

عنوان: خودمیزبانی ماژول های Nancy

نویسنده: سلمان عرب عامری

تاریخ: ۱۱:۵ ۱۳۹۱/۰۴/۱۴

آدرس: www.dotnettips.info

برچسب‌ها: C#, Nancy, Self-Hosting

در ادامه بررسی پروژه Nancy، در این مطلب به میزبانی پروژه‌های Nancy بدون نیاز به Asp.net می‌پردازیم. به این معنی که برنامه اجرایی که شما می‌نویسید خود یک سرور ایجاد می‌کند و کاربر با وارد کردن آدرس دستگاه شما در مرورگر خود، صفحات و ماژول‌های طراحی شده توسط شما را مشاهده می‌کند.

از کاربردهای چنین سیستمی به سایت‌های قابل حمل، و یا ارائه خدمات یک نرم افزار بر روی صفحات html می‌توان اشاره کرد. مثل گوگل دسکتاپ و یا گزارشات برخی سرویس‌های ویندوزی و یا حتی تنظیم یک سخت افزار متصل به سیستم از روی شبکه. یک ایده جالب می‌تواند ارسال اس ام اس از طریق شبکه و با جی اس ام مودم باشد. که به عنوان مثال کاربران با ورود به یک صفحه و ثبت پیام بتوانند از طریق جی اس ام مودم متصل به سرور آن را ارسال کنند. با یک مثال ساده ادامه می‌دهیم.

برای شروع یک پروژه از نوع Console بسازید و در Package manager کتابخانه Nancy.Hosting.Self را نصب کنید. حالا یک ماژول جدید به نام TestModule.cs به پروژه اضافه می‌کنیم.

```
public class TestModule:NancyModule
{
    public TestModule()
    {
        Get["/"] = x=> { return "It is a test for nancy self hosting."; };
    }
}
```

حالا وارد program.cs شده و در متد Main کد زیر را می‌نویسیم:

```
var selfHost = new NancyHost(new Uri("http://localhost:12345"));
selfHost.Start();
Console.ReadKey();
selfHost.Stop();
```

در خط اول پورتهای که منتظر دریافت درخواست‌های کاربران است را برابر 12345 قرار می‌دهیم. بنابراین برای تست این کد باید در مرورگر آدرس

<http://localhost:12345> را تایپ کنید. اگر بخواهیم کاربر عدد انتهایی را وارد نکند باید از پورت 80 استفاده کنیم که پیش فرض http است ولی اکثراً در سیستم برنامه نویسی‌ها توسط IIS مشغول می‌باشد.

در خط بعد سرور را اجرا کرده ایم و برنامه را به حالت انتظار برای فشردن کلیدی در کنسول برده ایم. وقتی کلیدی در کنسول فشرده شود سرور به حالت توقف می‌رود و اجرای برنامه پایان می‌یابد.

Nancy امکانات دیگری هم دارد. به عنوان مثال می‌توان برای طراحی نمای ماژول‌ها از موتورهای دید استفاده کرد (ViewEngines). موتورهای مثل Razor و ... در صورت علاقمندی دوستان، در این باره هم خواهیم نگاشت.

نظرات خوانندگان

نویسنده: شهروز جعفری
تاریخ: ۱۸:۹ ۱۳۹۱/۰۴/۱۶

لطفا ادامه بدید تشکر میکنم

در این مثال برای اینکه Instance Provider سفارشی خود را بتوانیم به عنوان یک Behavior به سرویس اضافه نماییم باید به خاصیت Description.Behaviors شی ServiceHost دسترسی داشته باشیم. زمانی که در پروژه‌های WCF از روش Self Hosting برای هاست سرویس‌ها استفاده کنیم به دلیل دسترسی مستقیم به شی ServiceHost هر گونه تنظیمات و عملیات Customization به راحتی امکان پذیر است؛ اما در IIS Hosting، از آن جا که به صورت پیش فرض از ServiceHostFactory موجود در WCF استفاده می‌شود ما دسترسی به شی ServiceHost نداریم. برای حل این مسئله باید یک CustomServiceHostFactory ایجاد نماییم که به راحتی در WCF این امکان تدارک دیده شده است.

