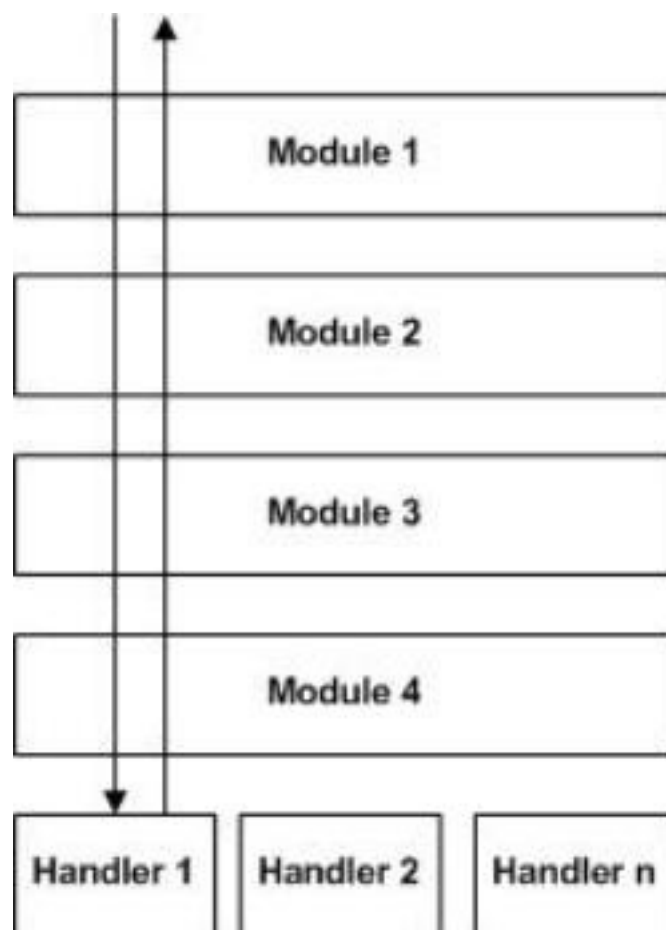


قبل از اینکه به httpmodule ها بپردازیم، اجازه بدید کمی در مورد [httphandler](#) اطلاعات کسب کنیم. httphandler ویژگی است که از asp.net به بعد ایجاد شد و در asp کلاسیک خبری از آن نیست. یک httphandler کامپوننتی است که از طریق اینترفیس System.Web.IHttpHandler پیاده سازی میشود و به پردازش درخواست های رسیده از httprequest رسیدگی می کند.

فرض کنید کاربری درخواست صفحه default.aspx را کرده است و سرور هم پاسخ آن را می دهد. در واقع پردازش اینکه چه پاسخی باید به کاربر یا کلاینت ارسال شود بر عهده این کامپوننت می باشد. برای وب سرویس هم موضوع به همین صورت است؛ هر نوع درخواست HTTP از این طریق انجام می شود.

حال به سراغ httpmodule می رویم. httpmodule ها اسمبلی یا ماژول هایی هستند که بر سر راه هر درخواست کاربر از سرور قرار گرفته و قبل از اینکه درخواست شما به httphandler برسد، اول از فیلتر این ها رد می شود. در واقع موقعی که شما درخواست صفحه default.aspx را می کنید، درخواست شما به موتور asp.net ارسال می شود و از میان فیلترهایی رد می شود تا به دست httphandler برای پردازش خروجی برسد. برای همین اگر گاهی به جای گفتن asp.net engine عبارت asp.net pipeline هم میگویند همین هست؛ چون درخواست شما از بین بخش های زیادی می گذرد تا به httphandler برسد که httpmodule یکی از آن بخش هاست. با هر درخواستی که سرور ارسال می شود، httpmodule ها صدا زده می شوند و به برنامه نویسی امکان بررسی اطلاعات درخواستی و پردازش درخواست ها را در ورودی و خروجی، می دهد و شما میتوانید هر عملی را که نیاز دارید انجام دهید. تعدادی از این ماژول های آماده، همان state ها و Authentication می باشند.

تصویر زیر نحوه ی ارسال و بازگشت یک درخواست را به سمت httphandler نشان می دهد



برنامه نویسی هم میتواند با استفاده از اینترفیس‌های IHttpModule و IHttpHandler در درخواست‌ها دخالت نماید. برای شروع یک کلاس ایجاد کنید که اینترفیس IHttpModule را پیاده سازی می‌کند. شما دو متد را باید در این کلاس بنویسید؛ یکی Init و دیگر Dispose. همانطور که مطلع هستید، اولی در ابتدای ایجاد شیء و دیگر موقع از دست رفتن شیء صدا زده می‌شود. متد Init یک آرگومان از نوع httpapplication دارد که مانند رسم نامگذاری متغیرها، بیشتر به اسم context یا app نام گذاری می‌شوند:

```
public void Init(HttpApplication app)
{
    app.BeginRequest += new EventHandler(OnBeginRequest);
}

public void Dispose(){ }
```

