Understanding ASP.NET

In this chapter, we look under the covers of ASP.NET. Many fundamentals of ASP.NET just scratch the surface but to get the most out of the framework it is a good idea to look much deeper.

Objectives in this chapter:

* Explain how ASP.NET works internally and highlight the relevant features needed for your everyday tasks.
* Outline the behind-the-scenes concepts: the application life cycle, the page life cycle and the control creation process.
* Cover the steps taken to translate your \*.aspx and \*.cs files into compiled code.
* Delve beyond the common ASP.NET features such as form stickiness and view state.

# A Promise in Advance

ASP.NET is a powerful and flexible architecture for building web applications. The high level parts – WebForms and WebServices – are well-known. However, as your applications scale up in real life, sooner or later you encounter performance issues, or find that things which seemed easy become challenging. That is what this book provides – a look beyond the big picture. The little details are harder to understand and sometimes more abstract; less fun for more work. But let me begin with a promise: it’s worth the effort.

# The Low Level Architecture of Request Handling

Understanding the innermost parts of a platform is highly satisfying. You will feel confident knowing that you can write better applications and recognizing why they work. In this section we will examine the system level of ASP.NET in order to understand how requests flow through the processing pipeline. This is not aimed at those learning the basics or creating a simple interactive page. This is what you need to know when you write or build large sites which involve hundreds of pages handling requests from thousands of users.

WebForms and WebServices are both sophisticated, high level implementations of HTTP handlers. They are built on top of the ASP.NET framework and exposed as default project templates. Most developers are happy with these project types, and are ignorant of, or misunderstand, the additional potential that exists. The basic HTTP handlers are built with managed code. This means that you can get highly customized behavior processing requests through the pipeline by using your very own code.

### What is ASP.NET

In general terms, 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 completely separate from HTTP runtime or the Web Server. In fact, the HTTP runtime is a component that you can host in your own applications outside of Internet Information Services (IIS) or any server side application altogether. Visual Studio and the integrated development server is a good example of an implementation that bypasses IIS.

The HTTP runtime is responsible for routing requests through this pipeline, a complex yet very elegant mechanism. Several interrelated objects, extensible via subclassing or through interfaces, are available for customization work. This makes the framework highly adaptable. In this book I’ll cover most of these extensibility points and show that there are virtually no limits when using ASP.NET.

Through this mechanism it’s possible to hook into low-level interfaces such as authentication, authorization, and caching. You can even filter content by routing incoming requests that match a specific signature directly to your code. Of course, there are a lot of different ways to accomplish the same thing – but all of the approaches are straightforward to implement.

The ASP.NET engine was written entirely in managed code and all of the extensibility functionality is provided via “managed code extensions”. The impressive part of ASP.NET is that it is very powerful but simple to work with. Despite its breadth and complexity, accomplishing your desired outcomes is easy. ASP.NET enables you to perform tasks that were previously the domain of ISAPI extensions and filters on IIS. ISAPI is a low-level Win32 API that has a very spartan interface. It was very difficult to develop anything on top of this interface. Since ISAPI is low-level, it is very fast. But writing ISAPI filters in C++ is not included in most current application level development. Thus, for some time ISAPI development has been largely relegated to providing bridge interfaces to other application or platforms, such as PHP. But ISAPI did not become obsolete with the appearance of ASP.NET. The ASP.NET engine interfaces with IIS through an ISAPI extension. This extension hosts .NET through the ASP.NET runtime.

ISAPI provides the core interface from the Web Server and ASP.NET uses the unmanaged ISAPI code portion to retrieve input and send output back to the client. The content that ISAPI provides is passed using common objects like HttpRequest and HttpResponse that expose the unmanaged data as managed objects. Back in the .NET world it becomes very easy to use these objects in your own code. I’ll later explain how the umanaged and managed world interoperate.

### The Lifetime of a Request

The lifetime starts with an ASP.NET request. Whenever the user types in a URL, clicks on a hyperlink, or submits an HTML form on the browser, a request is sent to the server.

Note: It’s essentially the same for web services. A client application calls an ASP.NET based Web Service by sending a request. Therefore, I refer to the term Request and don’t differentiate between Web Service and browser requests. You can assume that the examples given are intended to run in a browser environment.

For the sake of clarity, we left out the steps made behind the scenes in the browser and through the protocol stacks. This includes port assignment within the TCP/IP stack and name resolution using the DNS protocol. As long as we’re talking about ASP.NET we have no direct influence on DNS, so let’s keep this in mind but not complicate the description with side effects.

Domain NAME SYSTEM

The Domain Name System (DNS) is hierarchical naming system for internet resources. Mainly these are computers that get human readable names—the hostnames—the DNS translates into IP addresses. Beginning with some root servers the DNS is a hierarchy of name resolution servers for particular levels of the hostname. The protocol running between the servers and clients is called DNS protocol.

