Projektowanie aplikacji ASP.NET

Wykład 04/15
Routing, NET Core
Wprowadzenie do MVC

Wiktor Zychla 2024/2025

Spis treści

1 Routing w NET.Framework

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".

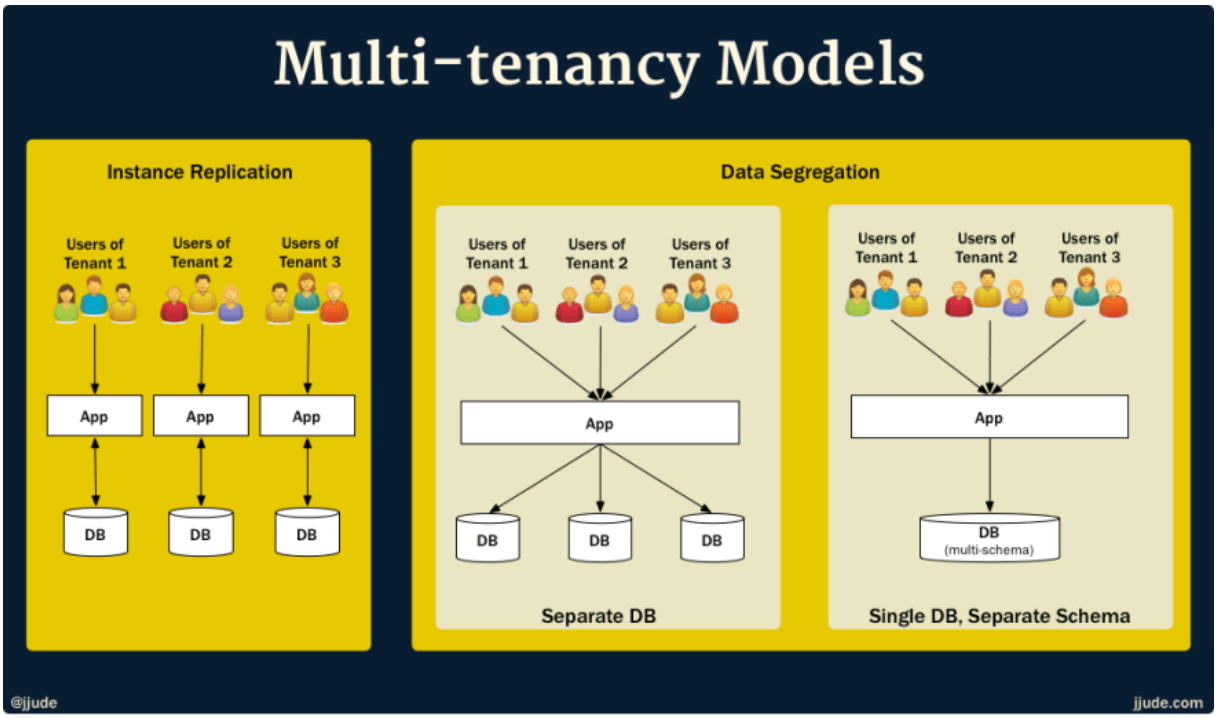
Potrzeba takiej architektury ma następujące uzasadnienie – wyobraźmy sobie witrynę sklepu, wystawioną na serwerze aplikacji pod adresem http://sklep.com. Sklep ma bazę danych towarów, rejestruje użytkowników – klientów, ma użytkowników – administratorów.

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
    }
}
```

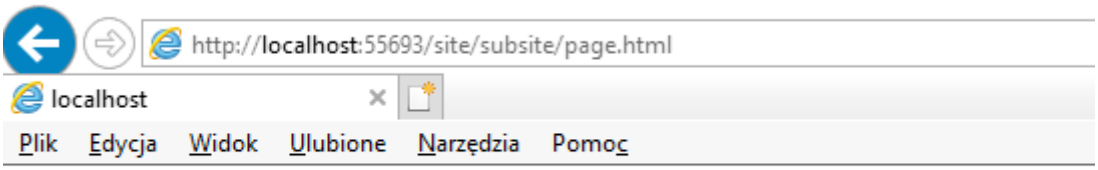
Aby cały przykład zadziałał, niezbędne jest jeszcze skonfigurowanie w **web.config** mapowania modułów ASP.NET dla **wszystkich** ścieżek