همانطور که می‌بینید این شیء یک رویداد دارد که ما این رویداد را به تابعی به نام OnBeginRequest متصل کردیم. سایر رویدادهای موجود در httpapplication به شرح زیر می‌باشند:

این رویداد اولین رویدادی است که اجرا می‌شود، هر نوع عملی که می‌خواهید در ابتدای ارسال درخواست انجام دهید، باید در این قسمت قرار بگیرد؛ مثلاً قرار دادن یک بنر بالای صفحه	BeginRequest
خود دانت از یک سیستم امنیتی توکار بهره مند است و اگر می‌خواهید در مورد آن خصوصی سازی انجام بدهید، این رویداد می‌تواند کمک‌تان کند	AuthenticateRequest
بعد از رویداد بالا، این رویداد برای شناسایی انجام می‌شود. مثلاً دسترسی‌ها؛ دسترسی به قسمت‌هایی خاصی از منابع به او داده شود و قسمت‌هایی بعضی از منابع از او گرفته شود.	AuthorizeRequest
این رویداد برای کش کردن اطلاعات استفاده می‌شود. خود دانت تمامی این رویدادها را به صورت توکار فراهم آورده است؛ ولی اگر باز خصوصی سازی خاصی مد نظر شماست می‌توانید در این قسمت‌ها، تغییراتی را اعمال کنید. مثلاً ایجاد file caching به جای memory cache و ...	ResolveRequestCache
این قسمت برای مدیریت state می‌باشد مثلاً مدیریت session ها	AcquireRequestState
این رویداد قبل از httphandler اجرا می‌شود.	PreRequestHandlerExecute
این رویداد بعد از httphandler اجرا می‌شود.	PostRequestHandlerExecute
این رویداد برای این صدا زده می‌شود که به شما بگوید عملیات درخواست پایان یافته است و باید state های ایجاد شده را release یا رها کنید.	ReleaseRequestState
برای خصوصی سازی output cache بکار می‌رود.	UpdateRequestCache
عملیات درخواست پایان یافته است. در صورتیکه قصد نوشتن دیباگری در طی تمامی عملیات دارید، میتواند به شما کمک کند.	EndRequest
این رویداد قبل از ارسال اطلاعات هدر هست. اگر قصد اضافه کردن اطلاعاتی به هدر دارید، این رویداد را به کار ببرید.	PreSendRequestHeaders
این رویداد موقعی صدا زده می‌شود که متد response.flush فراخوانی شود، اگر می‌خواهید به محتوا چیزی اضافه کنید، از اینجا کمک بگیرید.	PreSendRequestContent

این رویداد اولین رویدادی است که اجرا می‌شود، هر نوع عملی که می‌خواهید در ابتدای ارسال درخواست انجام دهید، باید در این قسمت قرار بگیرید؛ مثلاً قرار دادن یک بنر بالای صفحه	BeginRequest
این رویداد موقعی رخ می‌دهد که یک استثنای مدیریت نشده رخ بدهد. برای نوشتن سیستم خطایابی خصوصی از این قسمت عمل کنید.	Error
این رویداد موقعی صدا زده می‌شود که درخواست، بنا به هر دلیلی پایان یافته است. برای عملیات پاکسازی و .. می‌شود از آن استفاده کرد. مثلاً یک جور rollback برای کارهای انجام گرفته.	Disposed

کد زیر را در نظر بگیرید:

کد زیر یک رویداد را تعریف کرده و سپس خود httpapplication را به عنوان sender استفاده می‌کند. در اینجا قصد داریم یکی از صفحات را در خروجی تغییر دهیم. آدرس تایپ شده همان باشد ولی صفحه‌ی درخواست شده، صفحه‌ی دیگری است. این کار موقعی بیشتر کاربردی است که آدرس یک صفحه تغییر کرده و کاربر آدرس قبلی را وارد می‌کند. حالا یا از طریق بوک مارک یا از طریق یک لینک، در یک جای دیگر و شما می‌خواهید او را به صفحه‌ای جدید انتقال دهید، ولی در نوار آدرس، همان آدرس قبلی باقی بماند. همچنین کار دیگری که قرار است انجام بگیرد محاسبه مدت زمان رسیدگی به درخواست را محاسبه کند، برای همین در رویداد BeginRequest زمان شروع درخواست را ذخیره و در رویداد EndRequest با به دست آوردن اختلاف زمان فعلی با زمان شروع به مدت زمان مربوطه پی خواهیم برد. با استفاده از app.Context.Request.RawUrl اصلی و درخواست شده را یافته و در صورتی که شامل نام صفحه مربوطه بود، با نام صفحه‌ی جدید جابجا می‌کنیم تا اطلاعات به صفحه‌ی جدید پاس شوند ولی در نوار آدرس، هنوز آدرس قبلی یا درخواست شده، قابل مشاهده است. در خط app.Context.Items["start"] که یک کلاس ارث بری شده از اینترفیس IDictionary است، بر اساس کلید، داده شما را ذخیره و در مواقع لزوم در هر رویداد به شما باز می‌گرداند.

```
public class UrlPath : IHttpModule
{
    public void Init(HttpApplication app)
    {
        app.BeginRequest+=new EventHandler(_BeginRequest);
        app.EndRequest+=new EventHandler(_EndRequest);
    }

    public void Dispose()
    {
    }

    void _BeginRequest(object sender, EventArgs e)
    {
        HttpApplication app = (HttpApplication) sender;
        app.Context.Items["start"] = DateTime.Now;

        if (app.Context.Request.RawUrl.ToLower().Contains("tours_list.aspx"))
        {
            app.Context.RewritePath(app.Context.Request.RawUrl.ToLower().Replace("tours_list.aspx","tours_cat.aspx"));
        }
    }

    void _EndRequest(object sender, EventArgs e)
    {
        HttpApplication app = (HttpApplication)sender;
        string log = (DateTime.Now -
        DateTime.Parse(app.Context.Items["start"].ToString())).ToString();
        Debugger.Log(0,"duration","request took " + log+Environment.NewLine);
    }
}
```

```
}
```

