عنوان: **Owin چیست ؟ قسمت اول**
نویسنده: یاسر مرادی
تاریخ: ۱۳۹۲/۱۲/۰۲ ۱۸:۴۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, IIS, OWIN, Katana, helios

مطمئنا اکثر شما برنامه نویسان با معماری IIS و ASP.NET کمابیش آشنایی دارید
Request از سمت کلاینت به IIS ارسال می‌شود، و **عموما** بسته به نوع درخواست کلاینت یا به یک Static File مپ می‌شود(مثلا به یک عکس)، و یا به یک ISAPI
ISAPI کدی است که عموما با C++ نوشته می‌شود، و برای درخواست آمده از سمت کلاینت کاری را انجام می‌دهد
یکی از این ISAPIها برای ASP.NET است، که درخواست کلاینت را به یک کد مبتنی بر .NET. مپ می‌کند (به همین علت به آن ASP.NET می‌گویند)
نکته ای که در خطوط فوق به وضوح دیده می‌شود، وابستگی شدید ASP.NET به IIS است
بدیهتا کدی که بر روی بستر ASP.NET نوشته می‌شود نیز وابستگی فوق العاده ای به IIS دارد، که یکی از بدترین نوع این وابستگی‌ها در ASP.NET Web Forms دیده می‌شود.
خب، این مسئله چه مشکلاتی را ایجاد می‌کند ؟
مشکل اول که شاید کمتر به چشم بیاید، بحث کندی اجرای بار اول برنامه‌های ASP.NET است.
اما مشکل دوم عدم توانایی در نوشتن کد برنامه، بدون وابستگی به **وب سرور** (در اینجا IIS) است، که این مشکل دوم روز به روز در حال جدی‌تر شدن است.
این مشکل دوم را برنامه نویسان جاوا سالهاست که با آن درگیرند، نکته این است که بین دو وب سرور در نحوه پردازش یک درخواست کلاینت تفاوت هایی وجود دارد، که بالطبع این تفاوت‌ها در نحوه‌ی اجرای کد بالاخره خودش را جاهایی نشان می‌دهد، این که بگوییم رفتار وب سرورها نباید متفاوت باشد کمی مسخره است، زیرا تفاوت آنها با یکدیگر باعث شده که سرعت یکسان و امکانات یکسانی نداشته باشند و هر کدام برای یک سناریوی خاص مناسب‌تر باشند
این مسئله برای ما نیز روز به روز دارای اهمیت بیشتری می‌شود، دیگر این که Web Server ما فقط IIS صرف باشد، سناریوی متداول در پروژه‌های Enterprise نیست
در چه جاهایی می‌توان یک برنامه را هاست کرد ؟
IIS به همراه ASP.NET
IIS بدون ASP.NET (می‌خواهیم برنامه بر روی IIS هاست شود، ولی کاری با ASP.NET نداریم) CLR AppDomains

و وب سرورهای لینوکسی در صورت اجرای برنامه بر روی Mono
و ...
هم اکنون به میزان زیادی مشکل شفاف شده است، مطابق با معماری فعلی داریم
Request >> IIS >> aspnet_isapi.dll >> System.Web.dll >> Your codes
مشکل دیگری که وجود دارد این است که اگر تیمی بخواهد فریم ورکی برای برنامه نویسان نهایی فراهم کند، باید آنرا بر روی اکثر گزینه‌های هاست موجود سازگار کنید، برای مثال مشاهده می‌کنید که ASP.NET Signal R را هم می‌توان بر روی IIS و ASP.NET هاست کرد و هم بر روی یک App Domain کاملا معمولی و علاوه بر این که تیم SignalR باید این هزینه مضاعف را پرداخت کند، خروجی برای ما نیز چندان خوشایند نیست، برای مثال اجرای همزمان ASP.NET SignalR و ASP.NET Web API اگر چه که بر روی هاستی به غیر از ASP.NET نیز امکان پذیر است، اما متاسفانه به عنوان دو بازیگر جدا از هم کار می‌کنند و عملا تعاملی با یکدیگر ندارند، مگر این که بر روی ASP.NET هاست شوند، و یا بسیاری از امکانات Routing موجود در WCF بر روی بستری غیر از ASP.NET کار نمی‌کند. بدیهی است که این بازار پر آشوب به نفع هیچ کس نیست. و اما راه حل چیست ؟ تعدادی از برنامه نویسان حرفه ای .NET. دور یکدیگر جمع شدند و طی بررسی هایشان به این نتیجه رسیدند که هاست‌های مختلف نقاط اشتراک بسیار زیادی دارند و تفاوت‌ها نباید باعث این میزان مشکل شود.
پس استانداری را طراحی کردند با نام OWIN یا Open Web Interface for .NET
این استاندارد به صورت کاملا ریز به طراحی هر چیزی را که شما به آن فکر کنید پرداخته است، Request, Cookie, Response, Web Sokcet و ...
اما همانطور که از نامش مشخص است این یک استاندارد است و پیاده سازی ندارد، و هر هاستی باید یک بار این استاندارد را بر