```
<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.



tenant: site site: subsite page: page

2 ASP.NET Core

Przez wiele lat od udostępnienia .NET, czyli po 2001 roku, Microsoft próbował doprowadzić do ekspansji .NET na inne platformy systemowe. Częściowym sukcesem była kolaboracja w ramach projektu Mono. Największym organiczeniem Mono było przeciętne wsparcie dla ASP.NET – rozwój wsparcia zatrzymał się na wersji 2.0, liczba wspieranych API była ograniczona, podobnie możliwości hostowania aplikacji.

Pod koniec 2014 ogłoszono, a w 2016 opublikowano, implementację .NET wybudowaną od podstaw w Microsoft na licencji MIT. Tę wersję nazwano .NET Core – nawiązując do założeń:

- Kompatybilność z jak największą liczbą sytemów
- Instalacja wielu wersji środowiska w systemie
- Hostowanie ASP.NET zarówno w ramach IIS jak i samodzielne (dzięki czemu możliwe jest hostowanie w dowolnym wspieranym systemie oraz w kontenerach)

Wiele decyzji projektowych .NET Core sugeruje, że inspirację czerpano nie tylko w .NET Framework ale też od innych środowisk uruchomieniowych, na przykład node.js. Podobieństw jest wiele, m.in. konwencja użycia zufinikowanego interfejsu dostępu do aplikacji (tworzenie, kompilacja, uruchamianie):

- polecenia `node`/`npm` vs `dotnet`),
- użycie libuv do obsługi asynchronicznego IO (.NET Core dopiero w wersji 6 ostatecznie pozbył się libuv)
- otwarcie się na alternatywne środowiska zintegrowane – np. Visual Studio Code

Istnienie równolegle dwóch implementacji środowiska .NET oznacza problem zgodności API. Problem ten częściowo rozwiązano opracowując specyfikację .NET Standard.

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET and .NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]	N/A
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.1
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	202

Pierwsza wersja .NET Core miała bardzo ograniczony zestaw API (brakowało m.in. podstawowych usług kryptograficznych). Sytuacja zaczęła poprawiać się od wersji 2.0. NET Standard w wersji 2.0 wprowadził

pełną binarną zgodność bibliotek (*.dll), dzięki czemu możliwe stało się użycie w .NET Core wielu istniejących do tej pory dla .NET Framework bibliotek zewnętrznych.

Jako przykład należy przywołać tu sytuację z biblioteką Entity Framework. EF w wersji 6 jest jedną z często używanych technologii dostępu do danych. .NET Core zaproponował własną wersję EF, nazwaną EF Core. Pomimo wielu wysiłków, EF Core nadal (na chwilę obecną) nie uzyskał pełni funkcjonalności EF6 – porównanie a wręcz podstawowe wymagania (typu Entity lazy loading) otrzymywał dopiero w kolejnych aktualizacjach.

Dzięki wyżej wzmiankowanej zgodności, możliwe stało się użycie EF6 w aplikacjach .NET Core.

Istnienie .NET Core oznacza, że programując w ASP.NET należy podjąć decyzję o wyborze środowiska uruchomieniowego. Przygotowano pewne wytyczne, z których najważniejsze to:

- .NET Core ma większą wydajność
- .NET Core może być hostowany w kontenerach (np. Docker)
- W przyszłości nowe podsystemy będą dodawane tylko do .NET Core (gRPC)

Jeśli chodzi o minusy .NET Core należy uczciwie powiedzieć, że .NET Framework od początku ma bardzo stabilne API, aplikacja przygotowana dla .NET 2 prawdopodobnie skompiluje się i zadziała na .NET 4.8.

