A low-level Look at the ASP.NET Architecture

Getting Low Level

This article looks at how Web requests flow through the ASP.NET framework from a very low level perspective, from Web Server, through ISAPI all the way up the request handler and your code. See what happens behind the scenes and stop thinking of ASP.NET as a black box.

By Rick Strahl
www.west-wind.com
rstrahl@west-wind.com

Last Update:
August 24, 2008

Other Links:
Download Examples for this article
Leave a Comment or Question

ASP.NET is a powerful platform for building Web applications, that provides a tremendous amount of flexibility and power for building just about any kind of Web application. Most people are familiar only with the high level frameworks like WebForms and WebServices which sit at the very top level of the ASP.NET hierarchy. In this article I'll describe the lower level aspects of ASP.NET and explain how requests move from Web Server to the ASP.NET runtime and then through the ASP.NET Http Pipeline to process requests.

To me understanding the innards of a platform always provides certain satisfaction and level of comfort, as well as insight that helps to write better applications. Knowing what tools are available and how they fit together as part of the whole complex framework makes it easier to find the best solution to a problem and more importantly helps in troubleshooting and debugging of problems when they occur. The goal of this article is to look at ASP.NET from the System level and help understand how requests flow into the ASP.NET processing pipeline. As such we'll look at the core engine and how Web requests end up there. Much of this information is not something that you need to know in your daily work, but it's good to understand how the ASP.NET architecture routes request into your application code that usually sits at a much higher level.

Most people using ASP.NET are familiar with WebForms and WebServices. These high level implementations are abstractions that make it easy to build Web based application logic and ASP.NET is the driving engine that provides the underlying interface to the Web Server and routing mechanics to provide the base for these high level front end services typically used for your applications. WebForms and WebServices are merely two very sophisticated implementations of HTTP Handlers built on top of the core ASP.NET framework.

However, ASP.NET provides much more flexibility from a lower level. The HTTP Runtime and the request pipeline provide all the same power that went into building the WebForms and WebService implementations – these implementations were actually built with .NET managed code. And all of that same functionality is available to you, should you decide you need to build a custom platform that sits at a level a little lower than WebForms.

WebForms are definitely the easiest way to build most Web interfaces, but if you're building custom content handlers, or have special needs for processing the incoming or outgoing content, or you need to build a custom application server interface to another application, using these lower level handlers or modules can provide better performance and more control over the actual request process. With all the power that the high level implementations of WebForms and WebServices provide they also add quite a bit of overhead to requests that you can bypass by working at a lower level.

What is ASP.NET

Let's start with a simple definition: What is ASP.NET? I like to define ASP.NET as follows:

> **ASP.NET is a sophisticated engine using Managed Code for front to back processing of Web Requests.**
>
> **It's much more than just WebForms and Web Services...**

ASP.NET is a request processing engine. It takes an incoming request and passes it through its internal pipeline to an end point where you as a developer can attach code to process that request. This engine is actually completely separated from HTTP or the Web Server. In fact, the HTTP Runtime is a component that you can host in your own applications outside of IIS or any server side application altogether. For example, you can host the ASP.NET runtime in a Windows form (check out http://www.west-wind.com/presentations/aspnetruntime/aspnetruntime.asp for more detailed information on runtime hosting in Windows Forms apps).

The runtime provides a complex yet very elegant mechanism for routing requests through this pipeline. There are a number of interrelated objects, most of which are extensible either via subclassing or through event interfaces at almost every level of the process, so the framework is highly extensible. Through this mechanism it's possible to hook into very low level interfaces such as the caching, authentication and authorization. You can even filter content by pre or post processing requests or simply route incoming requests that match a specific signature directly to your code or

By Rick Strahl
Rick Strahl is president of West Wind Technologies on Maui, Hawaii. The company specializes in Web and distributed application development and tools with focus on .NET, Visual Studio and Visual FoxPro. Rick is author of West Wind Web Connection, a powerful and widely used Web application framework and West Wind HTML Help Builder and West Wind Web Store. He's also a C# MVP, a frequent speaker at international developer conferences and a frequent contributor to magazines and books. He is co-publisher of Code magazine. For more information please visit: http://www.west-wind.com/ or contact Rick at rstrahl@west-wind.com.

another URL. There are a lot of different ways to accomplish the same thing, but all of the approaches are straightforward to implement, yet provide flexibility in finding the best match for performance and ease of development.

> **The entire ASP.NET engine was completely built in managed code and all extensibility is provided via managed code extensions.**

The entire ASP.NET engine was completely built in managed code and all of the extensibility functionality is provided via managed code extensions. This is a testament to the power of the .NET framework in its ability to build sophisticated and very performance oriented architectures. Above all though, the most impressive part of ASP.NET is the thoughtful design that makes the architecture easy to work with, yet provides hooks into just about any part of the request processing.

With ASP.NET you can perform tasks that previously were the domain of ISAPI extensions and filters on IIS – with some limitations, but it's a lot closer than say ASP was. ISAPI is a low level Win32 style API that had a very meager interface and was very difficult to work for sophisticated applications. Since ISAPI is very low level it also is very fast, but fairly unmanageable for application level development. So, ISAPI has been mainly relegated for some time to providing bridge interfaces to other application or platforms. But ISAPI isn't dead by any means. In fact, ASP.NET on Microsoft platforms interfaces with IIS through an ISAPI extension that hosts .NET and through it the ASP.NET runtime. ISAPI provides the core interface from the Web Server and ASP.NET uses the unmanaged ISAPI code to retrieve input and send output back to the client. The content that ISAPI provides is available via common objects like HttpRequest and HttpResponse that expose the unmanaged data as managed objects with a nice and accessible interface.

From Browser to ASP.NET

Let's start at the beginning of the lifetime of a typical ASP.NET Web Request. A request starts on the browser where the user types in a URL, clicks on a hyperlink or submits an HTML form (a POST request). Or a client application might make call against an ASP.NET based Web Service, which is also serviced by ASP.NET. On the server side the Web Server – Internet Information Server 5 or 6 – picks up the request. At the lowest level ASP.NET interfaces with IIS through an ISAPI extension. With ASP.NET this request usually is routed to a page with an *.aspx* extension, but how the process works depends entirely on the implementation of the HTTP Handler that is set up to handle the specified extension. In IIS *.aspx* is mapped through an 'Application Extension' (aka. as a script map) that is mapped to the ASP.NET ISAPI dll - *aspnet_isapi.dll*. Every request that fires ASP.NET must go through an extension that is registered and points at *aspnet_isapi.dll*.

Depending on the extension ASP.NET routes the request to an appropriate handler that is responsible for picking up requests. For example, the *.asmx* extension for Web Services routes requests not to a page on disk but a specially attributed class that identifies it as a Web Service implementation. Many other handlers are installed with ASP.NET and you can also define your own. All of these *HttpHandlers* are mapped to point at the ASP.NET ISAPI extension in IIS, and configured in web.config to get routed to a specific HTTP Handler implementation. Each handler, is a .NET class that handles a specific extension which can range from simple Hello World behavior with a couple of lines of code, to very complex handlers like the ASP.NET Page or Web Service implementations. For now, just understand that an extension is the basic mapping mechanism that ASP.NET uses to receive a request from ISAPI and then route it to a specific handler that processes the request.