روی خود پیاده سازی کند
خبر خوش این است که تا این لحظه اکثر هاست‌های مهم این استاندارد را پیاده سازی کرده اند و یا در دست پیاده سازی دارند
پروژه Helios برای IIS
پروژه Katana برای IIS به در کنار و سازگار با ASP.NET برای پروژه هایی که تا این لحظه از امکانات سطح پایین ASP.NET استفاده زیادی کرده اند و فرصت تغییر ساختاری ندارند
پروژه هایی برای App Domains و ...
مرحله‌ی بعدی این است که فریم ورک‌ها خوشان را با Owin سازگار کنند
معروف‌ترین فریم ورک هایی که تا این لحظه اقدام به انجام این کار کرده اند، عبارتند از:

ASP.NET Web API

ASP.NET MVC

ASP.NET Identity

ASP.NET Signal R (در حال حاضر Signal R فقط بر روی Owin قابلیت استفاده دارد)

بدیهی است که زمانی که پروژه ASP.NET Web API بر روی استاندارد OWIN نوشته می‌شود، دیگر نیازی به تحمل هزینه مضاعف برای سازگاری خود با انواع هاست ها ندارد و این مسئله توسط Helios، Katana و ... انجام شده است، که بالطبع بزرگترین نفع آن برای ما جلوگیری از چند باره کاری توسط تیم Web API و ... است که بالطبع در زمان کمتر امکانات بیشتری را به ما ارائه می‌دهند.
البته واضح است فریم ورک هایی که با کلاینت و درخواست‌ها کاری ندارند، با این مقولات کاری ندارند، پس Entity Framework و ... از این داستان مستثنا هستند. و علاوه بر این فریم ورک هایی با طراحی اشتباه و قدیمی مانند ASP.NET Web Forms به صورت کلی قابلیت سازگار شدن با این استاندارد را ندارند، زیرا کاملا به ASP.NET وابسته هستند
و در نهایت در مرحله‌ی بعدی لازم است شما نیز از فریم ورک هایی استفاده کنید که مبتنی بر OWIN هستند، یعنی برای مثال پروژه بعدی تان را مبتنی بر ASP.NET MVC و ASP.NET Web API و ASP.NET Identity پیاده سازی کنید، در این صورت شما می‌توانید برنامه ای بنویسید که به Web Server هیچ گونه وابستگی ندارد.
به این صورت کد زدن چند مزیت بزرگ دیگر هم دارد که از کم اهمیت‌ترین آنها شروع می‌کنیم:
-1 سرعت بسیار بالاتر برنامه در هاست‌های غیر ASP.NET ای، مانند زمانی که شما از IIS به صورت مستقیم و بدون وابستگی به System.Web.dll استفاده می‌کنید.
توجه کنید که حتی در این حالت هم می‌توانید از ASP.NET Web API و Signal R و Identity استفاده کنید و تا 25% سرعت بیشتری داشته باشید (بسته به سناریو) 2- قابلیت توسعه آسانتر و با قابلیت نگهداری بالاتر پروژه‌های Enterprise، برای مثال در یکی از پروژه‌ها من مجبور بودم از ASP.NET Web API به صورتی استفاده کنم که هم توسط کلاینت JavaScript ای استفاده شود، و هم توسط کدهای Controllerهای MVC (بدون استفاده مستقیم از کد سرویس با رفرنس زدن به سرویس‌ها البته) که خوشبختانه OWIN به خوبی از پس این کار بر آمد، و عملا یک سرویس Web API را هم بر روی IIS هاست کردم و هم داخل یک AppDomain