.NET Framework supports DNS with several classes. However, for ASP.NET applications usually there is no need to program directly against this low level protocol. The request we focus on begins when the name resolution process is done.

On the server side the web server picks up the request. In this description we focus mainly on ISS7 with some words for IIS6, which is still widely used. However, most parts will not refer to a specific version, so the term IIS covers all from IIS5 to IIS7 (and beyond, probably). Within IIS the request is usually routed to the aspx page. How this process works internally depends entirely on the HTTP handler that handles the request. The mapping between the .aspx extension and the ISAPI DLL, aspnet\_isapi.dll, is responsible for this. Every request that handles applications extensions which we want to be served by ASP.NET must be routed that way. This means that the extension is a pre-defined, yet voluntary, definition. Imagine if you mapped .html pages to be processed by ASP.NET as well — it wouldn’t be obvious that ASP.NET was doing the trick.

This means, too, that different extensions might route to different handlers. For instance, the .asmx extension is routed to the web service handler. Instead of opening a page and starting a page parser, this request does not open a file but a specially attributed class that identifies the implementation. Many other handlers are installed with ASP.NET and you are also able to define your own. All these handlers are mapped to the ASP.NET ISAPI extension and configured in web.config to get routed to a specific implementation.

In coding terms the handler is a type, implemented by a .NET class that handles a specific extension. You can also attach your own handlers to existing extensions and route the request through both your own and the default implementation. The extension is the basic mapping where the processing starts. The basic mappings already available are shown in Table 1.

Table 1-1. Application mappings assigned to aspnet\_isapi.dll

|  |  |  |
| --- | --- | --- |
| Extension | Resource Type | Comments |
| .asax | ASP.NET application files. | Usually the global.asax file only. |
| .ascx | ASP.NET user control files. | Usually these files are not called directly. |
| .ashx | HTTP handlers. | The managed counterpart of ISAPI extensions; see chapter 3 for details. |
| .asmx | ASP.NET web services. | Obsolete since the appearance of Windows Communication Foundation (WCF). |
| .aspx | ASP.NET web pages. | The regular page handler. |
| .axd | ASP.NET internal HTTP handler. | Used for embedded resources like JavaScript or images pulled from compiled code. |
| .svc | Web service handler. | WCF based services now have its own extension. |

Let’s talk about the asmx extension shown in the table. With the introduction of .NET 3.0, Microsoft moved all the communication stuff like .NET remoting and web services to Windows Communication Foundation (WCF) base library[[1]](#footnote-1). This means that the ASP.NET based webservices are superseded. Even though they are still fully supported it makes sense to consider moving web service projects from ASP.NET platform to WCF. It’s not a significant change, as most classes are similar and WCF has some more powerful approaches. However, I don’t cover WCF in this book and therefore the following description is limited to the ASP.NET portion.

#### From ISAPI to ASP.NET

The Internet Service API[[2]](#footnote-2) (ISAPI) is a common Win32 API to access IIS on a very low level. This means that it’s both fast and “unfriendly”. The interface is optimized for performance but it’s also very simple and straightforward. For developers with .NET experience it’s pretty hard to use, because the coding style you use in .NET to, say, create sophisticated infrastructure solutions, is the opposite of the style you see when coding ISAPI extensions in C++. Many high level web development languages (such as PHP, Perl, and even ASP.NET) are built on top of ISAPI.

ISAPI is good for writing such environments. For application developers it’s not the best way to write our sites on time and within budget. However, ISAPI is the layer our ASP.NET engine is built on and a good understanding is helpful for getting the most out of ASP.NET. For ASP.NET the ISAPI level is just acting as a routing layer. The heavy stuff, such as processing and request management, occurs inside the ASP.NET engine and is mostly done in managed code.

You can think of ISAPI as a sort of protocol. This protocol supports two flavors, ISAPI extensions and ISAPI filters. Extensions act as a transaction interface; they handle the flow of data in to and out of the web server. Each request coming down the pipeline is going through the extensions and the code decides how they are treated. As you might imagine, ASP.NET is such an extension. ASP.NET has several ways to give you as much control as possible to hook into this extension and modify the default behavior. The low level ISAPI interfaces are now available as high level .NET interfaces, named IHttpHandler and IHttpModule. This is very powerful and still provides good performance, because it’s a well written balance between lean access to the lower level and an easy-to-use high level API.

