Deep into ASP.NET API Pipeline

Part 1

Delegating Pattern in ASP.NET API

By:

Masoud Bahrami

http://Refactor.Ir

MBahrami1990@Gmail.Com

در این قسمت از سری مقالات مربوط به توضیح و کالبد شکافی Pipeline مورد استفاده توسط ASP.NET API به یکی از بخش های بسیار مهم مورد استفاده توسط ASP.NET API Engine خواهم پرداخت.

هر کدام از فریمورک های ارائه شده به عنوان Application Server دارای یک مسیر و Pipeline مشخصی هستند که در طی اون فریمورک مورد نظر اقدام به دریافت Client Request از web server نموده و سپس طی اعمال فرآیندها و پردازش های مورد نظر Response مورد نظر رو در مسیر عکس به Client می فرستد. به عنوان مثالی Spring یک فریمورک محبوب و معروف Java هست که به عنوان بخشی از این فریمورک بزرگ و معروف Spring MVC به عنوان راهکار Application Server ارائه شده توسط Spring عمل می کند. پس از ارسال درخواست های از سمت کلاینت این درخواست ها از طریق وب سروری شبیه Tomcat؛ دریافت شده و تحویل Spring MVC Engine شده؛ سپس Spring MVC با استفاده و پیاده سازی کردن Front End Pattern وظیفه Orchestration نمودن این درخواست و ارسال اون به درون pipeline مربوط به spring mvc رو بر عهده داشته(شامل یافتن کنترلر مورد نظر؛ یافتن و invoke کردن Action مورد نظر و نهایتا یافتن View مورد نظر(در صورت نیاز)) و در نهایت پاسخ رو به کلاینت بر می گرداند.

در ASP.NET API وظیفه ی کنترل و Orchestration درخواست های کلاینت دریافت شده از طریق IIS(و سایر وب سرورها) بر عهده ی HttpMessageHandler ای هست که بصورت Built in وجود دارد.

```
namespace System.Net.Http
{
  /// <summary>
  /// A base type for HTTP message handlers.
  /// </summary>
  [__DynamicallyInvokable]
  public abstract class HttpMessageHandler : IDisposable
  {
    /// <summary>
    /// Initializes a new instance of the <see
cref="T:System.Net.Http.HttpMessageHandler"/> class.
    /// </summary>
    [__DynamicallyInvokable]
    protected HttpMessageHandler()
    {
      if (Logging.On)
        Logging.Enter(Logging.Http, (object) this, ".ctor", (string) null);
      if (!Logging.On)
        return;
      Logging.Exit(Logging.Http, (object) this, ".ctor", (string) null);
    }

    /// <summary>
    /// Send an HTTP request as an asynchronous operation.
```

```
    /// </summary>
    ///
    /// <returns>
    /// Returns <see cref="T:System.Threading.Tasks.Task`1"/>.The task object
representing the asynchronous operation.
    /// </returns>
    /// <param name="request">The HTTP request message to send.</param><param
name="cancellationToken">The cancellation token to cancel
operation.</param><exception cref="T:System.ArgumentNullException">The <paramref
name="request"/> was null.</exception>
    [__DynamicallyInvokable]
    protected internal abstract Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken cancellationToken);

    /// <summary>
    /// Releases the unmanaged resources used by the <see
cref="T:System.Net.Http.HttpMessageHandler"/> and optionally disposes of the managed
resources.
    /// </summary>
    /// <param name="disposing">true to release both managed and unmanaged resources;
false to releases only unmanaged resources.</param>
    [__DynamicallyInvokable]
    protected virtual void Dispose(bool disposing)
    {
    }

    /// <summary>
    /// Releases the unmanaged resources and disposes of the managed resources used
by the <see cref="T:System.Net.Http.HttpMessageHandler"/>.
    /// </summary>
    [__DynamicallyInvokable]
    public void Dispose()
    {
      this.Dispose(true);
      GC.SuppressFinalize((object) this);
    }
  }
}
```