-3 در چند سال آینده که اکثریت مطلق سایت‌ها از این روش استفاده کنند (شما چه بدانید و چه ندانید اگر در برنامه خودتان از Signal R نسخه 2 دارید استفاده می‌کنید، حتما از OWIN استفاده کرده اید)، مایکروسافت می‌تواند دست به تغییرات اساسی‌تری بزند، برای مثال معماری جدیدی از IIS ارائه دهد که مشکلات ساختاری فراوان فعلی IIS را که از حوصله توضیح این مقاله خارج است را نداشته باشد، و فقط یک پیاده سازی OWIN جدید بر روی آن ارائه دهد و برنامه‌های ما **بدون تغییر** بر روی آن نیز کار کنند، و یا این که بتواند تعدادی از فریم ورک‌های با طراحی قدیمی را راحت‌تر از دور خارج کند، مانند Web Forms
نکته پایانی، اگر هم اکنون پروژه ای دارید که در داخل آن از ASP.NET استفاده شده، و برای مثال تعدای فرم ASP.NET Web Forms نیز دارد، نگران نباشید، کماکان می‌توانید از Owin برای سایر قسمت‌ها مانند Web API استفاده کنید، البته در این حالت تاثیری در بهبود سرعت اجرای برنامه مشاهده نخواهید کرد، اما برای مهاجرت و اعمال تغییرات این آسانترین روش ممکن است در قسمت بعدی، مثالی را شروع می‌کنیم مبتنی بر ASP.NET Web API، ASP.NET Identity و Helios

نظرات خوانندگان

نویسنده: ناظم
تاریخ: ۱۳۹۲/۱۲/۰۳ ۱۰:۵۱

سلام
ممنون بابت مطلب مفیدتون.

بدون وابستگی به IIS یعنی هر web server ی که OWIN را پیاده سازی کند امکان اجرای برنامه هایی که مثلا با asp.net mvc نوشته شدن رو خواهند داشت؟
همین که مثلا با asp.net mvc برنامه نوشته شده به معنی این هست که برنامه بر اساس استاندارد OWIN هست؟ یا کارهایی برای این منظور باید انجام داد؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۲/۰۳ ۱۱:۴۸

بدون وابستگی به IIS بعنی شما امکان هاست کردن سرویس‌های Web APi رو به صورت Windows Service یا پروژه Console هم خواهید داشت.
به صورت پیش فرض یک پروژه MVC بدون وابستگی به Owin پیاده سازی می‌شود و برای این منظور می‌توانید یکی از موارد زیر را انجام دهید:
»امکان هاست سرویس‌ها روی IIS. در این صورت Owin فقط به صورت یک Middleware عمل خواهد نمود و در این حالت دیگر نیاز به نوشتن HttpModuleها نخواهید داشت. البته این روش به System.Web وابستگی دارد(**Microsoft.Owin.Host.SystemWeb**)
»استفاده از OwinHost.Exe که در واقع یک پیاده سازی دیگر برای Owin است و عملیات bootstrapping را بر عهده خواهد داشت. در نتیجه شما فقط موارد مربوط به middleWare در application انجام خواهید داد.
»استفاده از Owin Self Hosting برای هاست سرویس‌ها در قالب برنامه Console یا Windows Service
(**Microsoft.Owin.Host.HttpListener**)