Like any other ISAPI extension the code is provided as a DLL and is hooked into the IIS management. You can find this DLL here:

<.NET FrameworkDir>\aspnet\_isapi.dll

If you have several versions of .NET framework installed you may wonder why there is just one such DLL in the tree. I predict that you will find this in the v2.0.50727 folder or at least in one with v2.0 at the beginning. If you have .NET 1 installed, another version might reside in a folder named v1.1.x. The reason is that Microsoft has added a lot of features in .NET 3.0 and .NET 3.5 regarding ASP.NET, but the low level interfaces are still the same. All new functionality is completely written in managed code. This is indeed a transition as it shows that it’s possible to write infrastructure components in managed code. With powerful hardware the performance loss is not that critical whereas the benefits regarding security, reliability, and shorter development cycles matter.

Note: With the upcoming .NET 4.0 a new engine appears on the horizon that is closer to IIS7 and has several internal improvements. For now, it seems as though most of the techniques and internal behaviors described here have barely changed.

#### The extension mapping

As you have already seen, the web server recognizes resource requests by analyzing the file extension. The first step to get the ASP.NET ISAPI extension running is a mapping between the several file types and the DLL I mentioned earlier. In IIS7 you can see this by following these steps:

\* Open Internet Information Services (IIS) Manager

* \* Choose the server node
* \* In the right pane scroll to the IIS table

\* Double click Handler Mappings

In the table you will see a column Path with the extensions already assigned. In the column Handler you will see the assigned module. For the \*.aspx extension there are these mappings:

\* PageHandlerFactory-Integrated

\* PageHandlerFactory-ISAPI-2.0

Additionally, with .NET 4.0, two additional mappings are installed (on a 32-bit machine):

\* PageHandlerFactory-Integrated-4.0

\* PageHandlerFactory-ISAPI-4.0\_32bit

For the unmanaged side you will see the IsapiModule handler and for the managed one .NET types used to handle the pages.

Insert ASPEXTf0101.tif

Figure 1-1. IIS7 maps extensions like \*.aspx to ISAPI extensions

If the mapping isn’t there, ASP.NET is probably not correctly installed. Don’t try to map this by hand. There is a lot of configuration that happens behind the scenes. To register or re-register the mappings just invoke the following command:

cd <.NetFrameworkDirectory>

aspnet\_regiis - i

Again, this command is not available for all .NET framework versions. Version 2.0 will do the trick, even if you run 3.0 or 3.5. With ASP.NET 4.0, a new ISAPI DLL will be introduced — but backward compatibility is still guaranteed. If you still have version 1.x sites running, there is another version available. It’s a bit tricky, but in order to run two different ASP.NET versions, just register with the highest available. In IIS6 the properties of a specific site of a web server let you choose the right framework. In IIS7 it has been moved to the application pool settings dialog. Each application pool can run only one version of the runtime. To set another framework for a specific site, you have to create another application pool, set the appropriate framework version, and assign the application pool to the site.

ASPEXTf0102.tif

Insert ASPEXTf0102.tif

Figure 1-2. Adding a new application pool and get the right framework there

### The request comes in

When a request comes in, IIS checks for the script map and routes the request to the associated extension. In case of ASP.NET we assume that the request is something like Default.aspx, so it’s being routed to aspnet\_isapi.dll.

Insert ASPEXTf0103.tif

Figure 1-3. Request flow through IIS and down to ASP.NET runtime

I assume that you have already worked with and probably configured the application pool. The application pool was introduced with IIS6 and allows the complete isolation of applications from each other. This means that IIS is able to completely separate things happening in one application from those in another. Putting applications together in one pool could still make sense, because another pool would create its own worker process and, as shown in Figure 1-3, could use up many resources. Separate applications make the web server more reliable. If an application hangs, consumes too much CPU time, or behaves unpredictably, it influences its entire pool. Other application pools (and the applications within them) will continue to run. Additionally, the application pools are highly configurable. You’ve already learned that the framework version can be different for each pool, which is very useful for migration scenarios. You can configure the security environment by choosing the impersonation level and customizing the rights given to a web application. Application pools are executables that run as any other program. This makes them easy to monitor and configure. Although this does not sound very “low level”, application pools are highly optimized to talk directly to the kernel mode driver http.sys. Incoming requests are directly routed to the pool attached to the application. At this point you might be wondering where InetInfo is gone. It is still there, but it is basically just an administration and configuration service. The flow of data through the IIS system goes as directly and quickly as possible, straight down from http.sys to the application pools. This is one reason why IIS7 is much faster and more reliable than any other IIS before it.

An IIS7 application pool also has intrinsic knowledge of ASP.NET and in turn ASP.NET can communicate with the new low level APIs that allow direct access to the HTTP Cache APIs. This can offload caching from the ASP.NET level directly into the Web Server’s cache, which again will improve performance drastically.

In IIS7, ISAPI extensions run in the Application Pool’s worker process. The .NET Runtime also runs in this same process, and consequently communication between the ISAPI extension and the .NET runtime runs “in-process”, which is inherently more efficient.

