danych, natomiast z punktu widzenia serwera aplikacyjnego wyzwaniem jest tu dynamiczna struktura adresów, w której adres ma dowolnie wiele segmentów ścieżki (site/subsite/subsubsite) co oznacza że próba odwzorowania stron w strukturze fizycznej byłaby mocno uciążliwa.

Dodatkowym elementem o jaki wzbogacimy przykład jest obsługa wielu tzw. **tenantów** – powiemy o podejściu tzw. **multitenant**. Jest to taka architektura aplikacji webowej, w której z tej samej fizycznej aplikacji korzysta wielu niezależnych „klientów" rozumianych nie jako „użytkownicy" tylko „właściciele witryn". Pojęciem które stosuje się do architektury typu multitenant jest **multitenancy**, przekładane jako „wieloinstancyjność".

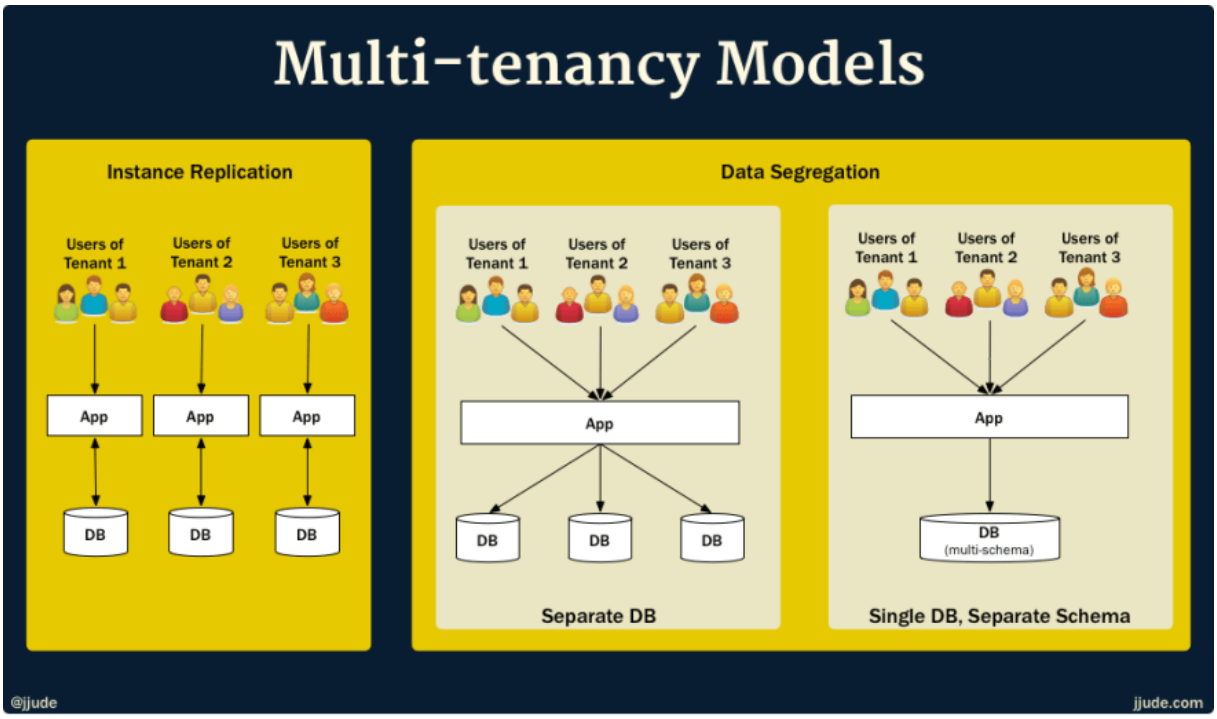
Potrzeba takiej architektury ma następujące uzasadnienie – wyobraźmy sobie witrynę sklepu, wystawioną na serwerze aplikacji pod adresem http://sklep.com. Sklep ma bazę danych towarów, rejestruje użytkowników – klientów, ma użytkowników – administratorów.

Po jakimś czasie pojawia się nowy klient – nowy sklep.

W naiwnym podejściu, duplikuje się instalację aplikacji tworząc nową witrynę na serwerze, http://sklep2.com Ta sama wersja aplikacji musi być więc instalowana dwa razy, za każdym razem po opublikowaniu nowej wersji.