W przypadku .NET Core, lista zmian między wersjami bywa duża, dotyczy to szczególnie migracji z aplikacji przygotowanych we wczesnych wersjach .NET Core (przykład). Z uwagi na relatywnie krótki czas życia poszczególnych wersji (~3 lata), aplikacja musi nadążać za zmianami i – jeśli kiedykolwiek zdarzy się tak że jakiś ważny podsystem zniknie – to utrzymanie aplikacji może być kłopotliwe.

2.1 Zarządzanie aplikacją – polecenie dotnet

Sprawdzenie wersji środowiska .NET Core

```
dotnet --version
```

Utworzenie nowej aplikacji

```
dotnet new {typ aplikacji}
```

Budowanie aplikacji

```
dotnet build
```

Uruchomienie aplikacji

```
dotnet run
```

W sytuacji kiedy do rozwijania apliakcji używane jest Visual Studio (pełne, nie VSC), te polecenia są wyłącznie ciekawostkami – znacznie wygodniej jest kontrolować proces tworzenia projektu/aplikacji z poziomu VS.

Warto zapoznać się z dokumentacją sposobu określania portu dla aplikacji uruchomionej w systemie – preferowany sposób to plik **launchSettings.json**, ale ten plik jest używany tylko przy uruchamianiu z poziomu Visual Studio. Aplikacja uruchamiana bezpośrednio z poziomu powłoki systemu domyślnie wiąże się na port 5000 (można to zmienić).

3 Elementy infrastruktury ASP.NET Core

Od strony środowiska obsługującego aplikacje internetowe (żądanie/odpowiedź), ASP.NET Core wykazuje bardzo duże podobieństwo do ASP.NET. W szczególności, programując przykładowo przy użyciu podsystemu MVC można z trudem zauważać różnice. Obiekty **Request/Response/Context** zachowują się podobnie, kontrolery, widoki, binding, helpery – te wszystkie mechanizmy zachowują się - z dokładnością do szczegółów - tak samo.

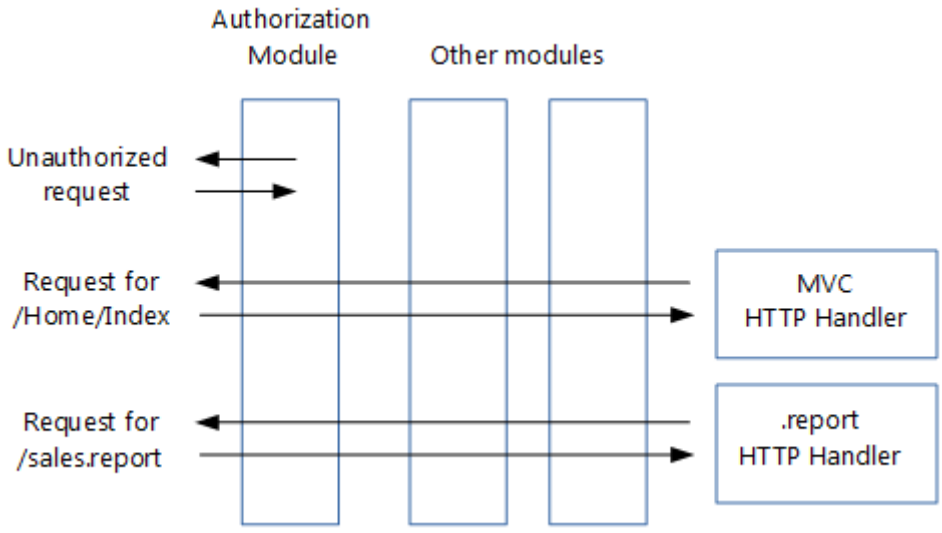
Istnieją jednak dwie fundamentalne różnice w podejściu do samej architektury aplikacji. Te różnice to:

- Użycie **middleware** jako architektury potoku przetwarzania
- Użycie kontenera DI do rozwiązywania zależności do usług

3.1 Moduły+handlery vs Middleware

Potok ASP.NET Framework zbudowany był w oparciu o **moduły** i **handlery** (omówione na jednym z poprzednich wykładów):