> **ISAPI is the first and highest performance entry point into IIS for custom Web Request handling.**

The ISAPI Connection

ISAPI is a low level unmanged Win32 API. The interfaces defined by the ISAPI spec are very simplistic and optimized for performance. They are very low level – dealing with raw pointers and function pointer tables for callbacks - but they provide he lowest and most performance oriented interface that developers and tool vendors can use to hook into IIS. Because ISAPI is very low level it's *not* well suited for building application level code, and ISAPI tends to be used primarily as a bridge interface to provide Application Server type functionality to higher level tools. For example, ASP and ASP.NET both are layered on top of ISAPI as is Cold Fusion, most Perl, PHP and JSP implementations running on IIS as well as many third party solutions such as my own Web Connection framework for Visual FoxPro. ISAPI is an excellent tool to provide the high performance plumbing interface to higher level applications, which can then abstract the information that ISAPI provides. In ASP and ASP.NET, the engines abstract the information provided by the ISAPI interface in the form of objects like Request and Response that read their content out of the ISAPI request information. Think of ISAPI as the plumbing. For ASP.NET the ISAPI dll is very lean and acts merely as a routing mechanism to pipe the inbound request into the ASP.NET runtime. All the heavy lifting and processing, and even the request thread management happens inside of the ASP.NET engine and your code.

As a protocol ISAPI supports both ISAPI extensions and ISAPI Filters. Extensions are a request handling interface and provide the logic to handle input and output with the Web Server – it's essentially a transaction interface. ASP and ASP.NET are implemented as ISAPI extensions. ISAPI filters are hook interfaces that allow the ability to look at EVERY request that comes into IIS and to modify the content or change the behavior of functionalities like Authentication. Incidentally ASP.NET maps ISAPI-like functionality via two concepts: Http Handlers (extensions) and Http Modules (filters). We'll look at these later in more detail.

ISAPI is the initial code point that marks the beginning of an ASP.NET request. ASP.NET maps various extensions to its ISAPI extension which lives in the .NET Framework directory:

<.NET FrameworkDir>\aspnet_isapi.dll

IIS 6 Wildcard Application Mappings

If you have an ASP.NET application that needs to handle EVERY single request to a virtual directory (or Web Server if configured in the root directory), IIS 6 introduces a new concept of Wildcard Application Mappings. An ISAPI extension that is mapped to a wildcard fires on every request against the server regardless of extension. This means every page runs through the extension. This is a powerful feature as you can use this mechanism to create custom vanity Urls and unix style URL patters that don't use filenames at all. However, be careful with this setting because it passes everything through your application including static htm files, images, style sheets etc.

You can interactively see these mapping in the IIS Service manager as shown in Figure 1. Look at the root of the Web Site and the Home Directory tab, then Configuration | Mappings.

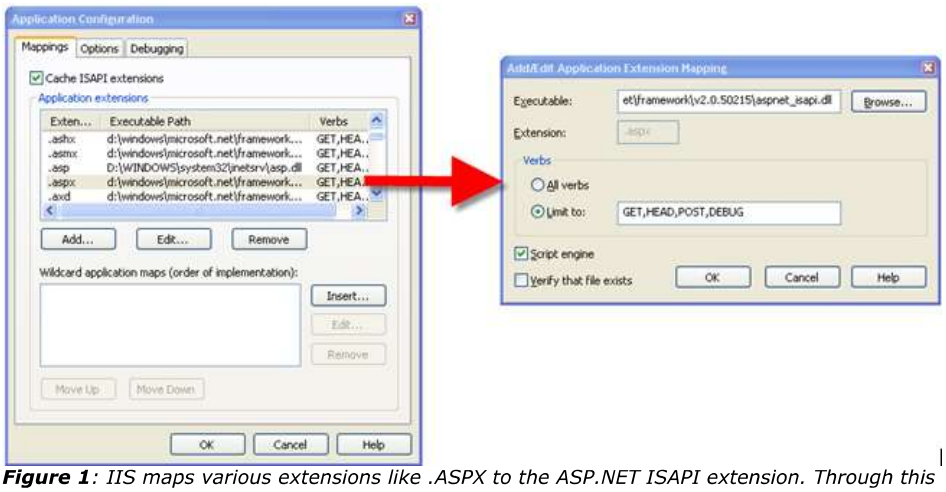

***Figure 1**: IIS maps various extensions like .ASPX to the ASP.NET ISAPI extension. Through this mechanism requests are routed into ASP.NET's processing pipeline at the Web Server level.*

You shouldn't set these extensions manually as .NET requires a number of them. Instead use the **aspnet_regiis.exe** utility to make sure that all the various scriptmaps get registered properly:

```
cd <.NetFrameworkDirectory>
aspnet_regiis - i
```

This will register the particular version of the ASP.NET runtime for the entire Web site by registering the scriptmaps and setting up the client side scripting libraries used by the various controls for uplevel browsers. Note that it registers the particular version of the CLR that is installed in the above directory. Options on aspnet_regiis let you configure virtual directories individually. Each version of the .NET framework has its own version of aspnet_regiis and you need to run the appropriate one to register a site or virtual directory for a specific version of the .NET framework. Starting with ASP.NET 2.0, an IIS ASP.NET configuration page lets you pick the .NET version interactively in the IIS management console.

IIS 5 and 6 work differently

When a request comes in, IIS checks for the script map and routes the request to the aspnet_isapi.dll. The operation of the DLL and how it gets to the ASP.NET runtime varies significantly between IIS 5 and 6. Figure 2 shows a rough overview of the flow.

In IIS 5 hosts aspnet_isapi.dll directly in the inetinfo.exe process or one of its isolated worker processes if you have isolation set to medium or high for the Web or virtual directory. When the first ASP.NET request comes in the DLL will spawn a new process in another EXE – *aspnet_wp.exe* – and route processing to this spawned process. This process in turn loads and hosts the .NET runtime. Every request that comes into the ISAPI DLL then routes to this worker process via Named Pipe calls.