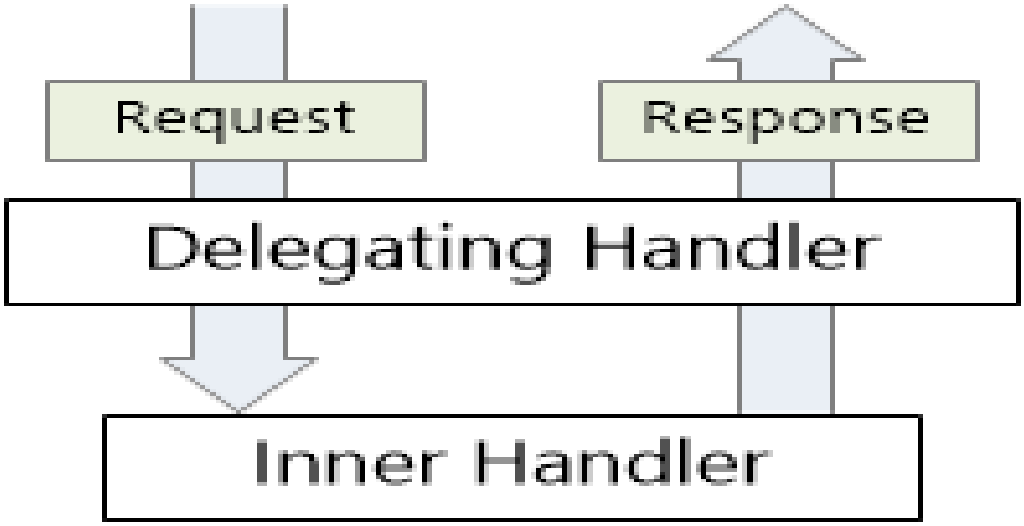
همانطور که در بالا اشاره شد HttpMessageHandler پس از دریافت HTTP Request وظیفه ی برگرداندن HTTP Response مناسب رو بعهده خواهد داشت. برای اینکار این کلاس کارهای متعددی انجام خواهد داد-از جمله انجام عمل Route Matching جهت Request با Route Table و یافتن Route Template مناسب؛ Resolve نمودن کنترلر مناسب و ... .البته باز همانطور که در بالا اشاره کردم HttpMessageHandler در واقع فقط کار Orcehstration را بر عهده خواهد داشت بدین معنی که این کلاس مثلا جهت انجام Route Matching کلاس و سرویس های مورد نظر که دات نت برای اینکار طراحی

کرده است را صدا می زند. و مقلا از Factory Controller جهت Instantiate نمودن کنترلر مورد نظر استفاده می کند.

خوب ضمن اینکه HttpMessageHandler تمام تمهیدات لازم را در نظر گرفته است جهت پردازش مناسب HTTP Request اما سناریو های متفاوت و بسیاری زیادی وجود خواهد داشت که نیاز می باشد که بسته ی به نیاز Custom Processing های مورد نظر Developer نیز در طی مسیر ارسال درخواست به درون Pipeline تا تولید پاسخ مناسب برای درخواست مذکورقابل اعمال باشد. بدین معنی که به Developer این امکان داده شود که قادر باشد که کنترل ها و پردازش های مورد نظر خود را نیز اعمال کند. مثلا فرض کنید برای درخواست های رسیده از دامین و سرور خاصی قصد افزودن HTTP Header های خاصی به HTTP Request قبل از فراخوانی Action مورد نظر هستیم. به عنوان مثالی دیگر قصد داریم که تمامی درخواست های رسیده از دامین خاص یا بدون HTTP Header خاص را در اولین فرصت و قبل از هر فراخوانی کنترلر و اکشن مورد نظر بررسی و در صورت عدم تائید بودن پاسخ/خطا ی مناسب برگشت داده شود.