- Moduły – miały wiele zdarzeń, w trakcie żądania wykonywane były po kolei metody implementujące zdarzenie w kolejnych modułach (po czym kolejne zdarzenie itd.)
- Handlery – implementowały tworzenie zawartości

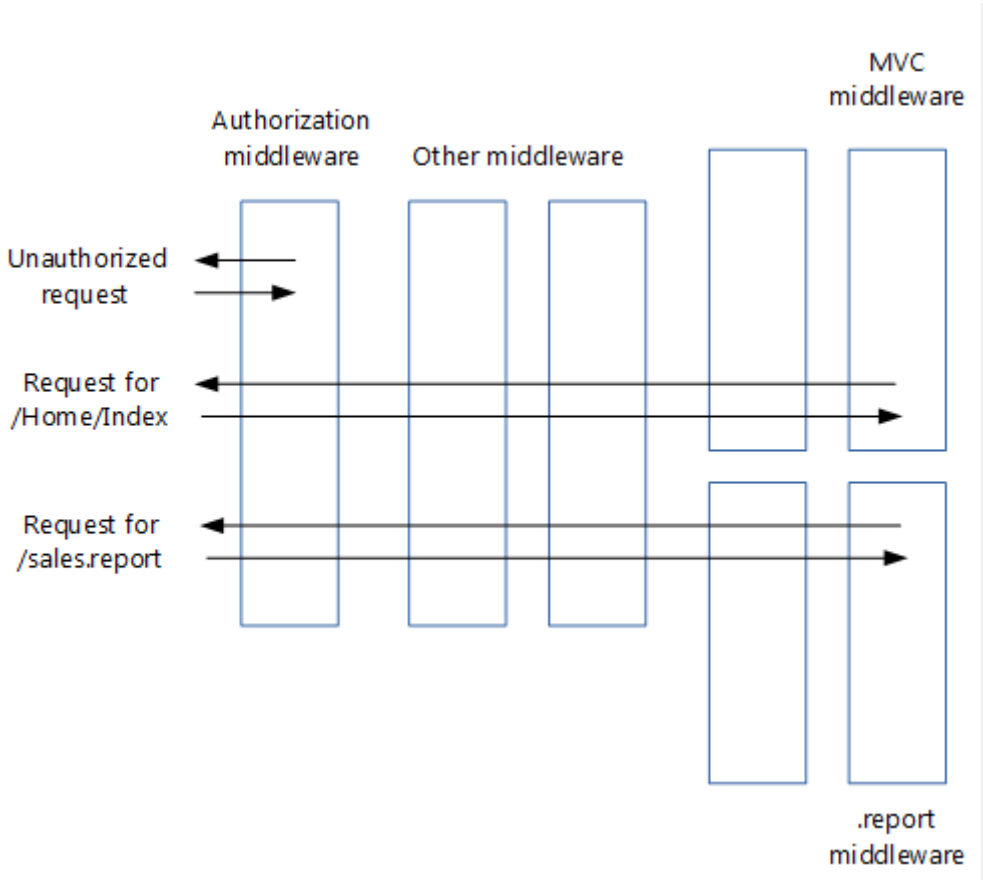


Rysunek 2 Za https://docs.microsoft.com/en-us/aspnet/core/migration/http-modules?view=aspnetcore-6.0

Potok ASP.NET Core oparty jest o tzw. **middleware**. Middleware to klasa implementująca część logiki potoku, która łączy ze sobą funkcjonalność modułu i handlera:

- Middleware może tworzyć zawartość
- Middleware może zawołać inne (kolejne) middleware
- Kolejność middleware jest więc znacząca

Część middleware jest opcjonalna, na przykład aplikacja która nie korzysta z sesji czy autentykacji może w ogóle nie korzystać z tych middleware.



Rysunek 3 (to samo źródło co wyżej)

W praktyce tworzenie własnych middleware, podobnie jak tworzenie własnych modułów czy handlerów, wymagane jest tylko w szczególnych przypadkach.