بررسی یک مثال:

ابتدا کلاسی به صورت زیر ایجاد نمایید. در این کلاس می‌توانید کدهای لازم برای سفارشی کردن شی ServiceHost را قرار دهید:

```
public class CustomServiceHost : ServiceHost
{
    public CustomServiceHost( Type t, params Uri baseAddresses ) :
        base( t, baseAddresses ) {}

    public override void OnOpening()
    {
        this.Description.Add( new MyServiceBehavior() );
    }
}
```

اگر از این به بعد به جای استفاده از ServiceHost مستقیماً از CustomServiceHost استفاده نماییم، MyServiceBehavior به صورت خودکار به عنوان یک ServiceBehavior برای سرویس مورد نظر در نظر گرفته می‌شود. برای این که هنگام هاست سرویس مورد نظر به صورت خودکار از این شی کلاس استفاده شود می‌توان کلاس Factory ساخت سرویس را تغییر داد به صورت زیر:

```
public class CustomServiceHostFactory : ServiceHostFactory
{
    public override ServiceHost CreateServiceHost( Type t, Uri[] baseAddresses )
    {
        return new CustomServiceHost( t, baseAddresses )
    }
}
```

حال بر روی سرویس مورد نظر کلیک راست کرده و گزینه View Markup را انتخاب نمایید، چیزی شبیه به گزینه زیر را مشاهده خواهید کرد:

```
<%@ ServiceHost Language="C#" Debug="true" Service="WcfService1.Service1" CodeBehind="Service1.svc.cs" %>
```

کافیست کلاس CustomServiceHostFactory را به عنوان Factory این سرویس مشخص نماییم. به صورت زیر:

```
<%@ ServiceHost Language="C#" Debug="true" Factory="CustomServiceHostFactory"
Service="WcfService1.Service1" CodeBehind="Service1.svc.cs" %>
```

از این به بعد عملیات وهله سازی از سرویس بر اساس تنظیمات پیش فرض صورت گرفته در این کلاس‌ها انجام می‌گیرد.

زمانیکه از Template های پیش فرض تدارک دیده شده در VS.Net برای اپلیکیشن های وب خود استفاده می کنید، وب اپلیکیشن و سرور با هم یکپارچه هستند و تحت IIS اجرا می شوند. به وسیله [Owin](#) می توان این دو مورد را بدون وابستگی به IIS به صورت مجزا اجرا کرد. در این پست قصد داریم سرویس های Web Api را در قالب یک Windows Service با استفاده از کتابخانه ی [TopShelf](#) هاست نماییم.

پیش نیاز ها:

« [Owin چیست](#) »

« [تبدیل برنامه های کنسول ویندوز به سرویس ویندوز ان تی](#) »

برای شروع یک برنامه Console Application ایجاد کرده و اقدام به نصب پکیج های زیر نمایید:

```
Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
Install-Package TopShelf
```

حال یک کلاس Startup برای پیاده سازی Configuration های مورد نیاز ایجاد می کنیم
در این قسمت می توانید تنظیمات زیر را پیاده سازی نمایید:

«سیستم Routing:

«تنظیم Dependency Resolver برای تزریق وابستگی کنترلرهای Web Api:

«تنظیمات hub های SignalR (در حال حاضر SignalR به صورت پیش فرض نیاز به Owin برای اجرا دارد):

«رجیستر کردن Owin Middleware های نوشته شده:

«تغییر در Asp.Net Pipeline:

«و...»