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۲/۱۲/۰۳ ۱۲:۱۳

بله، به همین معنی است
البته دقت کنید، پیاده سازی OWIN کار ساده ای نیست، و به سرعت نمی‌توان شاهد پیاده سازی آن بر روی هاست‌های مختلف بود، و این پروسه با سرعت فعلی از نظر من مدتی طول خواهد کشید.
برای مثال Katana که یک پیاده سازی قابل استفاده و خوب از آن به شمار می‌رود کار شرکت مایکروسافت است و سایر پیاده سازی Open Source سایر تیم‌ها که بالطبع امکان مانور شرکت مایکروسافت را ندارند، کمی طول می‌کشد تا واقعا آماده استفاده شود.
و همچنین پیاده سازی‌های فعلی در قسمت هایی مانند Web Socketها و سایر مسائل پیچیده دارای ضعف هایی هستند.
درست مانند استاندارد HTML 5 که بر روی مرورگرهای مختلف به میزان‌های مختلفی پیاده سازی شده است.
به بیان دیگر پیاده سازی OWIN صفر و صدی نیست، بلکه هر پیاده سازی ممکن است در داخل خود دارای ضعف‌ها و یا نواقصی باشد.
علاوه بر این اگر شما در کد نویسی ASP.NET MVC خود، بی جهت به امکانات پایه ASP.NET ایجاد وابستگی کنید، نیز در این عمل دچار مشکل خواهید شد، برای همین بدیهتا کاری را که می‌توانید با Action Filter انجام دهید را نباید با یک Http Module انجام دهید و ...
مهم‌ترین کار طراحی برنامه هایی که می‌نویسید به صورت سازگار با OWIN است که در پست‌های بعدی قرار است به همین قسم از مطالب بپردازیم
البته من آینده خوبی برای OWIN قائلم، و نفع آن در کوتاه مدت و بلند مدت کاملا آشکار و واضح است، کما این که در مطلب به آن اشاره شد.

برای مشاهده پیاده سازی‌های مختلف OWIN می‌توانید به سایت owin.org مراجعه کنید.
موفق و پایدار باشید

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۲/۱۲/۰۳ ۱۹:۲۶

ممنون از پاسختون، البته این رو در نظر داشته باشید که استفاده از IIS به همراه Owin لزوما به پیاده سازی Katana یا همان Microsoft.Owin.Host.SystemWeb وابسته نیست، در این حالت شما هیچ گونه بهبود سرعتی رو مشاهده نخواهید کرد و حتی به علت اضافه شدن Owin Middleware بر روی ASP.NET حتی کندتر هم خواهید شد، این حالت فقط برای پروژه هایی توصیه می‌شود که با استفاده از مواردی مانند Moduleهای ASP.NET و یا Web form به ASP.NET وابسته اند. برای پروژه‌های جدید استفاده از Helios که نه به System.Web احتیاجی دارد و نه به Owin.Host.SystemWeb توصیه می‌شود، به همراه SignalR ، Web API و MVC، که به نظر من این ۳ آنقدر کامل و کافی هستند که لزومی به استفاده از ASP.NET System.Web.dll و پیاده سازی Owin مربوطه ای که نام بردید نباشد، تا بتوان بیشتر از مزایای Owin به خصوص کارامدی بیشتر برنامه‌ها بهره برد

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۲/۰۳ ۲۱:۵۵

ممنونم.
در حال حاضر من استفاده از helios رو پیشنهاد نمی‌کنم چون اولین محدودیتی که در helios جلب توجه می‌کند Minimum system requirements مورد نظر است.
برای توسعه پروژه‌های helios :
» Windows 8 یا Windows Server 2012
»NET Framework 4.5.1
»Visual Studio 2012 یا Visual Studio 2013

و برای Web Server نیز :
»Windows Server 2012
»NET Framework 4.5.1
»Full trust مورد نیاز است.
البته به گفته تیم توسعه پروژه helios، احتمال رفع این محدودیت‌ها در آینده وجود دارد. در نتیجه به نظر من Microsoft.AspNet.WebApi.OwinSelfHost گزینه بهتری برای Owin Self Hosting است و از آن جا که در حالت Owin Self Hosting هیچ گونه وابستگی به IIS و البته System.Web نیز وجود ندارد در نتیجه مشکل performance نیز برطرف خواهد شد.