حالا باید کلاس نوشته شده را به عنوان یک httpmodule به سیستم معرفی کنیم. به همین منظور وارد web.config شوید و کلاس جدید را معرفی کنید:

```
<httpModules>
  <add name="UrlPath" type="UrlPath"/>
</httpModules>
```

اگر کلاس شما داخل یک namespace قرار دارد، در قسمت type حتما قبل از نام کلاس، آن را تعریف کنید namespace.ClassName حالا دیگر کلاس UrlPath به عنوان یک httpmodule به سیستم معرفی شده است. تگ httpmodule را بین تگ <system.web> قرار داده ایم.

در ادامه پروژه را start بزنید تا نتیجه کار را ببینید:

اگر IIS شما، هم نسخه‌ی IIS من باشد، نباید تفاوتی مشاهده کنید و می‌بینید که درخواست‌ها هیچ تغییری نکردند؛ چرا که اصلا httpmodule اجرا نشده است. در واقع در نسخه‌های قدیمی IIS یعنی 6 به قبل، این تعریف درست است ولی از نسخه‌ی 7 به بعد IIS، روش دیگری برای تعریف را قبول دارد و باید تگ httpmodule، بین دو تگ <system.webserver> قرار بگیرد و نام تگ httpmodule به module تغییر پیدا کند. پس کد فوق به این صورت تغییر می‌کند:

```
<system.webServer>
  <modules>
    <add name="UrlRewrite" type="UrlRewrite"/>
  </modules>
</system.webServer>
```

حالا اگر قصد دارید که پروژه‌ی شما در هر دو IIS مورد حمایت قرار گیرد، باید این مازول را در هر دو جا معرفی کرده و در تگ system.webserver نیاز است تگ زیر تعریف شود که به طوری پیش فرض در webconfig می‌باشد:

```
<validation validateIntegratedModeConfiguration="false"/>
```

در غیر این صورت خطای زیر را دریافت می‌کنید:

HTTP Error 500.22 - Internal Server Error در کل استفاده از این مازول به شما کمک می‌کند تمامی اطلاعات ارسالی به سیستم را قبل از رسیدن به قسمت پردازش بررسی نمایید و هر نوع تغییری را که می‌خواهید اعمال کنید و لازم نیست این تغییرات را روی هر بخش، جداگانه انجام دهید یا یک کلاس بنویسید که هر بار در یک جا صدا بزنید و خیلی از موارد دیگر

HttpModule و Global.asax

اگر با global.asax کار کرده باشید حتما می‌پرسید که الان چه تفاوتی با httpmodule دارد؟ در فایل global هم همین‌ها را دارید و دقیقا همین مزایا مهیاست؛ در واقع global.asax یک پیاده سازی از httpapplication هست. کلاس‌های httpmodule نام دیگری هم دارند به اسم Portable global .asax به معنی یک فایل global.asax قابل حمل یا پرتابل. دلیل این نام گذاری این هست که شما موقعی که یک کد را در فایل global می‌نویسید، برای همیشه آن کد متعلق به همان پروژه هست و قابل انتقال به یک پروژه دیگر نیست ولی شما می‌توانید httpmodule ها را در قالب یک پروژه به هر پروژه ای که دوست دارید رفرنس کنید و کد شما قابلیت استفاده مجدد و Reuse پیدا می‌کند و هم اینکه در صورت نیاز می‌توانید آن‌ها را در قالب یک dll منتشر کنید.