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        HttpConfiguration config = new HttpConfiguration();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
        appBuilder.UseWebApi(config);
    }
}
```

* به صورت پیش فرض نام این کلاس باید Startup و نام متد آن نیز باید Configuration باشد.

در این مرحله یک کنترلر Api به صورت زیر به پروژه اضافه نمایید:

```
public class ValuesController : ApiController
{
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    public string Get(int id)
    {
    }
```

```

        return "value";
    }

    public void Post([FromBody]string value)
    {
    }

    public void Put(int id, [FromBody]string value)
    {
    }
}

```

کلاسی به نام ServiceHost ایجاد نمایید و کدهای زیر را در آن کپی کنید:

```

public class ServiceHost
{
    private IDisposable webApp;

    public static string BaseAddress
    {
        get
        {
            return "http://localhost:8000/";
        }
    }

    public void Start()
    {
        webApp = WebApp.Start<Startup>(BaseAddress);
    }

    public void Stop()
    {
        webApp.Dispose();
    }
}

```

واضح است که متد Start در کلاس بالا با استفاده از متد Start کلاس WebApp، سرویس های Web Api را در آدرس مورد نظر هاست خواهد کرد. با فراخوانی متد Stop این سرویس ها نیز dispose خواهند شد. در مرحله آخر باید شروع و توقف سرویس ها را تحت کنترل کلاس HostFactory کتابخانه TopShelf در آوریم. برای این کار کفایست کلاسی به نام ServiceHostFactory ایجاد کرده و کدهای زیر را در آن کپی نمایید:

```

public class ServiceHostFactory
{
    public static void Run()
    {
        HostFactory.Run( config =>
        {
            config.SetServiceName( "ApiServices" );
            config.SetDisplayName( "Api Services" );
            config.SetDescription( "No Description" );

            config.RunAsLocalService();

            config.Service<ServiceHost>( cfg =>
            {
                cfg.ConstructUsing( builder => new ServiceHost() );

                cfg.WhenStarted( service => service.Start() );
                cfg.WhenStopped( service => service.Stop() );
            } );
        } );
    }
}

```

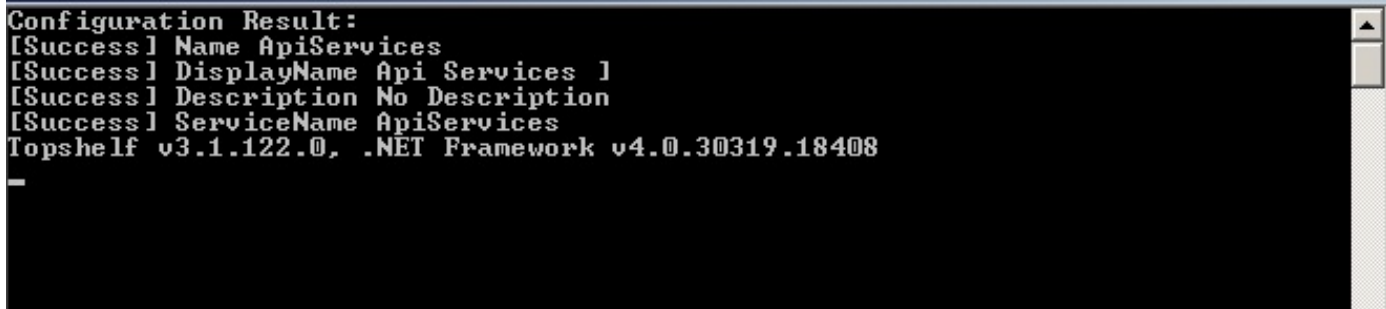
توضیح کدهای بالا:

ابتدا با فراخوانی متد Run سرویس مورد نظر اجرا خواهد شد. تنظیمات نام سرویس و نام مورد نظر جهت نمایش و همچنین توضیحات در این قسمت انجام می گیرد.

با استفاده از متد ConstructUsing عملیات و هله سازی از سرویس انجام خواهد گرفت. در پایان نیز متد Start و Stop کلاس ServiceHost، به عنوان عملیات شروع و پایان سرویس ویندوز مورد نظر تعیین شد.

حال اگر در فایل Program پروژه، دستور زیر را فراخوانی کرده و برنامه را ایجاد کنید خروجی زیر قابل مشاهده است.