W najprostszym wariancie, kiedy middleware konfigurowane jest warunkowo (w żargonie .NET Core mówi się że jest „mapowane"), cała aplikacja może wyglądać tak:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

(!!)

Jest to faktycznie cała aplikacja, nie potrzebuje nawet funkcji **Main** (nowsze wersje C# zniosły wymaganie klasy z funkcją Main). Jest to duża zmiana nawet w stosunku do .NET Core 5 i wcześniejszych, dlatego warto zapoznać się z rekomendacjami używania nowszych, zwięzłych konwencji (tzw. **Minimal API**).

W bardziej rozbudowanym wariancie, w potoku mogą pojawiać się albo funkcje middleware:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

```
app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("pierwsze middleware: start\n");
    await next();
    await context.Response.WriteAsync("pierwsze middleware: end\n");
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync("drugie middleware: start\n");
    await context.Response.WriteAsync("drugie middleware: end\n");
});

app.Run();
```

albo taka funkcja może być wyniesiona do osobnej klasy a dodanie middleware do potoku – wyniesione do metody rozszerzającej

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseFirstMiddleware();

app.Run(async (context) =>
{
    await context.Response.WriteAsync("drugie middleware: start\n");
    await context.Response.WriteAsync("drugie middleware: end\n");
});

app.Run();

public class FirstMiddleware
{
    private readonly RequestDelegate _next;

    public FirstMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        await context.Response.WriteAsync("pierwsze middleware: start\n");
        await _next(context);
        await context.Response.WriteAsync("pierwsze middleware: end\n");
    }
}

public static class FirstMiddlewareExtensions
{
    public static IApplicationBuilder UseFirstMiddleware(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<FirstMiddleware>();
    }
}
```

W powyższych przykładach pojawiają się trzy funkcje

- **app.Use(…)** – konfiguruje Middleware które może wywołać kolejne middleware

- **app.Run(…)** – konfiguruje Middleware które nie wywołuje innego middleware (tzw. middleware terminujące)
- **app.Run()** – blokujące przejście w tryb nasłuchu

3.2 Kontener usług

Proszę przypomnieć sobie przykład jednej z zaawansowanych technik MVC, w której domyślna fabryka kontrolerów była zamieniona na fabrykę używającą kontenera DI. W tamtym przykładzie można było w ten sposób do instancji kontrolera „wstrzykiwać" zależności do zewnętrznych usług.

Ten pomysł został „wbudowany" w ASP.NET Core – infrastruktura korzysta z kontenera przy rozwiazywaniu zależności do zewnętrznych usług. Kontener ma zasadniczo trzy polityki czasu życia:

- **AddSingleton** – do usług o jednym wystąpieniu per cała aplikacja
- **AddScoped** – do usług żyjących tyle ile pojedyncze żądanie
- **AddTransient** – do usług które za każdym razem tworzą nową instancję obiektu usługi

Ponieważ usługi są rozwiązywane przez kontener, to mogą być od siebie zależne (na przykład usługa **Foo** korzysta z usługi **Bar**).

Poniższy przykład to kontynuacja poprzedniego, tyle że middleware dostaje wstrzykniętą instancję usługi. W przypadku middleware wstrzykiwanie jest możliwe przez parametr metody **Invoke.**

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IExampleService, ExampleService>();

var app = builder.Build();

app.UseFirstMiddleware();

app.Run(async (context) =>
{
    await context.Response.WriteAsync("drugie middleware: start\n");
    await context.Response.WriteAsync("drugie middleware: end\n");
});

app.Run();

public class FirstMiddleware
{
    private RequestDelegate _next;

    public FirstMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context, IExampleService service)
    {
        await context.Response.WriteAsync($"{service.DoWork()}\n");
        await context.Response.WriteAsync("pierwsze middleware: start\n");
        await _next(context);
        await context.Response.WriteAsync("pierwsze middleware: end\n");
    }
}
```

```
public static class FirstMiddlewareExtensions
{
    public static IApplicationBuilder UseFirstMiddleware(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<FirstMiddleware>();
    }
}

public interface IExampleService
{
    string DoWork();
}

public class ExampleService : IExampleService, IDisposable
{
    public string DoWork()
    {
        return "ExampleService::DoWork";
    }

    public void Dispose()
    {

    }
}
```