نظرات خوانندگان

نویسنده: عباس حجتی
تاریخ: ۱۰:۲۴ ۱۳۹۳/۰۹/۲۹

با سلام؛ ممنون بابت مطلب خوبی که ارسال کردید.
من می‌خواهم از httpModule برای کنترل دسترسی کاربران (Authorization) استفاده کنم و به سشن کاربر نیاز دارم. مشکلی که هست موقع استفاده به این شکل HttpContext.Current.Session پیغام زیر رو میده:
Object not reference to instance of an object
آیا برای استفاده از سشن راه خاصی هست؟

نویسنده: وحید نصیری
تاریخ: ۱۴:۲۲ ۱۳۹۳/۰۹/۲۹

بله. در جدولی که تهیه کردند این مورد دقیقاً ذکر شده:
» **AcquireRequestState** : این قسمت برای مدیریت state می‌باشد مثلاً مدیریت session ها»
به این معنا که سشن در تمام رویدادگردان‌های آن مهیا نیست. فقط تعدادی از آن‌ها دسترسی به سشن دارند. برای مثال:

```
public class SimpleModule : IHttpModule
{
    void IHttpModule.Init(HttpApplication application)
    {
        application.BeginRequest += new System.EventHandler(BeginRequest);
        application.AcquireRequestState += new EventHandler(application_AcquireRequestState);
    }

    public void BeginRequest(object sender, EventArgs e)
    {
        // no session here
    }

    void application_AcquireRequestState(object sender, EventArgs e)
    {
        HttpApplication app = sender as HttpApplication;
        app.Session.Add("Message", "hello module");
    }

    public void Dispose()
    {
    }
}
```

نویسنده: علی یگانه مقدم
تاریخ: ۱۴:۵۰ ۱۳۹۳/۰۹/۲۹

از رویداد AcquireRequestState استفاده کنید ، همانطور که گفتیم این رویداد برای مدیریت stateهاست ، عموماً در رویدادهای ابتدایی sessionها هنوز ایجاد نشده‌اند
کد زیر نمونه ای از دسترسی به session هاست :

```
public void Init(HttpApplication app)
{
    app.AcquireRequestState+=new EventHandler(_AUTH);
}

private void _AUTH(object sender, EventArgs e)
{
    HttpApplication app = (HttpApplication) sender;
    HttpContext context = app.Context;
    HttpSessionState session = context.Session;

    if(session!=null)
    {
        string text = "session is not exist";
    }
}
```

```
        if (session["userid"] != null)
        {
            text = "session is exist";
        }
        context.Response.Write(text);
    }
}
```

توجه داشته باشید که خط `if(session!=null)` بسیار مهم هست و در صورت نبودن خط شرطی بعدی دچار خطایی که شما گرفتید می‌شود

نویسنده: عباس حجتی
تاریخ: ۱۷:۱۰ ۱۳۹۳/۰۹/۲۹

ممنون، درست شد.

یک مشکل دیگه اینکه درخواست‌های AJAX دیگه سمت سرور نمیره، می‌تونه به HttpModule ارتباطی داشته باشه؟

ممنون، این مشکل هم با فیلتر کردن درخواست‌های AJAX برطرف شد.

```
if (HttpContext.Current.Request.Headers["X-Requested-With"] != "XMLHttpRequest")
{
}
```

در [مقاله قبل](#) توضیح دادیم که وظیفه httphandler رندر و پردازش خروجی یک درخواست هست؛ حالا در این مقاله قصد داریم که مفهوم httphandler را بیشتر بررسی کنیم.

HttpHandler

برای تهیه‌ی یک httphandler، باید کلاسی را بر اساس اینترفیس IHttpHandler پیاده سازی کنیم و بعداً آن را در web.config برنامه معرفی کنیم. برای پیاده سازی این اینترفیس، به یک متد به اسم ProcessRequest با یک پارامتر از نوع HttpContext و یک پراپرتی به اسم IsReusable نیاز داریم که مقدار برگشتی این پراپرتی را false بگذارید؛ بعداً خواهیم گفت چرا اینکار را می‌کنیم. نحوه‌ی پیاده‌سازی یک httphandler به شکل زیر است:

```
public class MyHttpHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
    }

    public bool IsReusable
    {
        get { return false; }
    }
}
```

با استفاده از شیء context می‌توان به دو شیء httprequest و httpresponse دسترسی داشت. تکه کد زیر مثالی است در مورد نحوه‌ی تغییر در محتوای سایت:

```
public class MyHttpHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        HttpResponse response = context.Response;
        HttpRequest request = context.Request;

        response.Write("Every Page has a some text like this");
    }

    public bool IsReusable
    {
        get { return false; }
    }
}
```

بگذارید همین کد ساده را در وب کانفیگ معرفی کنیم:

```
<system.web>
  <httpHandlers>
    <add verb="*" path="*.aspx" type="MyHttpHandler"/>
  </httpHandlers>
</system.web>
```

اگر نسخه IIS شما همانند نسخه‌ی من باشد، نباید هیچ تغییری مشاهده کنید؛ زیرا کد بالا فقط در مورد نسخه‌ی IIS6 صدق می‌کند و برای نسخه‌های IIS 7 به بعد باید به شیوه زیر عمل کنید:

```
<configuration>
  <system.web>
    <httpHandlers>

    <add name="myhttphandler" verb="*" path="*.aspx" type="MyHttpHandler"/>
```

```
</httpHandlers>
</system.web>
</configuration>
```

خروجی نهایی باید تنها این متن باشد: Every Page has a some text like this

گزینه Type که نام کلاس می‌باشد و اگر کلاس داخل یک فضای نام قرار گرفته باشد، باید اینطور نوشت : namespace.ClassName

گزینه verb شامل مقادیری چون Get,Post,Head,Put و Delete می‌باشد و httpHandler را فقط برای این نوع درخواست‌ها اجرا می‌کند و در صورتیکه بخواهید چندتا از آن‌ها را استفاده کنید، با , از هم جدا می‌شوند. مثلا Get,post و در صورتیکه همه‌ی گزینه‌ها را بخواهید علامت * را میتوان استفاده کرد.