```
ServiceHostFactory.Run();
```



```
Configuration Result:
[Success] Name ApiService
[Success] DisplayName Api Services
[Success] Description No Description
[Success] ServiceName ApiService
Topshelf v3.1.122.0, .NET Framework v4.0.30319.18408
```

در حالیکه سرویس مورد نظر در حال اجراست، Browser را گشوده و آدرس `http://localhost:8000/api/values/get` را در AddressBar وارد کنید. خروجی زیر را مشاهده خواهید کرد:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>value1</string>
  <string>value2</string>
</ArrayOfstring>
```

در بسیاری از سناریوها این موضوع مطرح می شود که سرویس های طراحی شده بر اساس Asp.Net Web Api، فقط به یک سری آی پی های مشخص سرویس دهند. برای مثال اگر Ip کلاینت در لیست کلاینت های دارای لایسنس خریداری شده بود، امکان استفاده از سرویس میسر باشد؛ در غیر این صورت خیر. بسته به نوع پیاده سازی سرویس های Web api، پیاده سازی این بخش کمی متفاوت خواهد شد. در طی این پست این موضوع را برای سه حالت IIS Host و SelfHost و Owin Host بررسی می کنیم. در اینجا قصد داریم حالتی را پیاده سازی نماییم که اگر درخواست جاری از سوی کلاینتی بود که Ip آن در لیست Ip های غیر مجاز قرار داشت، ادامه ی عملیات متوقف شود.

IIS Hosting

حالت پیش فرض استفاده از سرویس های Web Api همین گزینه است؛ وابستگی مستقیم به System.Web. در مورد مزایا و معایب آن بحث نمی کنیم اما اگر این روش را انتخاب کردید تکه کد زیر این کار را برای ما انجام می دهد:

```
if (request.Properties.ContainsKey["MS_HttpContext"])
{
    var ctx = request.Properties["MS_HttpContext"] as HttpContextWrapper;
    if (ctx != null)
    {
        var ip = ctx.Request.UserHostAddress;
    }
}
```

برای بدست آوردن شی HttpContext می توان آن را از لیست Properties های درخواست جاری به دست آورد. حال کد بالا را در قالب یک Extension Method در خواهیم آورد؛ به صورت زیر:

```
public static class HttpRequestMessageExtensions
{
    private const string HttpContext = "MS_HttpContext";

    public static string GetClientIpAddress(this HttpRequestMessage request)
    {
        if (request.Properties.ContainsKey(HttpContext))
        {
            dynamic ctx = request.Properties[HttpContext];
            if (ctx != null)
            {
                return ctx.Request.UserHostAddress;
            }
        }
        return null;
    }
}
```

Self Hosting

در حالت Self Host می توان عملیات بالا را با استفاده از خاصیت [RemoteEndpointMessageProperty](#) انجام داد که تقریباً شبیه به حالت Web Host است. مقدار این خاصیت نیز در شی جاری *HttpRequestMessage* وجود دارد. فقط باید به صورت زیر آن را واکنشی نماییم:

```
if (request.Properties.ContainsKey[RemoteEndpointMessageProperty.Name])
{
    var remote = request.Properties[RemoteEndpointMessageProperty.Name] as RemoteEndpointMessageProperty;
```

```

    if (remote != null)
    {
        var ip = remote.Address;
    }
}

```

خاصیت [RemoteEndpointMessageProperty](#) به تمامی درخواست ها وارده در سرویس های WCF چه در حالت استفاده از Http و چه در حالت Tcp اضافه می شود و در اسمبلی System.ServiceModel نیز می باشد. از آنجا که Web Api از هسته ای WCF استفاده می کند (WCF Core) در نتیجه می توان از این روش استفاده نمود. فقط باید اسمبلی System.ServiceModel را به پروژه ای خود اضافه نمایید.

ترکیب حالت های قبلی:

اگر می خواهید کدهای نوشته شده شما وابستگی به نوع هاست پروژه نداشته باشد، یا به معنای دیگر، در هر دو حالت به درستی کار کند می توانید به روش زیر حالت های قبلی را با هم ترکیب کنید.
«در این صورت دیگر نیازی به اضافه کردن اسمبلی System.ServiceModel نیست.»

```

public static class HttpRequestMessageExtensions
{
    private const string HttpContext = "MS_HttpContext";
    private const string RemoteEndpointMessage =
        "System.ServiceModel.Channels.RemoteEndpointMessageProperty";

    public static string GetClientIpAddress(this HttpRequestMessage request)
    {
        if (request.Properties.ContainsKey(HttpContext))
        {
            dynamic ctx = request.Properties[HttpContext];
            if (ctx != null)
            {
                return ctx.Request.UserHostAddress;
            }
        }

        if (request.Properties.ContainsKey(RemoteEndpointMessage))
        {
            dynamic remoteEndpoint = request.Properties[RemoteEndpointMessage];
            if (remoteEndpoint != null)
            {
                return remoteEndpoint.Address;
            }
        }

        return null;
    }
}