HttpMessageHandler برای اینکار اقدام به پیاده سازی مناسبی از Delegating Pattern نموده که به Developer اجازه می دهد که هندلر مورد نظر خود را تعریف کند و سپس HttpMessageHandler به محض دریافت HTTP Request ابتدا این درخواست را به هندلر شما می فرستد و شما می توانید بسته به نیاز پاسخ را همانجا به کاربر برگردانید و Pipeline رو قطع کنید و یا اینکه به ارسال مجدد درخواست به HttpMessageHandler به HttpMessageHandler این اجازه را بدهید که درخواست مورد نظر رو جهت پردازش در طی Pipeline به پیش ببرد. در ادامه به نحوه ی عملکرد Delegating Pattern خواهم پرداخت.

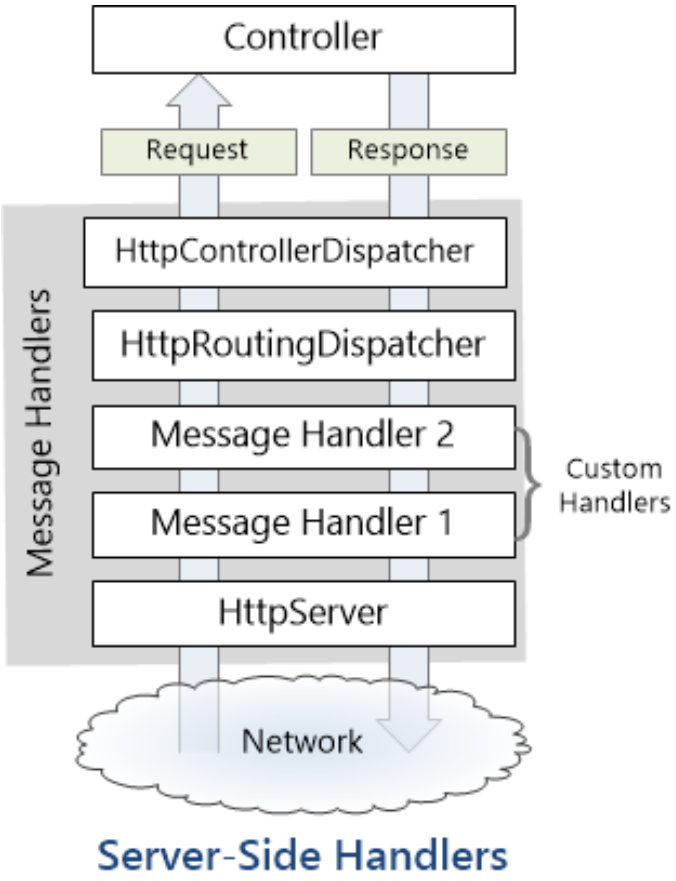
بطور معمول مجموعه ای از هندلر های مختلف بصورت زنجیر وار پشت سر هم قرار می گیرند و درخواست به ترتیب توسط هرکدام از این هندلر ها پردازش شده و به هندلر بعدی داده خواهد شد. در این تسلسل هندلرها هر کدام از هندلر ها می تواند این زنجیره را قطع و از ارسال درخواست به هندلر بعدی خودداری کند.



در ASP.NET API بصورت پیش فرض چند Message Handler وجود دارد؛ که در زیر به اونها اشاره میشه:

- **HTTP Server**: وظیفه ی این هندلر دریافت request از host می باشه.
- **HTTP Routing Dispatcher**: بر اساس route table ایجاد شده بر اساس route templateهای تعریف شده توسط کاربر کار dispatch کردن درخواست رو انجام می ده.
- **HTTP Controller Dispatcher**: در مرحله ی آخر درخواست رو به API Controller مورد نظر ارسال می کنه.