Wstrzykiwanie usług działa w wielu miejscach infrastruktury. Można wstrzykiwać zależności do klas middleware, do kontrolerów MVC itd. Jeżeli jakiś element infrastruktury nie obsługuje wprost wstrzykiwania zależności, można poszukać dostawcy usług (czyli kontenera) – w obiekcie **HttpContext** właściwość **RequestServices** wskazuje na kontener, którego metodą **GetService** można uzyskać instancję skonfigurowanej usługi.

Taka architektura ma kilka zalet, m.in.:

- Łatwiejsze testowanie – przykładowo, żeby w jakiejś usłudze dostać bieżący kontekst wywołania (**HttpContext**), w ASP.NET Framework można było użyć statycznej własności **HttpContext.Current**. To oznaczało jednak że testowanie usług w oderwaniu od całej infrastruktury było utrudnione. W ASP.NET Core istnieje dostawca usługi **IHttpContextAccessor** podający bieżący kontekst. Jego użycie wymaga jedynie zarejestrowanie dostawcy (**services.AddHttpContextAccessor()**). W teście jednostkowym można łatwiej dostarczyć zastępczej implementacji tego interfejsu. Podobna sprawa dotyczy interfejsu **IConfiguration** za pomocą którego usługa uzyskuje dostęp do konfiguracji aplikacji
- Czas życia **Scoped** powoduje że na końcu potoku przetwarzania dla usług implementujących interfejs **IDisposable** metoda **Dispose** będzie wołana automatycznie. Pozwala to na przykład na jeszcze inny sposób obsługi stosu aplikacyjnego korzystającego z bazy danych niż przykład wzorcowego stosu jaki widzieliśmy na wykładzie z ASP.NET Framework. W tym innym sposobie, usługa dostępu do bazy danych lub nawet – wprost kontekst Entity Framework dostępu do danych są rejestrowane jako usługi

Jedyną wadą takiej architektury jaką można wskazać jest to, że ten sposób aktywacji usług należy po prostu zrozumieć żeby w pełni z niego korzystać.

Przykład usługi dostępu do danych rejestrowanej jako interfejs mapowany na implementację (używa biblioteki Dapper):

```
[Table("Person")]
public class Person
{
    [Key]
    public int ID { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
}

public interface IDapperRepository<T>
{
    IEnumerable<T> Get(string query, object parameters);
    int? Insert(T t);
    int Update(T t);
    int Delete(T t);
}

public class DapperRepository<T> : IDapperRepository<T>, IDisposable
{
    IConfiguration _configuration;
    SqlConnection  _connection;

    public DapperRepository( IConfiguration configuration )
    {
        _configuration = configuration;

        var connectionString = configuration["AppSettings:ConnectionString"];
        _connection = new SqlConnection(connectionString);
    }

    public int Delete(T t)
    {
        return this._connection.Delete(t);
    }

    public void Dispose()
    {
        _connection.Dispose();
    }

    public IEnumerable<T> Get(string query, object parameters)
    {
        return this._connection.Query<T>(query, parameters);
    }

    public int? Insert(T t)
    {
        return this._connection.Insert(t);
    }

    public int Update(T t)
    {
        return this._connection.Update(t);
    }
}
```

```
...

services.AddScoped<IDapperRepository<Person>, DapperRepository<Person>>();
```

Przykład usługi w której rejestracja dotyczy konkretnej klasy, w tym przypadku EF6

```
public class PersonDbContext : DbContext
{
    public PersonDbContext(string connectionString) : base(connectionString) { }
    public DbSet<Person> Persons { get; set; }
}

...

services.AddScoped<PersonDbContext>(services =>
{
    var configuration = services.GetService<IConfiguration>();
    var connectionString = configuration["AppSettings:ConnectionString"];
    return new PersonDbContext(connectionString);
});
```