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۲/۱۲/۰۴ ۰:۲

روش برنامه نویسی مایکروسافت بیش از دو سالی می‌شود که به این شکل شده است که هر امکان و قابلیت جدیدی بر روی آخرین نسخه NET Framework. ارائه می‌شود و البته سپس به نسخ قبلی نیز تعمیم می‌یابد، در همین جا است که می‌بینید اکثر امکانات Entity Framework 5 & 6 ابتدا بر روی NET Framework 4.5. ارائه شدند، و سپس بر روی 4
اگر ما بخواهیم به NET Framework. به عنوان یک پیش نیاز دردسر زا نگاه کنیم، در اولین قدم خودمان را به دردسر انداخته ایم، چون نه برای Helios، بلکه برای صدها امکان دیگر مانند Data Flowهای جدید و ... نیز باید صبر کنیم، که عملا هزینه به فایده آن نمی‌صرفد. پس همیشه با فراغ بال از آخرین نسخه NET Framework. استقبال کنیم
نکته‌ی دیگر را که باید مد نظر داشته باشیم، این است که مطابق با سیاست هایی که مایکروسافت جدیدا اتخاذ کرده است، دیگر نباید خیلی نگران نسخه‌های جدید NET Framework. باشیم، چون دیگر از آن نسخه دهی‌های پشت سر هم و با حجم تغییرات بالا خبری نیست، بلکه اکثر فریم ورک‌های مهم جدا از NET. ارائه و به روز رسانی می‌شوند.
علاوه بر این، ارتقا به آخرین نسخه سیستم عامل ویندوز نیز به هیچ وجه مانند قبل (IIS 6 به IIS 7) دردسر زا نیست، و خوشبختانه این ارتقا (و یا تغییر هاست) بدون دردسر است.

به نظر من این ارتقاء را انجام دهید، چون نه فقط Helios که خیلی چیزهای دیگری را دارید از دست می‌دهید، مانند سرعت بالاتر توسعه برنامه بر روی Visual Studio 2013 و Windows 8.1 برای توسعه برنامه‌های وبی، سرعت و کارآمدی بسیار بالاتر NET Framework 4.5.1. با IIS 8.5 برای مشتری‌های برنامه و ...

به نظر من آنقدر این ارتقاء ارزشمند است، که ارزش Helios این میان کمتر ارزشش به چشم می‌آید.

یکی از دلایلی که برنامه‌های سمت وب به سرعت بر برنامه‌های دسکتاپی قدیمی چیره شدند، همین است: امکان ارتقای سرورها در مدت زمان کم و به شکل مدیریت شده و با کمترین تاثیر روی مشتری‌های نهایی، بارها این تصمیم را که در ابتدایش کمی سخت به نظر می‌آید را گرفته ام و در نهایت از مشتری تا برنامه نویس همه را راضی دیده ام، چون هیچ کسی از امکانات جدید که بدون دردسر حاصل شود بدش نمی‌آید، و خوشبختانه کیفیت محصولات مایکروسافت واقعا بهبود یافته و دیگر آن زمانی که از NET 2. به 3.5 می‌رفتیم و گرفتار چندین مشکل می‌شدیم گذشته است.

از این نگذرید که بالاخره روزی باید این مهاجرت‌ها را انجام دهید، پس چه بهتر که از سود آن زودتر بهره مند شوید، البته بی مهابا عمل کردن توصیه نمی‌شود، بد نیست زمانی شروع به ارتقاء کنید که صفحه Release Notes و سوالات موجود در سایت Stack over flow در رابطه با اشکالات رخ داده در زمان ارتقاء و Breaking Changes را از بر باشید، به این صورت عمل کنید تماما برد کرده اید.