خوب اینها هندلر هایی هستند که تشکیل pipeline رو در ASP.NET API رو می دهند. شما هم می تونید که هندلر های مورد نظر خود رو تعریف و به این لیست هندلرها جهت اضافه شدن به Pipeline اضافه کنید. بهترین و مناسب ترین مورد جهت استفاده از هندلر های cross-cutting concerns توسعه دهنده ها در سطح HTTP Request می باشه.



نحوه ی تعریف Message Handler توسط کاربر

جهت ایجاد یک Message Handler باید کلاسی تعریف کرده و سپس کلاس مذکور از کلاس **System.Net.Http.DelegatingHandler** باید ارث بری کرده و متد **SendAsync** آن را override کند.

```
protected async override Task<HttpResponseMessage> SendAsync(
                    HttpRequestMessage request, CancellationToken cancellationToken)
```

همانطور که ملاحظه می کنید این متد یک HTTP Request رو دریافت و یک HTTP Response Message رو بر می گردونه. پیاده سازی معمول این متد شامل مراحل زیر می باشد:

۱. پردازش Request Message
۲. صدا زدن `base.SendAsync` و ارسال درخواست به inner handler
۳. سپس بصورت غیر assynchrounous جواب رو بصورت http response message بر می گرداند.
۴. پردازش response و return کردن اون

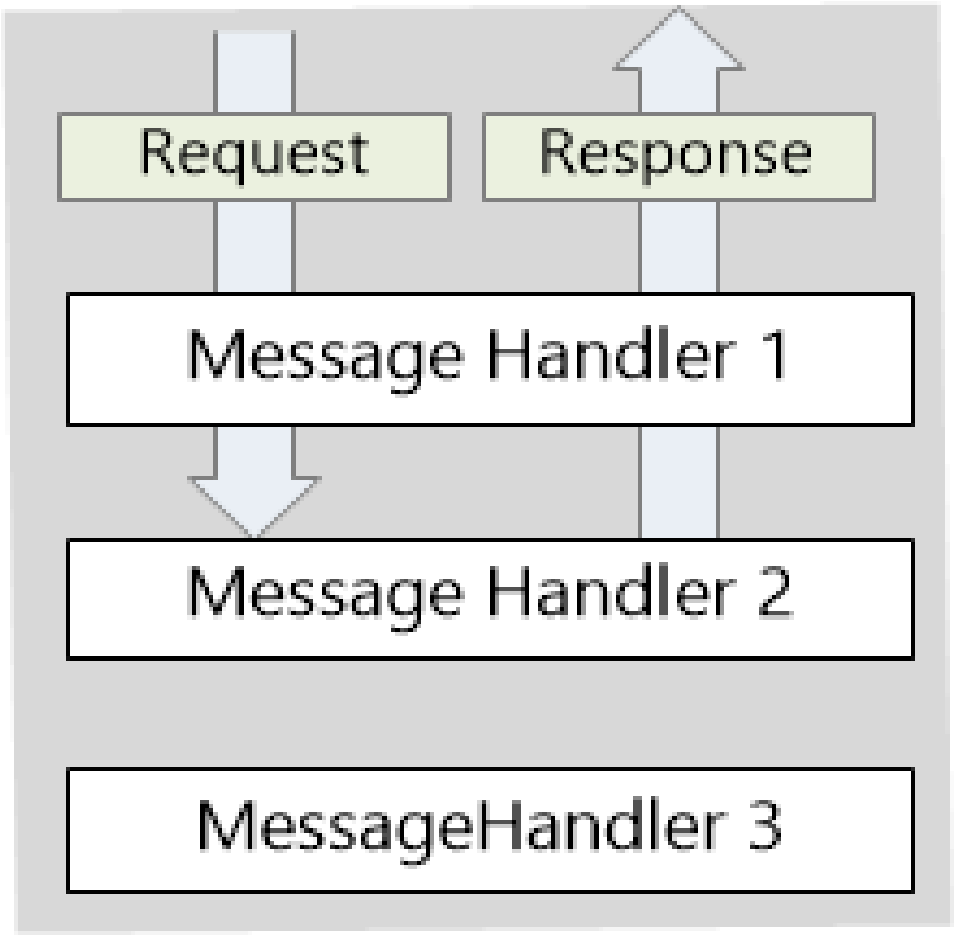
```
public class MessageHandler1 : DelegatingHandler
{
    protected async override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        Debug.WriteLine("Process request");
        // Call the inner handler.
        var response = await base.SendAsync(request, cancellationToken);
        Debug.WriteLine("Process response");
        return response;
    }
}
```