Proszę zwrócić uwagę na to jak w obu przypadkach pozyskuje się informacje z konfiguracji aplikacji

- Dla usługi w której mamy kontrolę nad klasą implementującą – można dodać jej zależność do kolejnej usługi (**IConfiguration**)
- Dla usługi w której nie mamy kontroli nad klasą implementującą (**DbContext**) – można samą rejestrację wykonać za pomocą *metody fabrykującej*

3.3 Konfiguracja

W aplikacjach .NET Framework parametry konfiguracyjne aplikacji umieszczamy w **web.config**, w sekcjach **appSettings** lub w **connectionStrings** lub nawet tworzymy własne sekcje konfiguracyjne. Z kolei do danych z konfiguracji dostajemy się za pomocą klasy **ConfigurationManager.**

W przypadku .NET Core, konfiguracja rozpoczyna się od określenia gdzie jest przechowywana. Domyślnie jest to plik **appsettings.json**, ale na starcie aplikacji można określić więcej źródeł oraz ich formaty.

```
builder.Configuration
    .AddJsonFile( "Configuration\\appSettings.json" );
```

Pierwszy sposób dostępu do konfiguracji to wstrzyknięcie do funkcji obsługującej żądanie obiektu typu **IConfiguration.** Za pomocą jego indeksera lub metody **GetSection** można wydobywać dane z konfiguracji.

Czyli konfiguracji:

```
"foo": "bar",
"foo2": {
  "bar2": 5
}
```

jej odczyt wygląda tak:

```
app.MapGet( "/",
    (IConfiguration config) =>
    {
        // pojedyncze ustawienie
        var fooSetting = config["foo"];
        var barSetting = config["foo2:bar2"];

        ...
    } );
```

Jeśli konfiguracja zawiera nie proste wartości, ale całe *obiekty*, do których chciałoby się mieć dostęp w całości, to przydatny jest wzorzec **IOptions.**

W poniższym przykładzie chcemy w .NET Core uzyskać podobny efekt jak **appSettings** w .NET Framework – listę par klucz-wartość.

Zaczynamy od samych danych w konfiguracji

```
"AppSettings": {
  "Settings": [
    {
      "Key": "Foo1",
      "Value": "Bar1"
    },
    {
      "Key": "Foo2",
      "Value": "Bar2"
    }
  ]
},
```

Potem definiujemy model

```
public class AppSettingsConfig
{
    public KeyValueConfig[] Settings { get; set; }
}

public class KeyValueConfig
{
    public string Key { get; set; }
    public string Value { get; set; }
}
```

a następnie odczytujemy dane.

Jest na to kilka sposobów.

Za pomocą **GetSection** dostaniemy model sekcji konfiguracyjnej, nie nasz wyżej przygotowany. To mało wygodne.

Za pomocą **GetSection** ale na nim jeszcze wywołane **Get** pozwala już powiązać model sekcji konfiguracyjnej z własnym:

```
app.MapGet( "/",
    (IConfiguration config) =>
    {
        var settings   = config.GetSection("AppSettings");

        // lista działa od razu czytana do listy
        var list = config.GetSection("AppSettings:Settings").Get<KeyValueCo
nfig[]>();

        // a do obiektu?
        var list2 = config.GetSection("AppSettings").Get<AppSettingsConfig>
();

    } );
```

To dość żmudne – stąd użyjmy wzorca IOptions. Wymaga on reguły wiązania:

```
builder.Configuration
    .AddJsonFile( "Configuration\\appSettings.json" );

builder.Services.AddOptions<AppSettingsConfig>().Bind( builder.Configuratio
n.GetSection( "AppSettings" ) );
```

ale dzięki niej do metody wstrzykiwany jest **IOptions<T>**

```
app.MapGet( "/",
    (IOptions<AppSettingsConfig> appSettings) =>
    {
        // obiekt typu AppSettingsConfig jest w .Value
        var list = appSettings.Value;

        . . .
    } );
```

3.4 Routing w .NET Core

W .NET Core pojawia się pojęcie tzw. **Endpointu** – punktu końcowego (adresu) który jest „adresowalny" z punktu widzenia klienta oraz powoduje zwrócenie wyniku.