گزینه‌ی path این امکان را به شما می‌دهد که مسیر و نوع فایل‌هایی را که قصد دارید روی آن‌ها فقط اجرا شود، مشخص کنید و ما در قطعه کد بالا گفته‌ایم که تنها روی فایل‌هایی با پسوند aspx اجرا شود و چون مسیری هم ذکر نکردیم برای همه‌ی مسیرها قابل اجراست. یکی از مزیت‌های دادن پسوند این است که می‌توانید پسوندهای اختصاصی داشته باشید. مثلا پسوند RSS برای فیدهای وب سائتتان. بسیاری از برنامه نویسان به جای استفاده از صفحات aspx از ashx استفاده می‌کنند که به مراتب سبک‌تر از aspx هست و شامل بخش ui نمی‌شود و نتیجه خروجی آن بر اساس کدی که می‌نویسید مشخص می‌شود که میتواند صفحه متنی یا عکس یا xml یا ... باشد. در [اینجا](#) در مورد ساخت صفحات ashx توضیح داده شده است.

IHttpHandlerFactory

کار این اینترفیس پیاده سازی یک کلاس است که خروجی آن یک کلاس از نوع IHttpHandler هست. اگر دقت کنید در مثال‌های قبلی ما برای معرفی یک هندلر در وب کانفیگ یک سری path را به آن میدادیم و برای نمونه *.aspx را معرفی می‌کردیم؛ یعنی این هندلر را بر روی همه‌ی فایل‌های aspx اجرا کن و اگر دو یا چند هندلر در وب کانفیگ معرفی کنیم و برای همه مسیر aspx را قرار بدهیم، یعنی همه این هندلرها باید روی صفحات aspx اجرا گردند ولی در httpHandlerFactory، ما چند هندلر داریم و می‌خواهیم فقط یکی از آن‌ها بر روی صفحات aspx انجام بگیرد، پس ما یک هندلر فکتوری را برای صفحات aspx معرفی می‌کنیم و در حین اجرا تصمیم می‌گیریم که کدام هندلر را ارسال کنیم.

اجازه بدهید نوشتن این نوع کلاس را آغاز کنیم، ابتدا دو هندلر به نام‌های httpHandler1 و httpHandler2 می‌نویسیم :

```
public class MyHttpHandler1 : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        HttpResponse response = context.Response;
        response.Write("this is httpHandler1");
    }

    public bool IsReusable
    {
        get { return false; }
    }
}

public class MyHttpHandler2 : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        HttpResponse response = context.Response;
        response.Write("this is httpHandler2");
    }

    public bool IsReusable
    {
        get { return false; }
    }
}
```

سپس کلاس MyFactory را بر اساس اینترفیس IHttpFactory پیاده سازی می‌کنیم و باید دو متد برای آن صدا بزنیم؛ یکی که هندلر انتخابی را بر میگرداند و دیگری هم برای رها کردن یا آزادسازی یک هندلر هست که در این مقاله کاری با آن نداریم. عموما

GC دات نت در این زمینه کارآیی خوبی دارد. در قسمت هندلرهای غیرهمزمان به طور مختصر خواهیم گفت که GC چطور آن‌ها را مدیریت می‌کند. کد زیر نمونه کلاسی است که توسط IHttpFactory پیاده سازی شده است:

```
public class MyFactory : IHttpHandlerFactory
{
    public IHttpHandler GetHandler(HttpContext context, string requestType, string url, string pathTrasnlated)
    {
    }

    public void ReleaseHandler(IHttpHandler handler)
    {
    }
}
```

در متد GetHandler چهار آرگومان وجود دارند که به ترتیب برای موارد زیر به کار می‌روند:

یک شی از کلاس httpcontext که دسترسی ما را برای اشیاء سروری چون response,request,session و... فراهم می‌کند.	Context
مشخص می‌کند که درخواست صفحه به چه صورتی است. این گزینه برای مواردی است که verb بیش از یک مورد را حمایت می‌کند. برای مثال دوست دارید یک هندلر را برای درخواست‌های Get ارسال کنید و هندلر دیگر را برای درخواست‌های نوع Post	RequestType
مسیر مجازی virtual Path صفحه صدا زده شده	URL
مسیر فیزیکی صفحه درخواست کننده را ارسال می‌کند.	PathTranslated