عنوان: **هاست سرویس های Asp.Net Web Api با استفاده از OWIN و TopShelf**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۲۷ ۲۳:۱۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, Self-Hosting, Windows Service, OWIN

زمانیکه از Templateهای پیش فرض تدارک دیده شده در VS.Net برای اپلیکیشن‌های وب خود استفاده می‌کنید، وب اپلیکیشن و سرور با هم یکپارچه هستند و تحت IIS اجرا می‌شوند. به وسیله Owin می‌توان این دو مورد را بدون وابستگی به IIS به صورت مجزا اجرا کرد. در این پست قصد داریم سرویس‌های Web Api را در قالب یک Windows Service با استفاده از کتابخانه‌ی TopShelf هاست نماییم.

پیش نیاز ها:

» Owin چیست

» تبدیل برنامه‌های کنسول ویندوز به سرویس ویندوز ان تی

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

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

حال یک کلاس Startup برای پیاده سازی Configurationهای مورد نیاز ایجاد می‌کنیم

در این قسمت می‌توانید تنظیمات زیر را پیاده سازی نمایید:

»سیستم Routing؛

»تنظیم Dependency Resolver برای تزریق وابستگی کنترلرهای Web Api؛

»تنظیمات hubهای SignalR(در حال حاضر SignalR به صورت پیش فرض نیاز به Owin برای اجرا دارد)؛

»رجیستر کردن Owin Middlewareهای 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، سرویس‌های Web Api را در آدرس مورد نظر هاست خواهد کرد. با فراخوانی متد Stop این سرویس‌ها نیز dispose خواهند شد.
در مرحله آخر باید شروع و توقف سرویس‌ها را تحت کنترل کلاس HostFactory کتابخانه TopShelf در آوریم. برای این کار کافیست 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());
                } );
            } );
        }
    }
```

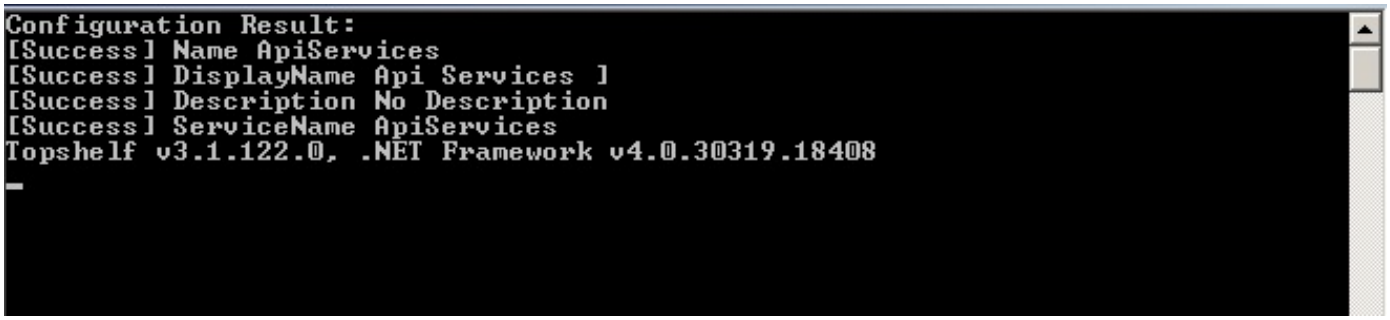
توضیح کدهای بالا:

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 می‌گیرد.

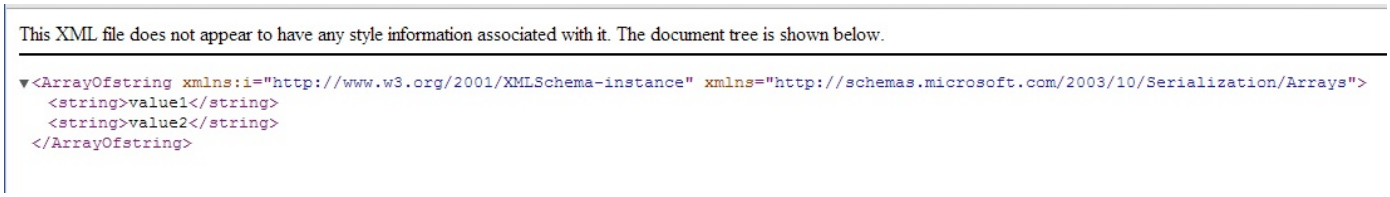
با استفاده از متد ConstructUsing عملیات وهله سازی از سرویس انجام خواهد گرفت. در پایان نیز متد Start و Stop کلاس ServiceHost، به عنوان عملیات شروع و پایان سرویس ویندوز مورد نظر تعیین شد.

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

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



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:



عنوان: **محدود کردن درخواست های Asp.Net Web Api بر اساس Client IP**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۶/۲۸ ۱۷:۱۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, Self-Hosting, IIS-Hosting, OWIN

در بسیاری از سناریوها این موضوع مطرح می‌شود که سرویس‌های طراحی شده بر اساس Asp.Net Web Api، فقط به یک سری آی پی‌های مشخص سرویس دهند. برای مثال اگر Ip کلاینت در لیست کلاینت‌های دارای لایسنس خریداری شده بود، امکان استفاده از سرویس میسر باشد؛ در غیر این صورت خیر. بسته به نوع پیاده سازی سرویس‌های 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 می‌کنیم.
در اینجا قصد داریم حالتی را پیاده سازی نماییم که اگر درخواست جاری از سوی کلاینتی بود که Ip آن در لیست Ipهای غیر مجاز قرار داشت، ادامه‌ی عملیات متوقف شود.

IIS Hosting:

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