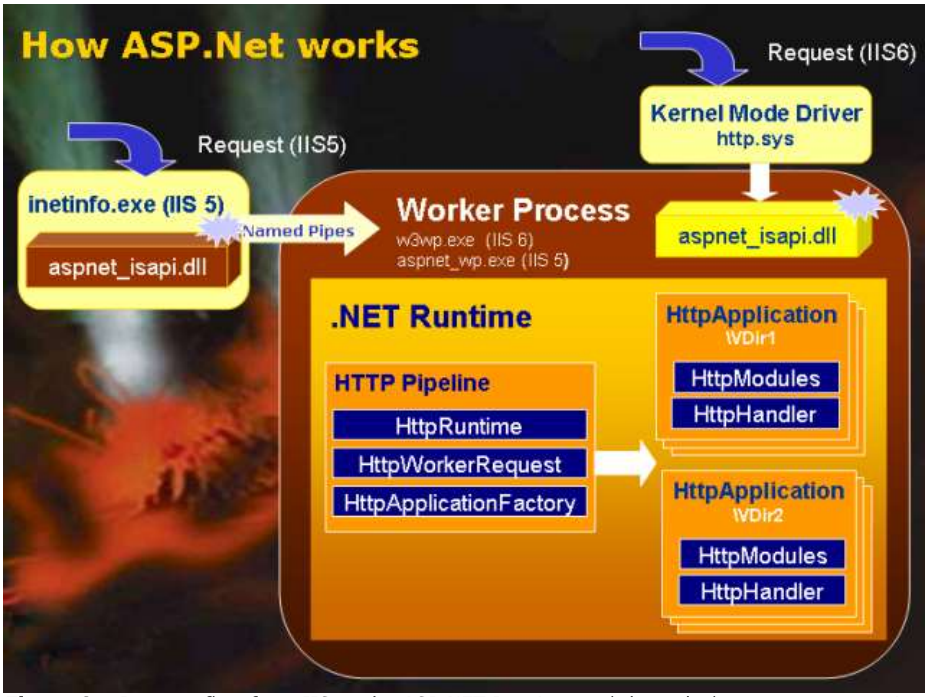

***Figure 2** – Request flow from IIS to the ASP.NET Runtime and through the request processing pipeline from a high level. IIS 5 and IIS 6 interface with ASP.NET in different ways but the overall process once it reaches the ASP.NET Pipeline is the same.*

IIS6, unlike previous servers, is fully optimized for ASP.NET

IIS 6 – Viva the Application Pool

IIS 6 changes the processing model significantly in that IIS no longer hosts any foreign executable code like ISAPI extensions directly. Instead IIS 6 **always** creates a separate worker process – an *Application Pool* – and all processing occurs inside of this process, including execution of the ISAPI dll. Application Pools are a big improvement for IIS 6, as they allow very granular control over what executes in a given process. Application Pools can be configured for every virtual directory or the entire Web site, so you can isolate every Web application easily into its own process that will be completely isolated from any other Web application running on the same machine. If one process dies it will not affect any others at least from the Web processing perspective.

In addition, Application Pools are highly configurable. You can configure their execution security environment by setting an execution impersonation level for the pool which allows you to customize the rights given to a Web application in that same granular fashion. One big improvement for ASP.NET is that the Application Pool replaces most of the ProcessModel entry in machine.config. This entry was difficult to manage in IIS 5, because the settings were global and could not be overridden in an application specific web.config file. When running IIS 6, the ProcessModel setting is mostly ignored and settings are instead read from the Application Pool. I say mostly – some settings, like the size of the ThreadPool and IO threads still are configured through this key since they have no equivalent in the Application Pool settings of the server.

Because Application Pools are external executables these executables can also be easily monitored and managed. IIS 6 provides a number of health checking, restarting and timeout options that can detect and in many cases correct problems with an application. Finally IIS 6's Application Pools don't rely on COM+ as IIS 5 isolation processes did which has improved performance and stability especially for applications that need to use COM objects internally.

Although IIS 6 application pools are separate EXEs, they are highly optimized for HTTP operations by directly communicating with a kernel mode HTTP.SYS driver. Incoming requests are directly routed to the appropriate application pool. InetInfo acts merely as an Administration and configuration service – most interaction actually occurs directly between HTTP.SYS and the Application Pools, all of which translates into a more stable and higher performance environment over IIS 5. This is especially true for static content and ASP.NET applications.

An IIS 6 application pool also has intrinsic knowledge of ASP.NET and ASP.NET can communicate with new low level APIs that allow direct access to the HTTP Cache APIs which can offload caching from the ASP.NET level directly into the Web Server's cache.

In IIS 6, ISAPI extensions run in the Application Pool worker process. The .NET Runtime also runs in this same process, so communication between the ISAPI extension and the .NET runtime happens in-process which is inherently more efficient than the named pipe interface that IIS 5 must use. Although the IIS hosting models are very different the actual interfaces into managed code are very similar – only the process in getting the request routed varies a bit.

The ISAPIRuntime.ProcessRequest() method is the first entry point into ASP.NET

Getting into the .NET runtime

The actual entry points into the .NET Runtime occur through a number of undocumented classes and interfaces. Little is known about these interfaces outside of Microsoft, and Microsoft folks are not eager to talk about the details, as they deem this an implementation detail that has little effect on developers building applications with ASP.NET.

The worker processes ASPNET_WP.EXE (IIS5) and W3WP.EXE (IIS6) host the .NET runtime and the ISAPI DLL calls into small set of unmanged interfaces via low level COM that eventually forward calls to an instance subclass of the ISAPIRuntime class. The first entry point to the runtime is the undocumented ISAPIRuntime class which exposes the IISAPIRuntime interface via COM to a caller. These COM interfaces low level IUnknown based interfaces that are meant for internal calls from the ISAPI extension into ASP.NET. Figure 3 shows the interface and call signatures for the IISAPIRuntime interface as shown in Lutz Roeder's excellent .NET Reflector tool (http://www.aisto.com/roeder/dotnet/). Reflector an assembly viewer and disassembler that makes it very easy to look at medadata and disassembled code (in IL, C#, VB) as shown in Figure 3. It's a great way to explore the bootstrapping process.

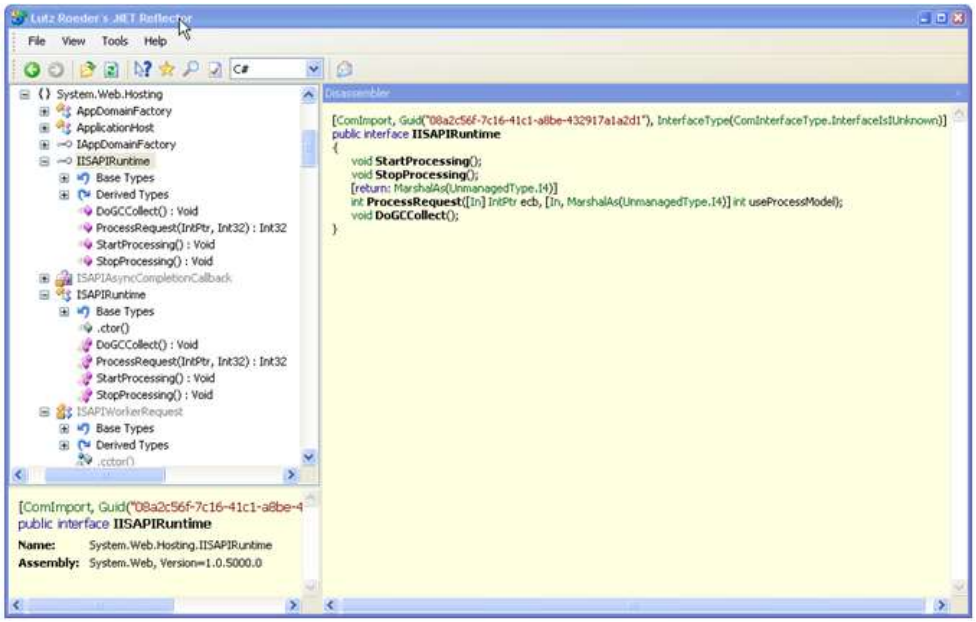

Figure 3 *– If you want to dig into the low level interfaces open up Reflector, and point at the System.Web.Hosting namespace. The entry point to ASP.NET occurs through a managed COM Interface called from the ISAPI dll, that receives an unmanaged pointer to the ISAPI ECB. The ECB contains has access to the full ISAPI interface to allow retrieving request data and sending back to IIS.*