متد GetHandler را بدین شکل می‌نویسیم و می‌خواهیم همه صفحات aspx هندلر شماره یک را انتخاب کنند و صفحات aspx نامشان با t شروع می‌شوند، هندلر شماره دو را انتخاب کند:

```
public IHttpHandler GetHandler(HttpContext context, string requestType, string url, string pathTrasnlated)
{
    string handlername = "MyHttpHandler1";
    if(url.Substring(url.LastIndexOf("/") + 1).StartsWith("t"))
    {
        handlername = "MyHttpHandler2";
    }

    try
    {
        return (IHttpHandler) Activator.CreateInstance(Type.GetType(handlername));
    }
    catch (Exception e)
    {
        throw new HttpException("Error: " + handlername, e);
    }
}

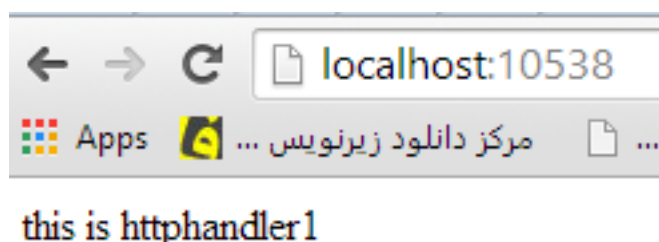
public void ReleaseHandler(IHttpHandler handler)
{
}
}
```

شی Activator که برای ساخت اشیاء با انتخاب بهترین constructor موجود بر اساس یک نوع Type مشخص به کار می‌رود و خروجی Object را می‌گرداند؛ با یک تبدیل ساده، خروجی به قالب اصلی خود باز می‌گردد. برای مطالعه بیشتر در مورد این کلاس به [اینجا](#) و [اینجا](#) مراجعه کنید.

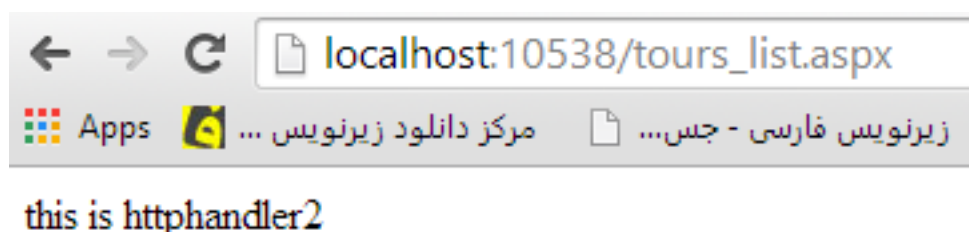
نحوه‌ی تعریف factory در وب کانفیگ مانند قبل است و فقط باید در Type به جای نام هندلر نام فکتوری را نوشت. برنامه را اجرا

کنید تا نتیجه آن را ببینیم:

تصویر زیر نتیجه صدا زده شدن فایل default.aspx است:



تصویر زیر نتیجه صدا زده شدن فایل Tours_List.aspx است:



AsyncHttpHandlers

برای اینکه کار این اینترفیس را درک کنید بهتر هست [اینجا](#) را مطالعه کنید. در اینجا به خوبی تفاوت متدهای همزمان و غیرهمزمان توضیح داده شده است.

متن زیر خلاصه‌ترین و بهترین توضیح برای این پرسش است، چرا غیرهمزمان؟ در عملی که disk I/O و یا network I/O دارند، پردازش موازی و اعمال async به شدت مقیاس پذیری سیستم را بالا می‌برند. به این ترتیب worker thread جاری (که تعداد آن‌ها محدود است)، سریعتر آزاد شده و به worker pool بازگشت داده می‌شود تا بتواند به یک درخواست دیگر رسیده سرویس دهد. در این حالت می‌توان با منابع کمتری، درخواست‌های بیشتری را پردازش کرد. موقعی که اینترفیس IHttpAsyncHandler را ارث بری کنید (این اینترفیس نیز از IHttpHandler ارث بری کرده است و دو متد اضافه‌تر دارد)، باید دو متد دیگر را نیز پیاده سازی کنید:

```
public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback callback, object obj)
{
}

public void EndProcessRequest(IAsyncResult result)
{
}
```

پراپرتی ISResuable هم موقعی که true برگشت بدهد، باعث می‌شود pooling فعال شده و این هندلر در حافظه باقی بماند و تمامی درخواست‌ها از طریق همین یک نمونه اجرا شوند.

به زبان ساده‌تر، این پراپرتی می‌گوید اگر چندین درخواست از طرف کلاینت‌ها برسد، توسط یک نمونه یا instance از هندلر پردازش خواهند شد؛ چون به طور پیش فرض موقعی که تمام درخواست‌های از pipeline بگذرند، هندلرها توسط httpapplication در یک لیست بازیافت قرار گرفته و همه‌ی آن‌ها با null مقداردهی می‌شوند تا از حافظه پاک شوند ولی اگر این پراپرتی true برگرداند، هندلر مربوطه نال نشده و برای پاسخگویی به درخواست‌های بعدی در حافظه خواهد ماند. مهمترین مزیت این گزینه، این می‌باشد که کآیی سیستم را بالا می‌برد و اشیا کمتری به GC پاس می‌شوند. ولی یک عیب هم دارد

که این تردهایی که ایجاد می‌کند، امنیت کمتری دارند و باید توسط برنامه نویسی این امنیت بالاتر رود. این پراپرتی را در مواقعی که با هندلرهای همزمان کار می‌کنید برابر با `false` بگذارید چون این گزینه بیشتر بر روی هندلرهای غیرهمزمان اثر دارد و هم اینکه بعضی‌ها توصیه می‌کنند که `false` بگذارید چون GC مدیریت خوبی در مورد هندلرها دارد و هم این که ارزش یافتن باگ در کد را ندارد.