```

مرحله بعدی طراحی یک DelegatingHandler جهت استفاده از IP به دست آمده است .

```

public class MyHandler : DelegatingHandler
{
    private readonly HashSet<string> deniedIps;

    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (deniedIps.Contains(request.GetClientIpAddress()))
        {
            return Task.FromResult( new HttpResponseMessage( HttpStatusCode.Unauthorized ) );
        }

        return base.SendAsync(request, cancellationToken);
    }
}

```


: Owin

زمانی که از [Owin برای هاست سرویس های Web Api](#) خود استفاده می کنید کمی روال انجام کار متفاوت خواهد شد. در این مورد نیز می توانید از DelegatingHandler ها استفاده کنید. معرفی DelegatingHandler طراحی شده به Asp.Net PipeLine به صورت زیر خواهد بود:

```
public class Startup
{
    public void Configuration( IApplicationBuilder appBuilder )
    {
        var config = new HttpConfiguration();

        var routeHandler = HttpClientFactory.CreatePipeline( new HttpControllerDispatcher( config
        ), new DelegatingHandler[]
        {
            new MyHandler(),
        } );

        config.Routes.MapHttpRoute(
            name: "Default",
            routeTemplate: "{controller}/{action}",
            defaults: null,
            constraints: null,
            handler: routeHandler
        );

        config.EnsureInitialized();

        appBuilder.UseWebApi( config );
    }
}
```

اما نکته ای را که باید به آن دقت داشت، این است که یکی از مزایای استفاده از Owin، یکپارچه سازی عملیات هاستینگ قسمت های مختلف برنامه است. برای مثال ممکن است قصد داشته باشید که بخش هایی که با Asp.Net SignalR نیز پیاده سازی شده اند، قابلیت استفاده از کدهای بالا را داشته باشند. در این صورت بهتر است کل عملیات بالا در قالب یک Owin Middleware عمل نماید تا تمام قسمت های هاست شده ی برنامه از کدهای بالا استفاده نمایند؛ به صورت زیر:

```
public class IpMiddleware : OwinMiddleware
{
    private readonly HashSet<string> _deniedIps;

    public IpMiddleware(OwinMiddleware next, HashSet<string> deniedIps) :
        base(next)
    {
        _deniedIps = deniedIps;
    }

    public override async Task Invoke(OwinRequest request, OwinResponse response)
    {
        var ipAddress = (string)request.Environment["server.RemoteIpAddress"];

        if (_deniedIps.Contains(ipAddress))
        {
            response.StatusCode = 403;
            return;
        }

        await Next.Invoke(request, response);
    }
}
```

برای نوشتن یک Owin Middleware کافیست کلاس مورد نظر از کلاس OwinMiddleware ارث ببرد و متد Invoke را Override کنید. لیست Ip های غیر مجاز، از طریق سازنده در اختیار Middleware قرار می گیرد. اگر درخواست مجاز بود از طریق دستور Next.Invoke(request,response) کنترل برنامه به مرحله بعدی منتقل می شود در غیر صورت عملیات با کد 403 متوقف می شود. در نهایت برای معرفی این Middleware طراحی شده به Application، مراحل زیر را انجام دهید.

```
public class Startup
{
    public void Configuration( IApplicationBuilder appBuilder )
```

```
{
    var config = new HttpConfiguration();
    var deniedIps = new HashSet<string> {"192.168.0.100", "192.168.0.101"};

    app.Use(typeof(IpMiddleware), deniedIps);
    appBuilder.UseWebApi( config );
}
```