The IISAPIRuntime interface acts as the interface point between the unmanaged code coming from the ISAPI extension (directly in IIS 6 and indirectly via the Named Pipe handler in IIS 5). If you take a look at this class you'll find a ProcessRequest method with a signature like this:

```
[return: MarshalAs(UnmanagedType.I4)]
int ProcessRequest([In] IntPtr ecb,
                   [In, MarshalAs(UnmanagedType.I4)] int useProcessModel);
```

The ecb parameter is the ISAPI Extension Control Block (ECB) which is passed as an unmanaged resource to ProcessRequest. The method then takes the ECB and uses it as the base input and output interface used with the Request and Response objects. An ISAPI ECB contains all low level request information including server variables, an input stream for form variables as well as an output stream that is used to write data back to the client. The single ecb reference basically provides access to all of the functionality an ISAPI request has access to and ProcessRequest is the entry and exit point where this resource initially makes contact with managed code.

The ISAPI extension runs requests asynchronously. In this mode the ISAPI extension immediately returns on the calling worker process or IIS thread, but keeps the ECB for the current request alive. The ECB then includes a mechanism for letting ISAPI know when the request is complete (via ecb.ServerSupportFunction) which then releases the ECB. This asynchronous processing releases the ISAPI worker thread immediately, and offloads processing to a separate thread that is managed by ASP.NET.

ASP.NET receives this ecb reference and uses it internally to retrieve information about the current request such as server variables, POST data as well as returning output back to the server. The ecb stays alive until the request finishes or times out in IIS and ASP.NET continues to communicate with it until the request is done. Output is written into the ISAPI output stream (ecb.WriteClient()) and when the request is done, the ISAPI extension is notified of request completion to let it know that the ECB can be freed. This implementation is very efficient as the .NET classes essentially act as a fairly thin wrapper around the high performance, unmanaged ISAPI ECB.

Loading .NET – somewhat of a mystery

Let's back up one step here: I skipped over how the .NET runtime gets loaded. Here's where things get a bit fuzzy. I haven't found any documentation on this process and since we're talking about native code there's no easy way to disassemble the ISAPI DLL and figure it out.

My best guess is that the worker process bootstraps the .NET runtime from within the ISAPI extension on the first hit against an ASP.NET mapped extension. Once the runtime exists, the unmanaged code can request an instance of an ISAPIRuntime object for a given virtual path if one doesn't exist yet. Each virtual directory gets its own AppDomain and within that AppDomain the ISAPIRuntime exists from which the bootstrapping process for an individual application starts. Instantiation appears to occur over COM as the interface methods are exposed as COM callable methods.

To create the ISAPIRuntime instance the System.Web.Hosting.AppDomainFactory.Create() method is called when the first request for a specific virtual directory is requested. This starts the 'Application' bootstrapping process. The call receives parameters for type and module name and virtual path information for the application which is used by ASP.NET to create an AppDomain and launch the ASP.NET application for the given virtual directory. This HttpRuntime derived object is created in a new AppDomain. Each virtual directory or ASP.NET application is hosted in a separate AppDomain and they get loaded only as requests hit the particular ASP.NET Application. The ISAPI extension manages these instances of the HttpRuntime objects, and routes inbound requests to the right one based on the virtual path of the request.

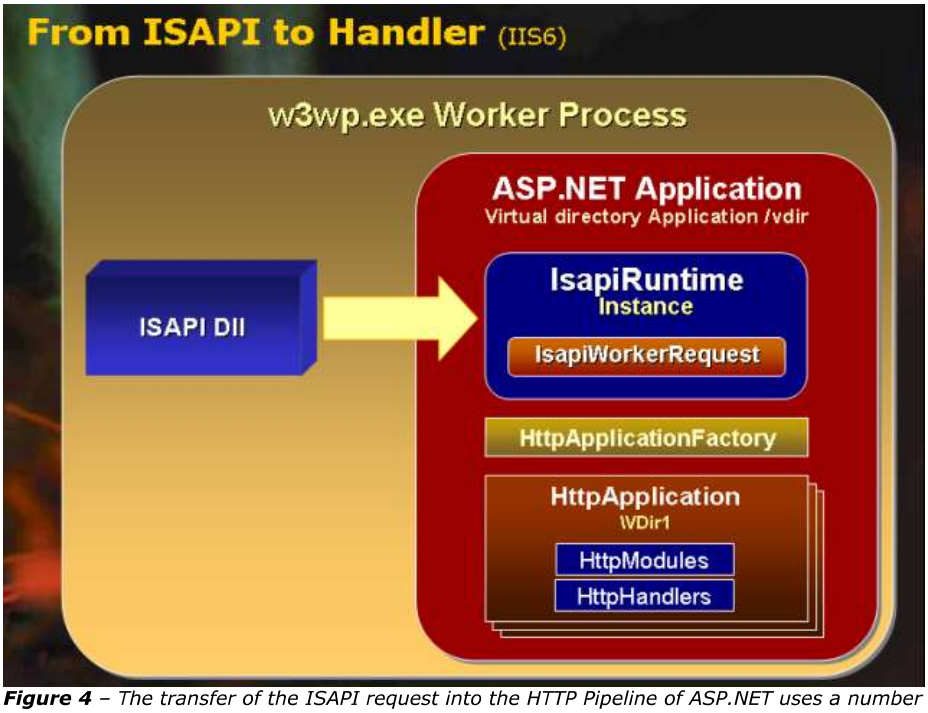

Figure 4 *– The transfer of the ISAPI request into the HTTP Pipeline of ASP.NET uses a number of undocumented classes and interfaces and requires several factory method calls. Each Web Application/Virtual runs in its own AppDomain with the caller holding a reference to an IISAPIRuntime interface that triggers the ASP.NET request processing.*

Back in the runtime

At this point we have an instance of ISAPIRuntime active and callable from the ISAPI extension. Once the runtime is up and running the ISAPI code calls into the ISAPIRuntime.ProcessRequest() method which is the real entry point into the ASP.NET Pipeline. The flow from there is shown in Figure 4.

Remember ISAPI is multi-threaded so requests will come in on multiple threads through the reference that was returned by ApplicationDomainFactory.Create(). Listing 1 shows the disassembled code from the IsapiRuntime.ProcessRequest method that receives an ISAPI ecb object and server type as parameters. The method is thread safe, so multiple ISAPI threads can safely call this single returned object instance simultaneously.