بر میگردیم سراغ کد نویسی هندلر غیر همزمان. در آخرین قطعه کد نوشته شده، ما دو متد دیگر را پیاده سازی کردیم که یکی از آن‌ها `BeginProcessRequest` است و خروجی آن کلاسی است که از اینترفیس `AsyncResult` ارث بری کرده است. پس یک کلاس با ارث بری از این اینترفیس می‌نویسیم و در این کلاس نیاز است که 4 پراپرتی را پیاده سازی کنیم که این کلاس به شکل زیر در خواهد آمد:

```
public class AsyncOperation : IAsyncResult
{
    private bool _completed;
    private Object _state;
    private AsyncCallback _callback;
    private HttpContext _context;

    bool IAsyncResult.IsCompleted { get { return _completed; } }
    WaitHandle IAsyncResult.AsyncWaitHandle { get { return null; } }
    Object IAsyncResult.AsyncState { get { return _state; } }
    bool IAsyncResult.CompletedSynchronously { get { return false; } }
}
```

متدهای `private` اجباری نیستند؛ ولی برای ذخیره مقادیر `get` و `set` نیاز است. همانطور که از اسامی آن‌ها پیداست مشخص است که برای چه کاری ساخته شده اند. خب اجازه بدهید یک تابع سازنده به آن برای مقداردهی اولیه این متغیرهای خصوصی داشته باشیم:

```
public AsyncOperation(AsyncCallback callback, HttpContext context, Object state)
{
    _callback = callback;
    _context = context;
    _state = state;
    _completed = false;
}
```

همانطور که می‌بینید موارد موجود در متد `BeginProcessRequest` را تحویل می‌گیریم تا اطلاعات درخواستی مربوطه را داشته باشیم و مقدار `_Completed` را هم برابر با `false` قرار می‌دهیم. سپس نوبت این می‌رسد که ما درخواست را در صف `pool` قرار دهیم. برای همین تکه کد زیر را اضافه می‌کنیم:

```
public void StartAsyncWork()
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(StartAsyncTask), null);
}
```

با اضافه شدن درخواست به صف، هر موقع درخواست‌های قبلی تمام شوند و [callback](#) خودشان را ارسال کنند، نوبت درخواست‌های جدیدتر هم میرسد. `StartAsyncTask` هم متدی است که وظیفه‌ی اصلی پردازش درخواست را به دوش دارد و موقعی که نوبت درخواست برسد، کدهای این متد اجرا می‌گردد که ما در اینجا مانند مثال اول روی صفحه چیزی نوشتیم:

```
private void StartAsyncTask(Object workItemState)
{
    _context.Response.Write("<p>Completion IsThreadPoolThread is " +
        Thread.CurrentThread.IsThreadPoolThread + "</p>\r\n");

    _context.Response.Write("Hello World from Async Handler!");
    _completed = true;
    _callback(this);
}
```

دو خط اول اطلاعات را چاپ کرده و در خط سوم متغیر `_completed` را `true` کرده و در آخر این درخواست را فراخوانی مجدد

می‌کنیم تا بگوییم که کار این درخواست پایان یافته‌است؛ پس این درخواست را از صف بیرون بکش و درخواست بعدی را اجرا کن. نهایتاً کل این کلاس را در متد `BeginProcessRequest` صدا بزنید:

```
context.Response.Write("<p>Begin IsThreadPoolThread is " + Thread.CurrentThread.IsThreadPoolThread +
"</p>\r\n");
    AsyncOperation async = new AsyncOperation(callback, context, obj);
    async.StartAsyncWork();
    return async;
```

