

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 پروژه های WCF از روش Self Hosting برای هاست سرویس ها 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 می شود 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 می توانید 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 می شود. برای این که هنگام هاست سرویس مورد نظر به صورت خودکار از این شی کلاس استفاده شود می توان 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 و هله سازی از سرویس بر اساس تنظیمات پیش فرض صورت گرفته در این کلاس ها انجام می گیرد.

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 );
}
```

## نظرات خوانندگان

نویسنده: امیر بختیاری  
تاریخ: ۱۳۹۳/۰۶/۲۹ ۲۳:۲۹

با سلام؛ مطلب جالب و مفیدی بود فقط برای استفاده از UserHostAddress در یک پروژه در حال استفاده بودم بعد متوجه شدم تمامی لاگ‌ها با یک آی پی ثبت می‌شود بعد از جستجو فهمیدم که تمام درخواست‌ها از یک فایروال عبور می‌کند و تمام آی پی‌ها یکی می‌شود. به جاش از

```
Request.ServerVariables["HTTP_X_FORWARDED_FOR"]
```

استفاده کردم. البته خالی بودنش رو هم چک کردم و مشکلم حل شد. می‌خواستم بدونم راه حل دیگه ای هم داره یا نه. با تشکر

نویسنده: مسعود پاکدل  
تاریخ: ۱۳۹۳/۰۶/۳۰ ۱۳:۴۴

راه حل شما منطقی و درست است. در حالاتی که برای درخواست‌ها عمل forwarding صورت بگیرد تنها آدرسی که مشاهده خواهید کرد آدرس Proxy Server است. در نتیجه در این حالات مقدار آدرس اصلی در خاصیت **HTTP\_X\_FORWARDED\_FOR** ذخیره خواهد شد. و مقدار خاصیت **REMOTE\_ADDR** برابر با آدرس Proxy Server است. از آن جا که دستور `Request.UserHostAddress` برابر با کد زیر می‌باشد:

```
Request.ServerVariables["REMOTE_ADDR"]
```

دلیل یکی بودن تمام IP ها نیز همین است که شما همیشه آدرس Proxy Server را مشاهده می‌کنید.