Listing 1: The Process request method receives an ISAPI Ecb and passes it on to the Worker request

```
public int ProcessRequest(IntPtr ecb, int iWRType)
{
      HttpWorkerRequest request1 = ISAPIWorkerRequest.CreateWorkerRequest(ecb,
iWRType);

      string text1 = request1.GetAppPathTranslated();
      string text2 = HttpRuntime.AppDomainAppPathInternal;
      if (((text2 == null) || text1.Equals(".")) ||
         (string.Compare(text1, text2, true, CultureInfo.InvariantCulture) == 0))
      {
            HttpRuntime.ProcessRequest(request1);
            return 0;
      }

      HttpRuntime.ShutdownAppDomain("Physical application path changed from " +
            text2 + " to " + text1);
      return 1;
}
```

The actual code here is not important, and keep in mind that this is disassembled internal framework code that you'll never deal with directly and that might change in the future. It's meant to demonstrate what's happening behind the scenes. ProcessRequest receives the unmanaged ECB reference and passes it on to the ISAPIWorkerRequest object which is in charge of creating the Request Context for the current request as shown in Listing 2.

The System.Web.Hosting.ISAPIWorkerRequest class is an abstract subclass of HttpWorkerRequest, whose job it is to create an abstracted view of the input and output that serves as the input for the Web application. Notice another factory method here: CreateWorkerRequest, which as a second parameter receives the type of worker request object to create. There are three different versions: ISAPIWorkerRequestInProc, ISAPIWorkerRequestInProcForIIS6, ISAPIWorkerRequestOutOfProc. This object is created on each incoming hit and serves as the basis for the Request and Response objects which will receive their data and streams from the data provided by the WorkerRequest.

The abstract HttpWorkerRequest class is meant to provide a highlevel abstraction around the low level interfaces so that regardless of where the data comes from, whether it's a CGI Web Server, the Web Browser Control or some custom mechanism you use to feed the data to the HTTP Runtime. The key is that ASP.NET can retrieve the information consistently.

In the case of IIS the abstraction is centered around an ISAPI ECB block. In our request processing, ISAPIWorkerRequest hangs on to the ISAPI ECB and retrieves data from it as needed. Listing 2 shows how the query string value is retrieved for example.

Listing 2: An ISAPIWorkerRequest method that uses the unmanged

What's new in ASP.NET 2.0

ASP.NET 2.0 hasn't changed the underlying architecture much. The main features that are new are that the HttpApplication object gains a number of new events – mostly pre and post event hooks – that make the Application event pipeline even more granular. ASP.NET 2.0 also supports a new ISAPI function - HSE_REQ_EXEC_URL – that allows redirecting to another URL internally from within the ASP.NET process. This makes it possible for ASP.NET to be set up as a wildcard extension in IIS and process every request and then either handle the request directly with an HTTP handler or fall through to a new DefaultHttpHandler. DefaultHttpHandler then internally calls back to ISAPI to execute the original URL. This allows ASP.NET to handle things like Authentication and Logins prior to other pages like ASP to be fired.

```
// *** Implemented in ISAPIWorkerRequest
public override byte[] GetQueryStringRawBytes()
{
      byte[] buffer1 = new byte[this._queryStringLength];
      if (this._queryStringLength > 0)
      {
            int num1 = this.GetQueryStringRawBytesCore(buffer1,
this._queryStringLength);
            if (num1 != 1)
            {
                  throw new HttpException( "Cannot_get_query_string_bytes");
            }
      }
      return buffer1;
}

// *** Implemented in a specific implementation class ISAPIWorkerRequestInProcIIS6
internal override int GetQueryStringCore(int encode, StringBuilder buffer, int size)
{
      if (this._ecb == IntPtr.Zero)
      {
            return 0;
      }
      return UnsafeNativeMethods.EcbGetQueryString(this._ecb, encode, buffer, size);
}
```

ISAPIWorkerRequest implements a high level wrapper method, that calls into lower level *Core* methods, which are responsible for performing the actual access to the unmanaged APIs – or the 'service level implementation'. The *Core* methods are implemented in the specific ISAPIWorkerRequest instance subclasses and thus provide the specific implementation for the environment that it's hosted in. This makes for an easily pluggable environment where additional implementation classes can be provided later as newer Web Server interfaces or other platforms are targeted by ASP.NET. There's also a helper class System.Web.UnsafeNativeMethods. Many of these methods operate on the ISAPI ECB structure performing unmanaged calls into the ISAPI extension.

HttpRuntime, HttpContext, and HttpApplication – Oh my

When a request hits, it is routed to the ISAPIRuntime.ProcessRequest() method. This method in turn calls HttpRuntime.ProcessRequest that does several important things (look at System.Web.HttpRuntime.ProcessRequestInternal with Reflector):

- Create a new HttpContext instance for the request
- Retrieves an HttpApplication Instance
- Calls HttpApplication.Init() to set up Pipeline Events
- Init() fires HttpApplication.ResumeProcessing() which starts the ASP.NET pipeline processing

First a new *HttpContext* object is created and it is passed the ISAPIWorkerRequest that wrappers the ISAPI ECB. The Context is available throughout the lifetime of the request and ALWAYS accessible via the static *HttpContext.Current* property. As the name implies, the HttpContext object represents the context of the currently active request as it contains references to all of the vital objects you typically access during the request lifetime: Request, Response, Application, Server, Cache. At any time during request processing HttpContext.Current gives you access to all of these object.

The HttpContext object also contains a very useful Items collection that you can use to store data that is request specific. The context object gets created at the begging of the request cycle and released when the request finishes, so data stored there in the Items collection is specific only to the current request. A good example use is a request logging mechanism where you want to track start and end times of a request by hooking the Application_BeginRequest and Application_EndRequest methods in Global.asax as shown in Listing 3. HttpContext is your friend – you'll use it liberally if you need data in different parts of the request or page processing.

Listing 3 – Using the HttpContext.Items collection lets you save data between pipeline events

```
protected void Application_BeginRequest(Object sender, EventArgs e)
{
      //*** Request Logging
      if (App.Configuration.LogWebRequests)
            Context.Items.Add("WebLog_StartTime",DateTime.Now);
}

protected void Application_EndRequest(Object sender, EventArgs e)
{
      // *** Request Logging
      if (App.Configuration.LogWebRequests)
      {
            try
            {
                  TimeSpan Span = DateTime.Now.Subtract(
                                    (DateTime) Context.Items["WebLog_StartTime"] );
                  int MiliSecs = Span.TotalMilliseconds;

                  // do your logging
                  WebRequestLog.Log(App.Configuration.ConnectionString,
                                    true,MilliSecs);
      }
}
```

Once the Context has been set up, ASP.NET needs to route your incoming request to the appropriate application/virtual directory by way of an HttpApplication object. Every ASP.NET application must be set up as a Virtual (or Web Root) directory and each of these 'applications' are handled independently.

> **The HttpApplication is like a master of ceremonies – it is where the processing action starts**

Master of your domain: HttpApplication
Each request is routed to an HttpApplication object. The HttpApplicationFactory class creates a pool of HttpApplication objects for your ASP.NET application depending on the load on the application and hands out references for each incoming request. The size of the pool is limited to the setting of the MaxWorkerThreads setting in machine.config's ProcessModel Key, which by default is 20.

The pool starts out with a smaller number though; usually one and it then grows as multiple simulataneous requests need to be processed. The Pool is monitored so under load it may grow to its max number of instances, which is later scaled back to a smaller number as the load drops.