کل کد مربوطه : (توجه: کدها از داخل سایت *msdn* برداشته شده است و اکثر کدهای موجود در نت هم به همین قالب می‌نویسند)

```
public class MyHttpHandler : IHttpAsyncHandler
{
    public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback callback, object obj)
    {
        context.Response.Write("<p>Begin IsThreadPoolThread is " +
Thread.CurrentThread.IsThreadPoolThread + "</p>\r\n");
        AsyncOperation async = new AsyncOperation(callback, context, obj);
        async.StartAsyncWork();
        return async;
    }

    public void EndProcessRequest(IAsyncResult result)
    {
    }

    public void ProcessRequest(HttpContext context)
    {
        throw new InvalidOperationException();
    }

    public bool IsReusable
    {
        get { return false; }
    }
}

public class AsyncOperation : IAsyncResult
{
    private bool _completed;
    private Object _state;
    private AsyncCallback _callback;
    private HttpContext _context;

    bool IAsyncResult.IsCompleted { get { return _completed; } }
    WaitHandle IAsyncResult.AsyncWaitHandle { get { return null; } }
    Object IAsyncResult.AsyncState { get { return _state; } }
    bool IAsyncResult.CompletedSynchronously { get { return false; } }

    public AsyncOperation(AsyncCallback callback, HttpContext context, Object state)
    {
        _callback = callback;
        _context = context;
        _state = state;
        _completed = false;
    }

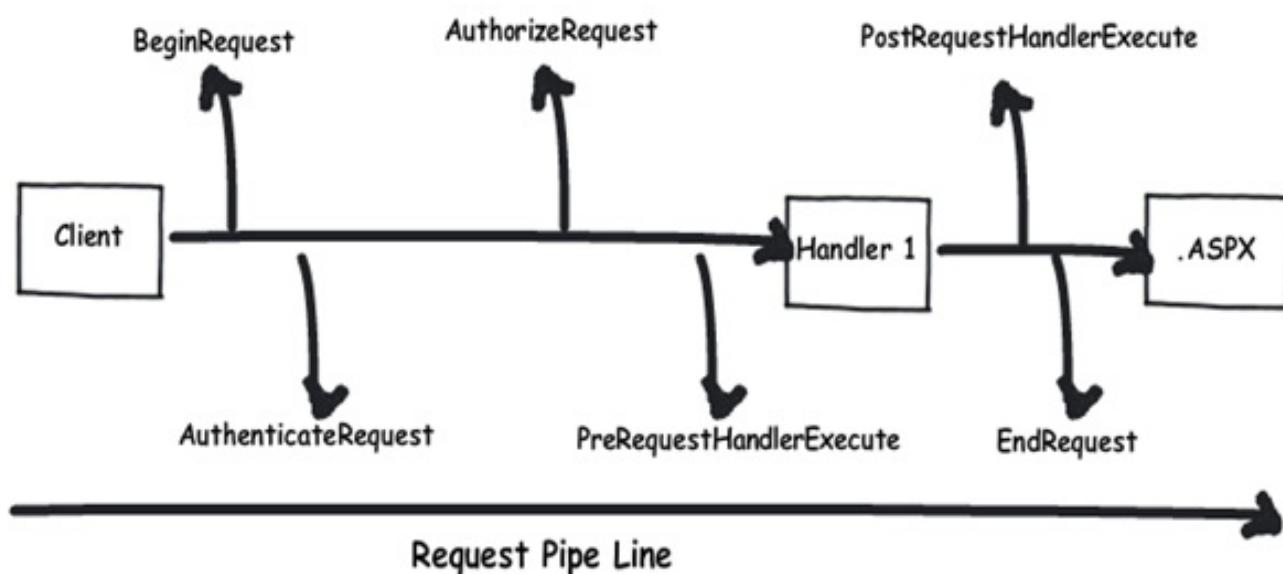
    public void StartAsyncWork()
    {
        ThreadPool.QueueUserWorkItem(new WaitCallback(StartAsyncTask), null);
    }

    private void StartAsyncTask(Object workItemState)
    {
        _context.Response.Write("<p>Completion IsThreadPoolThread is " +
Thread.CurrentThread.IsThreadPoolThread + "</p>\r\n");

        _context.Response.Write("Hello World from Async Handler!");
        _completed = true;
        _callback(this);
    }
}
```

آشنایی با فایل ASHX

در مطالب بالاتر به فایل‌های Ashx اشاره کردیم. این فایل به نام Generic Web Handler شناخته می‌شوند و می‌توانید با Add New Item این نوع فایل‌ها را اضافه کنید. این فایل شامل هیچ UI ای نمی‌باشد و فقط شامل بخش کد می‌باشد. برای همین نسبت به aspx سبک‌تر بوده و شامل یک directive به اسم **@WebHandler** است. مایکروسافت در MSDN نوشته است که http handler ها در واقع فرآیندهایی هستند (به این فرایندها بیشتر End Point می‌گویند) که در پاسخ به درخواست‌های رسیده شده توسط asp.net application اجرا می‌شوند و بیشترین درخواست‌هایی هم که می‌رسد از نوع صفحات Aspx می‌باشد و موقعی که کاربری درخواست صفحه‌ای aspx می‌کند هندلرهای مربوط به page اجرا می‌شوند. در متن بالا به خوبی روشن هست که ashx به دلیل نداشتن UI، تعداد کمتری از handler ها را در مسیر Pipeline قرار می‌دهند و اجرای آن‌ها سریعتر است. غیر از این دو هندلر aspx و ashx، هندلر توکار دیگری چون asmx که مختص وب سرویس هست و axd مربوط به اعمال trace نیز وجود دارند.



created with Balsamiq Mockups - www.balsamiq.com

در [این لینک](#) که در بالاتر هم درج شده بود یک نمونه هندلر برای نمایش تصویر نوشته است. اگر تصاویرتان را بدین صورت اجرا کنید می‌توان جلوی درخواست‌های رسیده از وب سایت‌های دیگر را سد کرد. برای مثال یک نفر مطالب شما را کپی می‌کند و در داخل وبلاگ یا وب سایتش می‌گذارد و شما در اینجا درخواست‌های رسیده خارج از وب سایت خود را لغو خواهید کرد و تصاویر کپی شده نمایش داده نخواهند شد.