```
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 می‌توان آن را از لیست Propertiesهای 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 می‌توان عملیات بالا را با استفاده از خاصیت RemoteEndpointMessageProperty انجام داد که تقریبا شبیه به حالت Web Host است. مقدار این خاصیت نیز در شی جاری *HttpRequestMessage* وجود دارد. فقط باید به صورت زیر آن را واکشی 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 درخواست‌ها وارده در سرویس‌های WCF چه در حالت استفاده از Http و چه در حالت Tcp اضافه می‌شود و در اسمبلی System.ServiceModel نیز می‌باشد. از آنجا که Web Api از هسته‌ی WCF استفاده می‌کند (WCF Core) در نتیجه می‌توان از این روش استفاده نمود. فقط باید اسمبلی System.ServiceModel را به پروژه‌ی خود اضافه نمایید.

ترکیب حالت‌های قبلی:

اگر می‌خواهید کدهای نوشته شده شما وابستگی به نوع هاست پروژه نداشته باشد، یا به معنای دیگر، در هر دو حالت به درستی کار کند می‌توانید به روش زیر حالت‌های 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 سرویس‌های Web Api خود استفاده می‌کنید کمی روال انجام کار متفاوت خواهد شد. در این مورد نیز می‌توانید از DelegatingHandlerها استفاده کنید. معرفی DelegatingHandler طراحی شده به Asp.Net PipeLine به صورت زیر خواهد بود:

```
public class Startup
    {
        public void Configuration( IAppBuilder 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 قسمت‌های مختلف برنامه است. برای مثال ممکن است قصد داشته باشید که بخش هایی که با Asp.Net SignalR نیز پیاده سازی شده‌اند، 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 قسمت‌های هاست شده‌ی 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 Ipهای غیر مجاز، از طریق سازنده در اختیار Middleware قرار می‌گیرد. اگر درخواست مجاز بود از طریق دستور Next.Invoke(request,response) کنترل برنامه به مرحله بعدی منتقل می‌شود در غیر صورت عملیات با کد 403 متوقف می‌شود. در نهایت برای معرفی این Middleware طراحی شده به Application، مراحل زیر را انجام دهید.

```
public class Startup
    {
        public void Configuration( IAppBuilder 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 IPها نیز همین است که شما همیشه آدرس Proxy Server را مشاهده می‌کنید.