Istnienie punktów końcowych ułatwia zarządzanie tym co ostatecznie „wykona" się i zostanie zwrócone jako wynik żądania – silnik ma jawną listę punktów końcowych z przypisaną do nich logiką konstruowania wyniku.

Zasady routingu .NET Core są opisane szczegółowo w dokumentacji i z uwagi na ramy tych notatek, nie będą tu powielane.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// Location 1: before routing runs, endpoint is always null here.
// This will run for any endpoint. "/" or "/abc" or "/def" or whatever.
// It won't have any endpoint because it's placed before app.UseRouting()
app.Use( async ( context, next ) =>
{
    Console.WriteLine( $"1. Endpoint: {context.GetEndpoint()?.DisplayName ?
? "(null)"}" );
    await next( context );
} );

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-
null if routing found a match.
// This will run for any endpoint. "/" or "/abc" or "/def" or whatever.
// It will have an endpoint only for "/" because there's a match for "/" be
low (app.MapGet("/"...)
app.Use( async ( context, next ) =>
{
    Console.WriteLine( $"2. Endpoint: {context.GetEndpoint()?.DisplayName ?
? "(null)"}" );
    await next( context );
} );

// Location 3a: runs when this endpoint matches
// This will run only for "/"
// Will have an endpoint (Of course! Duh!)
app.MapGet( "/", ( HttpContext context ) =>
{
    Console.WriteLine( $"3a. Endpoint: {context.GetEndpoint()?.DisplayName
?? "(null)"}" );
    return "Hello World!";
} ).WithDisplayName( "Hello" );

// Location 3b: runs when this endpoint matches
app.MapGet( "/foo/{id?}", ( HttpContext context, string id ) =>
{
    Console.WriteLine( $"3b. id={id}, Endpoint: {context.GetEndpoint()?.Dis
playName ?? "(null)"}" );
    return "Hello World!";
} );
```

```
app.UseEndpoints( endpoints =>
{
    // Location 3c: old-style endpoint
    endpoints.Map( "/bar",
        async context =>
        {
            Console.WriteLine( $"3c. Endpoint: {context.GetEndpoint()?.Disp
layName ?? "(null)"}" );
            await context.Response.WriteAsync( $"3c. Endpoint: {context.Get
Endpoint()?.DisplayName ?? "(null)"}" );
        }
    );
} );

app.Map( "/qux", app =>
{
    // Location 3d: Map without endpoint
    app.Run(
        async context =>
        {
            Console.WriteLine( $"3d. Endpoint: {context.GetEndpoint()?.Disp
layName ?? "(null)"}" );
            await context.Response.WriteAsync( $"3d. Endpoint: {context.Get
Endpoint()?.DisplayName ?? "(null)"}" );
        }
    );
} );

//UseEndpoints middleware is terminal when a match is found
//The middleware after UseEndpoints execute only when no match is found
app.UseEndpoints( _ => { } );

// Location 4: runs after UseEndpoints - will only run if there was no matc
h.
// This will run for any endpoint except "/". For eg: "/abc" or "/def" or w
hatever.
app.Use( async ( context, next ) =>
{
    Console.WriteLine( $"4. Endpoint: {context.GetEndpoint()?.DisplayName ?
? "(null)"}" );
    await next( context );
} );

app.Run();
```

4 ASP.NET MVC

4.1 MVC NET.Framework

Aby skonfigurować podsystem MVC należy zarejestrować routę i handler w potoku przetwarzania

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Start", action = "Index", id = UrlParameter
.Optional }
);
```

4.2 MVC Core

Aby skonfigurować podsystem MVC należy:

- Zarejestrować usługi

```
services.AddControllersWithViews();
```

- Dodać endpoint

```
app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
```