HttpApplication is the outer container for your specific Web application and it maps to the class that is defined in Global.asax. It's the first entry point into the HTTP Runtime that you actually see on a regular basis in your applications. If you look in Global.asax (or the code behind class) you'll find that this class derives directly from HttpApplication:

```
public class Global : System.Web.HttpApplication
```

HttpApplication's primary purpose is to act as the event controller of the Http Pipeline and so its interface consists primarily of events. The event hooks are extensive and include:

- BeginRequest
- AuthenticateRequest
- AuthorizeRequest
- ResolveRequestCache
- AquireRequestState
- PreRequestHandlerExecute
- **...Handler Execution...**
- PostRequestHandlerExecute
- ReleaseRequestState
- UpdateRequestCache
- EndRequest

Each of these events are also implemented in the Global.asax file via empty methods that start with an Application_ prefix. For example, Application_BeginRequest(), Application_AuthorizeRequest(). These handlers are provided for convenience since they are frequently used in applications and make it so that you don't have to explicitly create the event handler delegates.

It's important to understand that each ASP.NET virtual application runs in its own AppDomain and that there inside of the AppDomain multiple HttpApplication instances running simultaneously, fed out of a pool that ASP.NET manages. This is so that multiple requests can process at the same time without interfering with each other.

To see the relationship between the AppDomain, Threads and the HttpApplication check out the code in Listing 4.

Listing 4 – Showing the relation between AppDomain, Threads and HttpApplication instances

```
private void Page_Load(object sender, System.EventArgs e)
{
      // Put user code to initialize the page here
      this.ApplicationId = ((HowAspNetWorks.Global)
                  HttpContext.Current.ApplicationInstance).ApplicationId ;
      this.ThreadId = AppDomain.GetCurrentThreadId();

      this.DomainId = AppDomain.CurrentDomain.FriendlyName;

      this.ThreadInfo = "ThreadPool Thread: " +
            System.Threading.Thread.CurrentThread.IsThreadPoolThread.ToString() +
                        "<br>Thread Apartment: " +
            System.Threading.Thread.CurrentThread.ApartmentState.ToString();

      // *** Simulate a slow request so we can see multiple
      //     requests side by side.
      System.Threading.Thread.Sleep(3000);
}
```

This is part of a demo is provided with your samples and the running form is shown in Figure 5. To check this out run two instances of a browser and hit this sample page and watch the various Ids.

Figure 5 *– You can easily check out how AppDomains, Application Pool instances, and Request Threads interact with each other by running a couple of browser instances simultaneously. When multiple requests fire you'll see the thread and Application ids change, but the AppDomain staying the same.*

You'll notice that the AppDomain ID stays steady while thread and HttpApplication Ids change on most requests, although they likely will repeat. HttpApplications are running out of a collection and are reused for subsequent requests so the ids repeat at times. Note though that Application instance are not tied to a specific thread – rather they are assigned to the active executing thread of the current request.

Threads are served from the .NET ThreadPool and by default are Multithreaded Apartment (MTA) style threads. You can override this apartment state in ASP.NET pages with the ASPCOMPAT="true" attribute in the @Page directive. ASPCOMPAT is meant to provide COM components a safe environment to run in and ASPCOMPAT uses special Single Threaded Apartment (STA) threads to service those requests. STA threads are set aside and pooled separately as they require special handling.

The fact that these HttpApplication objects are all running in the same AppDomain is very important. This is how ASP.NET can guarantee that changes to web.config or individual ASP.NET pages get recognized throughout the AppDomain. Making a change to a value in web.config causes the AppDomain to be shut down and restarted. This makes sure that all instances of HttpApplication see the changes made because when the AppDomain reloads the changes from ASP.NET are re-read at startup. Any static references are also reloaded when the AppDomain so if the application reads values from App Configuration settings these values also get refreshed.

To see this in the sample, hit the ApplicationPoolsAndThreads.aspx page and note the AppDomain Id. Then go in and make a change in web.config (add a space and save). Then reload the page. You'll I find that a new AppDomain has been created.

In essence the Web Application/Virtual completely 'restarts' when this happens. Any requests that are already in the pipeline processing will continue running through the existing pipeline, while any new requests coming in are routed to the new AppDomain. In order to deal with 'hung requests' ASP.NET forcefully shuts down the AppDomain after the request timeout period is up even if requests are still pending. So it's actually possible that two AppDomains exist for the same HttpApplication at a given point in time as the old one's shutting down and the new one is ramping up. Both AppDomains continue to serve their clients until the old one has run out its pending requests and shuts down leaving just the new AppDomain running.

Flowing through the ASP.NET Pipeline

The HttpApplication is responsible for the request flow by firing events that signal your application that things are happening. This occurs as part of the HttpApplication.Init() method (look at System.Web.HttpApplication.InitInternal and HttpApplication.ResumeSteps() with Reflector) which sets up and starts a series of events in succession including the call to execute any handlers. The event handlers map to the events that are automatically set up in global.asax, and they also map any attached *HTTPModules*, which are essentially an externalized event sink for the events that HttpApplication publishes.

Both *HttpModules* and *HttpHandlers* are loaded dynamically via entries in Web.config and attached to the event chain. HttpModules are actual event handlers that hook specific HttpApplication events, while HttpHandlers are an end point that gets called to handle 'application level request processing'.

Both Modules and Handlers are loaded and attached to the call chain as part of the HttpApplication.Init() method call. Figure 6 shows the various events and when they happen and which parts of the pipeline they affect.

Watch out for Response.End with Modules

When creating HttpModules or implementing the event hooks in Global.asax, be careful when you call Response.End or Appplication.CompleteRequest. Both of these methods terminate the current request and stop further events from firing on the HTTP pipeline and immediately return control back to the Web server. This can bite you when you have things like logging or content manipulators running at the end of the processing chain where they don't get fired. For example, the logging sample shown in the article would fail to write the log entry because EndRequest would never fire if Response.End() was called.

Asynchronous Http Handlers

In this article I've talked most about synchronous processing, but the ASP.NET Runtime also supports Aysnchronous operation via Asynchronous HTTP handlers. These handlers automatically offload processing to separate thread pool threads and freeing the main ASP.NET threads to process additional requests. Unfortunately in Version 1.x of