همچنین باید توجه نمود که جهت قطع نمودن Pipeline هندلر مورد نظر می تونه از فراخوانی base.SendAsync خود داری کرده و خود پاسخ مورد نظر رو بر گردونه. با اینکار عملا از ادامه روند کار و نهایتا از invoke کردن action مورد نظر در API Controller جلوگیری خواهد شد.

```
public class MessageHandler2 : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        // Create the response.
        var response = new HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = new StringContent("Hello!")
        };

        // Note: TaskCompletionSource creates a task that does not contain a
delegate.
        var tsc = new TaskCompletionSource<HttpResponseMessage>();
        tsc.SetResult(response);   // Also sets the task state to "RanToCompletion"
```

```
        return tsc.Task;
    }
}
```



اما نحوه ی افزودن یک Custom Message Handler به Pipeline

برای این منظور می توان براحتی با افزودن هندلر مورد نظر به **HttpConfiguration.MessageHandlers** این کار رو انجام داد. باید توجه داشت که هر تعداد هندلری رو میشه به این طریق اضافه کرد و ترتیب ارسال HTTP Request Message به این هندلرها به ترتیب Register شدن اونها هست.

```
public static class WebApiConfig
```

```
{
    public static void Register(HttpConfiguration config)
    {
        config.MessageHandlers.Add(new MessageHandler1());
        config.MessageHandlers.Add(new MessageHandler2());

        // Other code not shown...
    }
}
```

نکته ی بسیار مهم اینکه از طریق Message Handler نه تنها به HTTP Request دسترسی و امکان کنترل رو داریم بلکه HTTP Response ها نیز پس از تولید به ترتیب به این Message Handler ها رسیده و امکان اعمال کنترل و پردازش بر روی Response نیز می باشد. طبیعتا ترتیب دریافت Response عکس ترتیب دریافت Request توسط Message Handler هاست. مثلا در کد بالا ابتدا Message Handler 1 و سپس ۲ HTTP Request Message رو دریافت می کنند؛ اما پس از ارسال درخواست به action مورد نظر و ایجاد HTTP Response ابتدا Message Handler2 و سپس ۱ این پاسخ رو دریافت و نهایت اون رو به کلاینت بر می گردونند.

در ادامه چند نمونه و سناریو مختلف استفاده از Message Handler اورده می شود.

Checking for an API Key

```
public class ApiKeyHandler : DelegatingHandler
{
    public string Key { get; set; }

    public ApiKeyHandler(string key)
    {
        this.Key = key;
    }

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
```

```
        if (!ValidateKey(request))
        {
            var response = new HttpResponseMessage(HttpStatusCode.Forbidden);
            var tsc = new TaskCompletionSource<HttpResponseMessage>();
            tsc.SetResult(response);
            return tsc.Task;
        }
        return base.SendAsync(request, cancellationToken);
    }

    private bool ValidateKey(HttpRequestMessage message)
    {
        var query = message.RequestUri.ParseQueryString();
        string key = query["key"];
        return (key == Key);
    }
}
```

Adding a Custom Response Header

```
// .Net 4.5
public class CustomHeaderHandler : DelegatingHandler
{
    async protected override Task<HttpResponseMessage> SendAsync(
            HttpRequestMessage request, CancellationToken cancellationToken)
    {
        HttpResponseMessage response = await base.SendAsync(request,
cancellationToken);
        response.Headers.Add("X-Custom-Header", "This is my custom header.");
        return response;
    }
}
```

اعمال **Message Handler** بر اساس **Route Table**

هندلرهایی که به روش گفته شده در بالا Register می شوند(یعنی درون **HttpConfiguration.MessageHandlers**) بصورت global اعمال خواهند شد.