Takie naiwne podejście replikowania instalacji przestaje się skalować szybko. Przy kilku klientach jest wyobrażalne, przy klientach idących w setki czy tysiące – nie.

Zamiast tego stosuje się architekturę w której **jedna instalacja aplikacji** obsługuje wiele baz danych niezależnych klientów lub jedną współdzieloną bazę w której każda dana jest oznakowana nazwą „tenanta" (czyli klienta) do którego należy:



Rysunek 1 za https://dev.to/jjude/what-is-a-multi-tenant-system-bpd

Z technicznego punktu widzenia do rozwiązania problemu wieloinstancyjności przydaje się możliwość umieszczenia nazwy instancji gdzieś w adresie, w taki sposób żeby poszczególni klienci w czytelny sposób wiedzieli jak adresować „swoją" instancję.

Możliwości są dwie:

- Nazwa instancji jako część nazwy nagłówka hosta – w tym podejściu zmienia się adresowanie aplikacji, zamiast http://www.app.com będzie http://klient1.app.com i http://klient2.app.com itd. Z punktu widzenia konfiguracji – ta sama witryna na IIS musi być skonfigurowana na obsługę wielu nagłówków (site bindings) ale o tym że tak można wiemy już z pierwszego wykładu. Problem zaczyna się przy wieluset i więcej klientach, gdy zarządzanie wiązaniami nagłówków do witryny musi być zautomatyzowane
- Nazwa instancji jako segment adresu żądania – w tym podejściu nie zmienia się nagłówek hosta, zamiast tego pojawia się dodatkowy segment adresu żądania. Mamy więc http://www.app.com/klient1/strona.aspx i http://www.app.com/klient2/strona.aspx

To drugie podejście – trudne w starszym ASP.NET, dzięki mechanizmowi routingu jest możliwe do wykonania.

Zobaczmy przykład klasy dziedziczącej z **Route**, implementującej obsługę ścieżek CMS w takiej postaci w której elementem ścieżki są trzy kategorie danych:

- Nazwa instancji
- Nazwa witryny
- Nazwa strony dla witryny

gdzie nazwa instancji to dodatkowy element ścieżki, oprócz omówionych już nazwy witryny i nazwy strony dla witryny.

```
namespace ASPRouting.Code
{
```

```
    /// <summary>
    /// Przykładowa routa z rzeczywistej aplikacji
    ///
    /// Adresowalna /[tenant]/[site]/[page.html]
    ///
    /// np.
    /// tenant1/site1/subsite1/page.html
    /// tenant2/site1
    /// </summary>
    public class CustomRoute : Route
    {
        public const string DEFAULTPAGEEXTENSION = ".html";

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

        public CustomRoute(
            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>
        /// <remarks>
        /// Wywołuje ją ASP.NET dla przychodzącgo URL
        /// </remarks>
        public override RouteData GetRouteData( HttpContextBase httpContext )
        {
            RouteData routeData = new RouteData( this, this.RouteHandler );

            string virtualPath =
                httpContext.Request.AppRelativeCurrentExecutionFilePath
                    .Substring( 2 ) + ( httpContext.Request.PathInfo ?? string.E
mpty );

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

            if ( segments.Length >= 1 )
            {
                routeData.Values[TENANT] = segments.First();

                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, segments.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[TENANT] is string )
                baseSegments.Add( (string)values[TENANT] );

            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.IsNu
llOrEmpty( s ) ) );

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

```
}
```

Żeby ASP.NET uwzględniał tę routę w globalnym routingu, należy ją skonfigurować w globalnej liście obsługiwanych rout, najlepiej w **Application_Start**:

```
public class Global : System.Web.HttpApplication
 {

     protected void Application_Start( object sender, EventArgs e )
     {
         RouteTable.Routes.Add(
             "customroute",
             new CustomRoute(
                 new RouteValueDictionary( new { tenant = "detault" } ),
                 new CustomRouteHandler() )
                 );
     }
```

Argumentem konstruktora routy jest obiekt typu **IRouteHandler.** Jego zadaniem jest wybranie handlera http dla zadanych parametrów, w tym może uwzględniać adres żądania, tablicę RouteData wypełnioną przez wcześniej uruchomioną routę, itp.

