عنوان: **با HttpHandler بیشتر آشنا شوید**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۳/۰۹/۲۷ ۱۵:۳۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, IhttpHandler, IhttpHandlerFactory, IHttpAsyncHandler

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 بعدا 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؛ بعدا خواهم 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 httpresponse و httprequest 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 درصورتیکه 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 سایتتان. 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 هندلرفکتوری 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:

Context	یک شی از کلاس httpcontext که دسترسی ما را برای اشیاء سروری چون response,request,session و... فراهم می‌کند.
RequestType	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
URL	مسیر مجازی virtual Path صفحه صدا زده شده
PathTranslated	مسیر فیزیکی صفحه درخواست کننده را ارسال می‌کند.

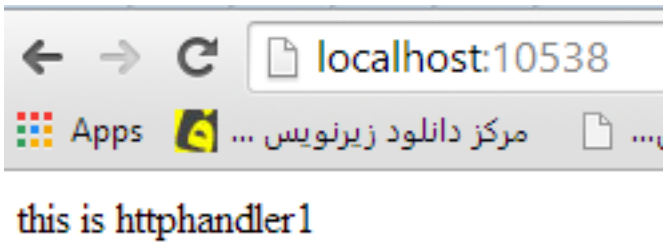
متد GetHandler را بدین شکل می‌نویسیم و میخواهیم همه صفحات aspx هندلر شماره یک را انتخاب کنند و صفحات aspx که نامشان با t شروع می‌شوند، هندلر شماره دو را انتخاب کند:

```
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)
    {

    }
}
```

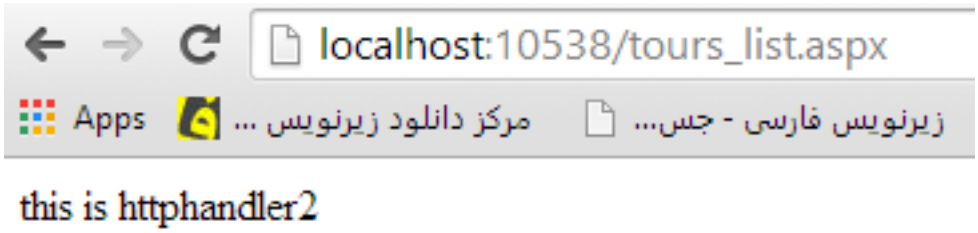
شی Activator که برای ساخت اشیاء با انتخاب بهترین constructor موجود بر اساس یک نوع Type مشخص به کار می‌رود و خروجی Object را می‌گرداند؛ با یک تبدیل ساده، خروجی به قالب اصلی خود باز میگردد. برای مطالعه بیشتر در مورد این کلاس به اینجا و اینجا مراجعه کنید.

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 اعمالی 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 کاآیی 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 نویس 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 IAsyncResult 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 AsynchOperation : 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 AsynchOperation(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. نهایتا کل این کلاس را در متد BeginProcessRequest صدا بزنید:

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

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

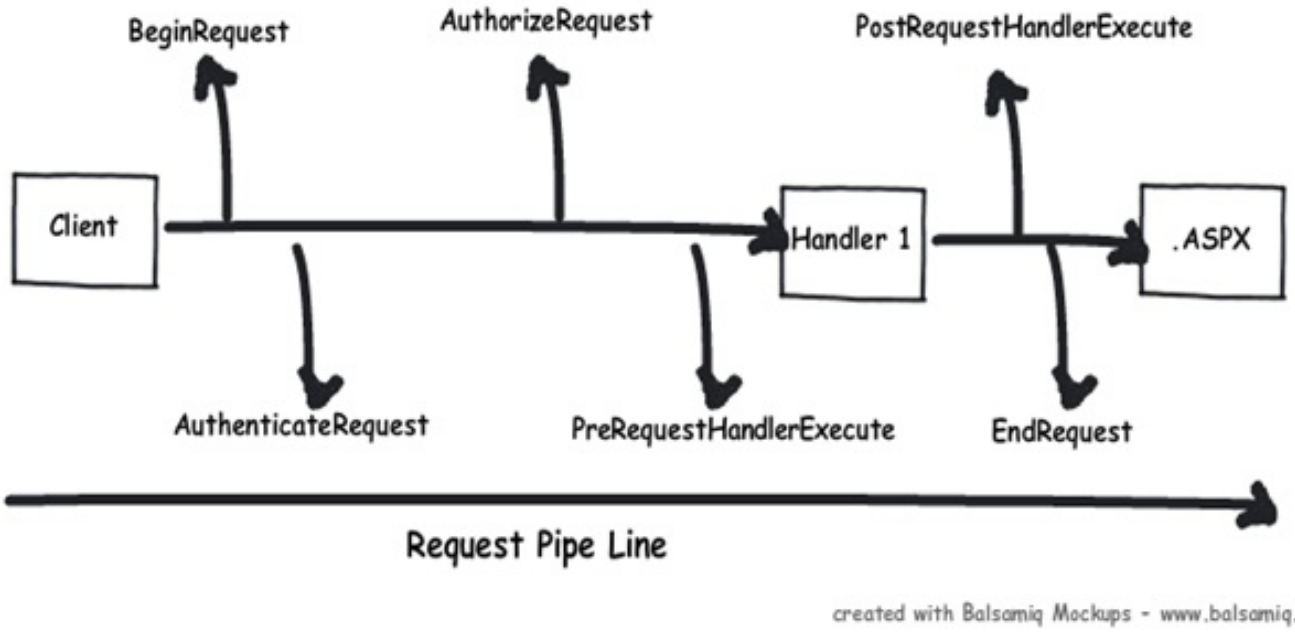
```
public class MyHttpHandler : IHttpAsyncHandler
{
    public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback callback, object obj)
    {
        context.Response.Write("<p>Begin IsThreadPoolThread is " +
Thread.CurrentThread.IsThreadPoolThread + "</p>\r\n");
        AsynchOperation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

    public void EndProcessRequest(IAsyncResult result)
    {

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

    }

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

public class AsynchOperation : 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 AsynchOperation(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 ایی 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 httphandlerها 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 درخواست هایی هم که می‌رسد از نوع صفحات Aspx می‌باشد و موقعی که کاربری درخواست صفحه‌ی aspx می‌کند هندلرهای مربوط به page اجرا می‌شوند.

در متن بالا به خوبی روشن هست که ashx به دلیل نداشتن UI، تعداد کمتری از handlerها 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.



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.