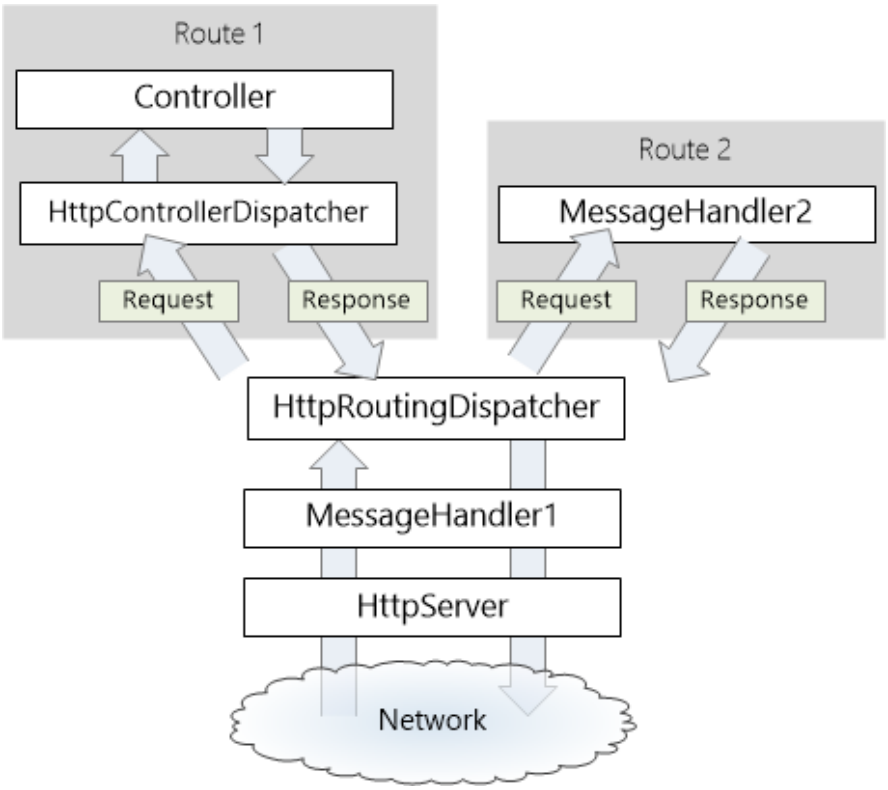
اما این امکان نیز وجود دارد که هندلر مورد نظر رو فقط در صورتی که HTTP Request Message با route template خاصی match باشد اعمال شود. برای اینکار می توان به هنگام تعریف route template بصورت زیر عمل کرد.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "Route1",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        config.Routes.MapHttpRoute(
            name: "Route2",
            routeTemplate: "api2/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional },
            constraints: null,
            handler: new MessageHandler2()  // per-route message handler
        );