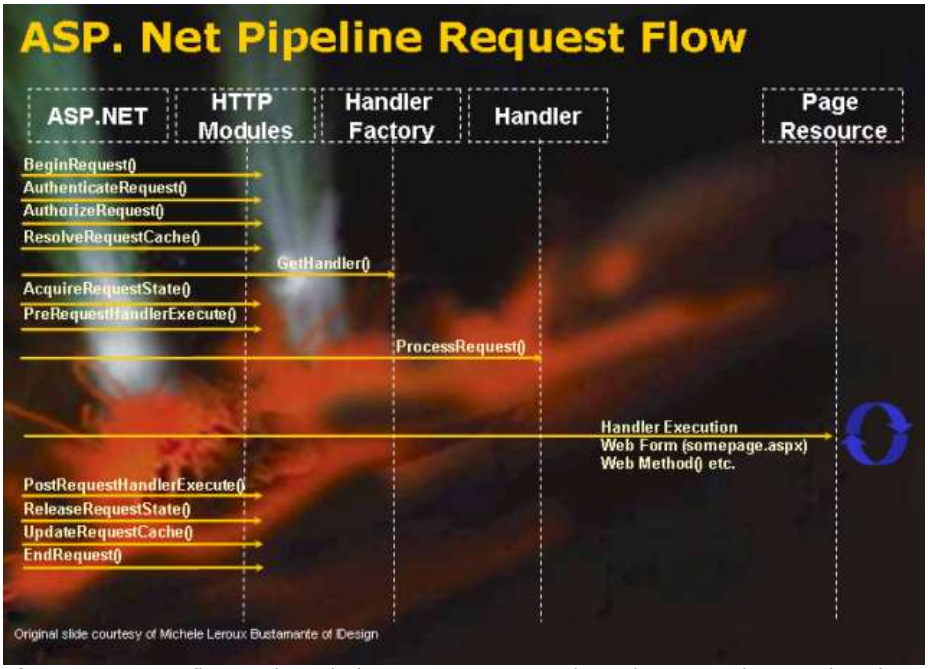

Figure 6 *– Events flowing through the ASP.NET HTTP Pipeline. The HttpApplication object's events drive requests through the pipeline. Http Modules can intercept these events and override or enhance existing functionality.*

HttpContext, HttpModules and HttpHandlers

The HttpApplication itself knows nothing about the data being sent to the application – it is a merely messaging object that communicates via events. It fires events and passes information via the HttpContext object to the called methods. The actual state data for the current request is maintained in the HttpContext object mentioned earlier. It provides all the request specific data and follows each request from beginning to end through the pipeline. Figure 7 shows the flow through ASP.NET pipeline. Notice the Context object which is your compadre from beginning to end of the request and can be used to store information in one event method and retrieve it in a later event method.

Once the pipeline is started, HttpApplication starts firing events one by one as shown in Figure 6. Each of the event handlers is fired and if events are hooked up those handlers execute and perform their tasks. The main purpose of this process is to eventually call the HttpHandler hooked up to a specific request. Handlers are the core processing mechanism for ASP.NET requests and usually the place where any application level code is executed. Remember that the ASP.NET Page and Web Service frameworks are implemented as HTTPHandlers and that's where all the core processing of the request is handled. Modules tend to be of a more core nature used to prepare or post process the Context that is delivered to the handler. Typical default handlers in ASP.NET are Authentication, Caching for pre-processing and various encoding mechanisms on post processing.

There's plenty of information available on HttpHandlers and HttpModules so to keep this article a reasonable length I'm going to provide only a brief overview of handlers.

HttpModules

As requests move through the pipeline a number of events fire on the HttpApplication object. We've already seen that these events are published as event methods in Global.asax. This approach is application specific though which is not always what you want. If you want to build generic HttpApplication event hooks that can be plugged into any Web applications you can use HttpModules which are reusable and don't require application specific code except for an entry in web.config.

Modules are in essence filters – similar in functionality to ISAPI filters at the ASP.NET request level. Modules allow hooking events for EVERY request that pass through the ASP.NET HttpApplication object. These modules are stored as classes in external assemblies that are configured in web.config and loaded when the Application starts. By implementing specific interfaces and methods the module then gets hooked up to the HttpApplication event chain. Multiple HttpModules can hook the same event and event ordering is determined by the order they are declared in Web.config. Here's what a handler definition looks like in Web.config:

```
<configuration>
  <system.web>
    <httpModules>
     <add name= "BasicAuthModule"
      type="HttpHandlers.BasicAuth,WebStore" />
    </httpModules>
  </system.web>
</configuration>
```

Note that you need to specify a full typename and an assembly name without the DLL extension.

Modules allow you look at each incoming Web request and perform an action based on the events that fire. Modules are great to modify request or response content, to provide custom authentication or otherwise provide pre or post processing to every request that occurs against ASP.NET in a particular application. Many of ASP.NET's features like the Authentication and Session engines are implemented as HTTP Modules.

While HttpModules feel similar to ISAPI Filters in that they look at *every* request in that comes through an ASP.NET Application, they are limited to looking at requests mapped to a single specific ASP.NET application or virtual directory and then only against requests that are mapped to ASP.NET.

.NET the offloading occurs onto threads in the same pool of threads that are serving requests so little is actually gained by this feature. To truly build asynchronous behavior you have to create your own threads and manage them with callback handlers.

The current release of ASP.NET 2.0 Beta 2 includes some improvements to the Asynchronous Handlers both for the IHttpHandlerAsync interface and the Page class that provide better performance, but it's unclear whether these features will make it into the final release at this time

Thus you can look at all ASPX pages or any of the other custom extensions that are mapped to this application. You cannot however look at standard .HTM or image files unless you explicitly map the extension to the ASP.NET ISAPI dll by adding an extension as shown in Figure 1. A common use for a module might be to filter content to JPG images in a special folder and display a 'SAMPLE' overlay ontop of every image by drawing ontop of the returned bitmap with GDI+.

Implementing an HTTP Module is very easy: You must implement the IHttpModule interface which contains only two methods Init() and Dispose(). The event parameters passed include a reference to the HTTPApplication object, which in turn gives you access to the HttpContext object. In these methods you hook up to HttpApplication events. For example, if you want to hook the AuthenticateRequest event with a module you would do what's shown in Listing 5.

Listing 5: The basics of an HTTP Module are very simple to implement

```
public class BasicAuthCustomModule : IHttpModule
{

      public void Init(HttpApplication application)
      {
            // *** Hook up any HttpApplication events
            application.AuthenticateRequest +=
                new EventHandler(this.OnAuthenticateRequest);
      }
      public void Dispose() { }

      public void OnAuthenticateRequest(object source, EventArgs eventArgs)
      {
            HttpApplication app = (HttpApplication) source;
            HttpContext Context = HttpContext.Current;
            … do what you have to do…                                    }
}
```

Remember that your Module has access the HttpContext object and from there to all the other intrinsic ASP.NET pipeline objects like Response and Request, so you can retrieve input etc. But keep in mind that certain things may not be available until later in the chain.

You can hook multiple events in the Init() method so your module can manage multiple functionally different operations in one module. However, it's probably cleaner to separate differing logic out into separate classes to make sure the module is modular. <g> In many cases functionality that you implement may require that you hook multiple events – for example a logging filter might log the start time of a request in Begin Request and then write the request completion into the log in EndRequest.

Watch out for one important gotcha with HttpModules and HttpApplication events: Response.End() or HttpApplication.CompleteRequest() will shortcut the HttpApplication and Module event chain. See the sidebar "Watch out for Response.End() " for more info.

HttpHandlers

Modules are fairly low level and fire against *every* inbound request to the ASP.NET application. Http Handlers are more focused and operate on a specific request mapping, usually a page extension that is mapped to the handler.

Http Handler implementations are very basic in their requirements, but through access of the HttpContext object a lot of power is available. Http Handlers are implemented through a very simple IHttpHandler interface (or its asynchronous cousin, IHttpAsyncHandler) which consists of merely a single method – ProcessRequest() – and a single property IsReusable. The key is ProcessRequest() which gets passed an instance of the HttpContext object. This single method is responsible for handling a Web request start to finish.