```
public class CustomRouteHandler : IRouteHandler
{
    #region IRouteHandler Members

    public IHttpHandler GetHttpHandler( RequestContext requestContext )
    {
        return new CustomHttpHandler();
    }

    #endregion
}
```

W przykładzie, route handler jest bardzo prosty, w dodatku nie zwraca żadnego wbudowanego handlera (typu **MvcRouteHandler** w którym nasz własny router delegowałby obsługę własnego formatu ścieżki do podsystemu MVC) tylko własny handler, w którym zademonstrujemy jak odczytywać wartości tablicy RouteData.

Uwaga! ASP.NET WebForms również integruje się z mechanizmem routingu. Zwykle nie pisze się własnego routera tylko używa metody rozszerzającej **MapPageRoute** która pozwala opisać własną postać ścieżki i zamapować ją na istniejącą stronę WebForms.

Ten własny handler mógłby wyglądać na przykład tak:

```
namespace ASPRouting.Code
{
    public class CustomHttpHandler : IHttpHandler
    {
```

```
        #region IHttpHandler Members

        public bool IsReusable
        {
            get { return true; }
        }

        public void ProcessRequest( HttpContext context )
        {
            var routeData = context.Request.RequestContext.RouteData.Values;

            string response =
                ( string.Format( "tenant: {0} site: {1} page: {2}", routeData["t
enant"], routeData["siteName"], routeData["pageName"] ) );

            context.Response.Write( response );
        }

        #endregion
    }
}
```

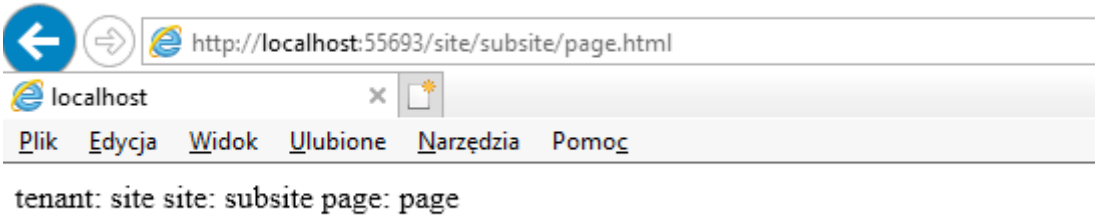
Aby cały przykład zadziałał, niezbędne jest jeszcze skonfigurowanie w **web.config** mapowania modułów ASP.NET dla **wszystkich** ścieżek

```
<configuration>

        <system.web>
                <compilation debug="true" targetFramework="4.5" />
                <httpRuntime targetFramework="4.5" />
        </system.web>

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

                </modules>
        </system.webServer>

</configuration>
```

Po uruchomieniu przykładu możliwa jest nawigacja do ścieżek postaci

/tenant/site/subsite/page.html

Uwaga! Ścieżka pusta nie jest obsługiwana przez powyższy router.



2 ASP.NET MVC - wprowadzenie

Podsystem MVC zmienia sposób wytwarzania klasycznych aplikacji webowych, wprowadzając na platformę ASP.NET implementację wzorca Model-View-Controller. Wykorzystuje omówiony wyżej mechanizm routingu do wychwycenia ścieżek dopasowujących się do konwencji

http://……/kontroler/akcja

i w wyniku wywołania takiej ścieżki uruchamia akcję **akcja** kontrolera **kontroler.**

Przykładowo: dla takiego kontrolera

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return Content("hello world");
    }
}
```

i takiego routingu, wykorzystującego gotowy route handler

```
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 = UrlPa
rameter.Optional }
           );
       }
   }
```

możliwe jest wołanie aplikacji dla ścieżek

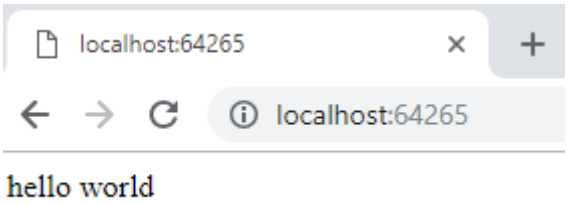
http://…..

http://……/home

oraz http ://…..home/index

(z uwagi na domyślne wartości skonfigurowane w **defaults** wszystkie trzy ścieżki są równoważne)

Efektem wywołania jest



Uwaga! Zainicjowanie tablicy routingu musi być zawołane np. w Application_Start:

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