        config.MessageHandlers.Add(new MessageHandler1());  // global message handler
    }
}
```

در مثال بالا در صورتی که request URI با route 2 مطابقت داشته باشد در نتیجه request به سمت MessageHandler1 ؛ dispatch خواهد شد.

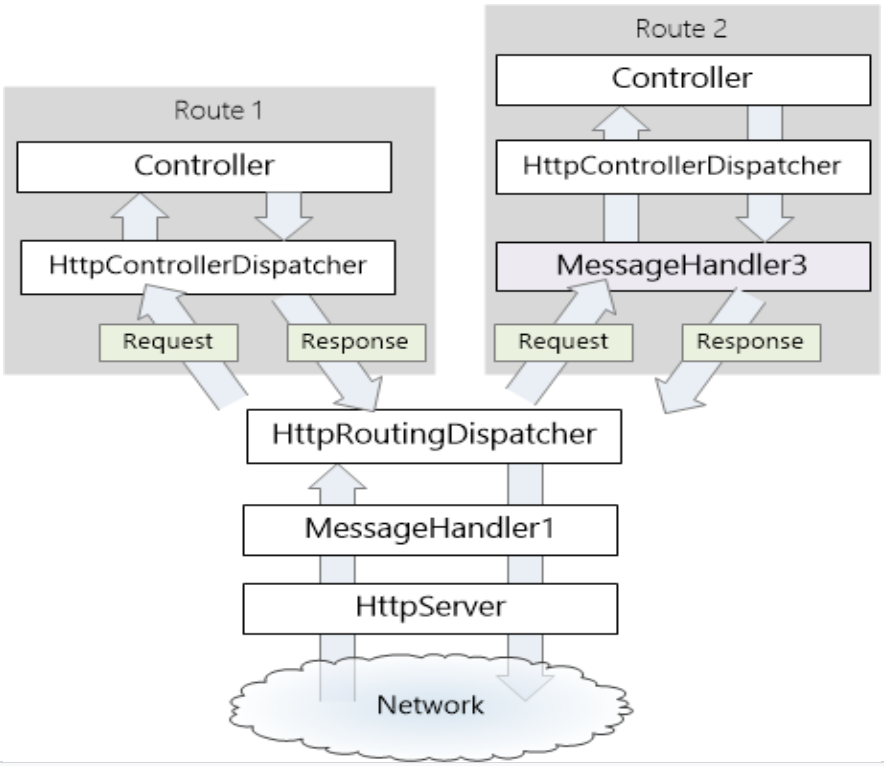


در مثالی که در بالا اشاره شد، همانطور که می بینید MessageHandler2 جایگزین **HttpControllerDispatcher** شده است. و این به شما مکانیزمی خواهد داد که کل Web API Controller mechanism رو بطور کامل کنترل کرده و endpoint مورد نظر خود را بنویسید. در این مثال مورد نظر همانطور که در بالا آورده شده MesageHandler2 چون base.SendAsync رو صدا نمی زد عملا چرخه Pipeline پس از رسیدن به این هندل قطع میشد. پس درخواست های Route هیچگاه به کنترلر نمی رسیدند و خود MessageHandler2؛ HTTPResponseMessage مورد نظر رو ایجاد و به کلاینت بر می گرداند.

نکته: در مورد HTTP Routing Dispatcher و HTTP Controller Dispatcherدر مقاله ی بعدی مفصل صحبت خواهد شد.

نکته(مهم): همانطور که در تصویر بالا مشاهده می کنید MessageHadler1 که بصورت global اعمال شده بسیار زودتر وارد چرخه ی Pipeline شده و به HTTP Request Message مورد نظر دسترسی خواهد داشت. در صورتی که MessageHandler2 پس از HttpRoutingDispatcher اعمال خواهد شد. پس حتما بهنگام استفاده از هندلر مورد نیاز؛ به زمان وارد شدن این هندلر ها در Pipeline نیز توجه نمایید.

و کلام آخر این بخش اینکه per-route message handler را می توان به **HttpControllerDispatcher** نیز واگذار کرد که در این حالت به controller نیز dispatch خواهد شد.




```
// List of delegating handlers.
DelegatingHandler[] handlers = new DelegatingHandler[] {
    new MessageHandler3()
};

// Create a message handler chain with an end-point.
var routeHandlers = HttpClientFactory.CreatePipeline(
    new HttpControllerDispatcher(config), handlers);

config.Routes.MapHttpRoute(
    name: "Route2",
    routeTemplate: "api2/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional },
    constraints: null,
    handler: routeHandlers
);
```

Trying to Be Agile…

Masoud Bahrmai