Single, simple method? Must be too simple, right? Well, simple interface, but not simplistic in what's possible! Remember that WebForms and WebServices are both implemented as Http Handlers, so there's a lot of power wrapped up in this seemingly simplistic interface. The key is the fact that by the time an Http Handler is reached all of ASP.NET's internal objects are set up and configured to start processing of requests. The key is the HttpContext object, which provides all of the relevant request functionality to retireve input and send output back to the Web Server.

For an HTTP Handler all action occurs through this single call to ProcessRequest(). This can be as simple as:

```
public void ProcessRequest(HttpContext context)
{
   context.Response.Write("Hello World");
}
```

to a full implementation like the WebForms Page engine that can render complex forms from HTML templates. The point is that it's up to you to decide of what you want to do with this simple, but powerful interface!

Because the Context object is available to you, you get access to the Request, Response, Session and Cache objects, so you have all the key features of an ASP.NET request at your disposal to figure out what users submitted and return content you generate back to the client. Remember the Context object – it's your friend throughout the lifetime of an ASP.NET request!

The key operation of the handler should be eventually write output into the Respone object or more specifically the Response object's OutputStream. This output is what actually gets sent back to the client. Behind the scenes the ISAPIWorkerRequest manages sending the OutputStream back into the ISAPI ecb.WriteClient method that actually performs the IIS output generation.

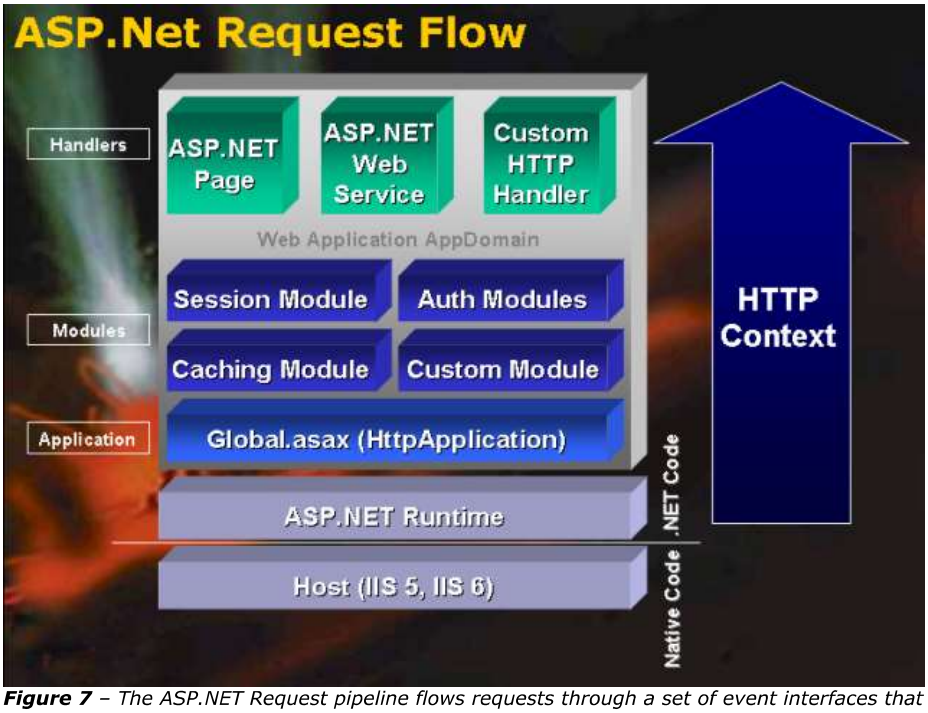

Figure 7 *– The ASP.NET Request pipeline flows requests through a set of event interfaces that provide much flexibility. The Application acts as the hosting container that loads up the Web application and fires events as requests come in and pass through the pipeline. Each request follows a common path through the Http Filters and Modules configured. Filters can examine each request going through the pipeline and Handlers allow implementation of application logic or application level interfaces like Web Forms and Web Services. To provide Input and Output for the application the Context object provides request specific information throughout the entire process.*

WebForms implements an Http Handler with a much more high level interface on top of this very basic framework, but eventually a WebForm's Render() method simply ends up using an HtmlTextWriter object to write its final final output to the context.Response.OutputStream. So while very fancy, ultimately even a high level tool like Web forms is just a high level abstraction ontop of the Request and Response object.

You might wonder at this point whether you need to deal with Http Handlers at all. After all WebForms provides an easily accessible Http Handler implementation, so why bother with something a lot more low level and give up that flexibility?

WebForms are great for generating complex HTML pages and business level logic that requires graphical layout tools and template backed pages. But the WebForms engine performs a lot of tasks that are overhead intensive. If all you want to do is read a file from the system and return it back through code it's much more efficient to bypass the Web Forms Page framework and directly feed the file back. If you do things like Image Serving from a Database there's no need to go into the Page framework – you don't need templates and there surely is no Web UI that requires you to capture events off an Image served.
There's no reason to set up a page object and session and hook up Page level events – all of that stuff requires execution of code that has nothing to do with your task at hand.

So handlers are more efficient. Handlers also can do things that aren't possible with WebForms such as the ability to process requests without the need to have a physical file on disk, which is known as a virtual Url. To do this make sure you turn off 'Check that file exists' checkbox in the Application Extension dialog shown in Figure 1.

This is common for content providers, such as dynamic image processing, XML servers, URL Redirectors providing vanity Urls, download managers and the like, none of which would benefit from the WebForm engine.

Have I stooped low enough for you?

Phew – we've come full circle here for the processing cycle of requests. That's a lot of low level information and I haven't even gone into great detail about how HTTP Modules and HTTP Handlers work. It took some time to dig up this information and I hope this gives you some of the same satisfaction it gave me in understanding how ASP.NET works under the covers.

Before I'm done let's do the quick review of the event sequences I've discussed in this article from IIS to handler:

- IIS gets the request
- Looks up a script map extension and maps to aspnet_isapi.dll
- Code hits the worker process (aspnet_wp.exe in IIS5 or w3wp.exe in IIS6)
- .NET runtime is loaded
- IsapiRuntime.ProcessRequest() called by non-managed code
- IsapiWorkerRequest created once per request
- HttpRuntime.ProcessRequest() called with Worker Request
- HttpContext Object created by passing Worker Request as input
- HttpApplication.GetApplicationInstance() called with Context to retrieve instance from pool
- HttpApplication.Init() called to start pipeline event sequence and hook up modules and handlers
- HttpApplicaton.ProcessRequest called to start processing
- Pipeline events fire
- Handlers are called and ProcessRequest method are fired
- Control returns to pipeline and post request events fire

It's a lot easier to remember how all of the pieces fit together with this simple list handy. I look at it from time to time to remember. So now, get back to work and do something non-abstract...

Although what I discuss here is based on ASP.NET 1.1, it looks that the underlying processes described here haven't changed in ASP.NET 2.0.

Many thanks to Mike Volodarsky from Microsoft for reviewing this article and providing a few additional hints and Michele Leroux Bustamante for providing the basis for the ASP.NET Pipeline Request Flow slide.

If you have any comments or questions feel free to post them on the Comment link below.

Comments or Questions