### Getting into the .NET runtime

Now let’s look at what happens to the request when accessing the managed level. The worker process w3wp.exe hosts the .NET runtime, and the ISAPI DLL calls into a small set of unmanaged interfaces via low level COM[[3]](#footnote-3). Unfortunately, there is not much information available from Microsoft. Apart from a few blogs that we can’t always trust, it’s hard to say how the interaction between ASP.NET and ISAPI is made. Reading the manual doesn’t help either. Microsoft states that the API “supports the .NET Framework infrastructure and is not intended to be used directly from your code”, which confirms the existence of the interface but nothing more.

Using a disassembly tool like .NET Reflector (from red-gate.com) is one way to look into the details. Let’s examine the System.Web DLL, which contains everything we’d like to know. First, the entry point is the namespace System.Web.Hosting, where you’ll see how the runtime interacts with the ISAPI part.

Insert ASPEXTf0104.tif

Figure 1-4. If there is no information available we can use Reflector to reveal the internal code

If the runtime handles a request, it calls the ProcessRequest method of the IISAPIRuntime interface. The interface is part of ISAPI and exposed as COM. The first parameter will return a pointer which gives access to the ISAPI module.

Listing 1-1. Definition of the ProcessRequest method

[return: MarshalAs(UnmanagedType.I4)]

int ProcessRequest([In] IntPtr ecb, ⮰

[In, MarshalAs(UnmanagedType.I4)] int useProcessModel);

The parameter called “ecb” returns the ISAPI Extension Control Block (ECB, (Microsoft)) that is passed as an unmanaged resource. The method 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. This includes 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 output is later accessible via the Response objects, but you can see the tight relation between incoming request and outgoing response. The ecb pointer basically provides access to the functionality of an ISAPI request. The ProcessRequest method is the entry and exit point where this resource initially enters the street up to our managed code world and where the managed processing ends.

#### Threads and processes

So far things are not quite easy but the flow of the request through the code is straight. However, in real life things tend to become more complex. The ISAPI extension runs requests asynchronously. This means that 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. This asynchronous processing releases the ISAPI worker thread immediately, and forwards processing to a separate thread that is managed by ASP.NET. We’ll look into threading later to understand what ASP.NET is doing. For now, ASP.NET receives this ECB reference and uses it internally to retrieve information about the current request.

This includes such information as server variables, POST data, and output returning to the server. The ECB data block stays alive until the request finishes or times out in IIS. The ASP.NET engine continues to communicate with it until the request is done. Any output is written into the ISAPI output stream using the appropriate method call and when the request is done, the ISAPI extension is notified of request completion. The extension will then free the ECB from memory. Just remember that in this cruel unmanaged world the code is responsible for handling all in-memory actions, and not releasing memory leads to memory leaks. However, the implementation is very efficient as the .NET classes essentially act as a thin wrapper around ISAPI.

#### Loading the .NET runtime

Again, there is not much information available about the loading procedure. I assume that the runtime is loaded, if not yet present, when the first request to ASP.NET is made by a mapped extension. The managed ISAPIRuntime is then instantiated and starts talking to the unmanaged world. For isolation purposes, each virtual directory creates a new application domain (AppDomain). Within this AppDomain the ISAPIRuntime resides. Starting the application is also the beginning of the application’s life cycle, which we’ll look into later in this chapter. It’s also likely that the instantiation is made by the unmanaged part, because the wrapper interface is exposing the ComVisible attribute. This makes it available from the other side.

To create the ISAPIRuntime instance the AppDomainFactory.Create method is called. The following code snippet, created with Reflector, shows how it’s done internally.

Listing 1-2. The code of the AppDomainFactory.Create methode

[return: MarshalAs(UnmanagedType.Interface)]

public object Create(string appId, string appPath)

{

object obj2;

try

{

if (appPath[0] == '.')

{

FileInfo info = new FileInfo(appPath);

appPath = info.FullName;

}

if (!StringUtil.StringEndsWith(appPath, '\\'))

{

appPath = appPath + @"\";

}

ISAPIApplicationHost appHost = new ISAPIApplicationHost(appId, appPath, false);

ISAPIRuntime o = (ISAPIRuntime) this.\_appManager.CreateObjectInternal(⮰

appId, typeof(ISAPIRuntime), ⮰

appHost, false, null);

o.StartProcessing();

obj2 = new ObjectHandle(o);

}

catch (Exception)

{

throw;

}

return obj2;

}

Even if it is not visible in this code snippet, internally the appId is used to cache the relation with the requested virtual directory. Therefore subsequent calls do not start the application again but process the existing AppDomain. Finally, in this code StartProcessing forces ISAPI to process the request, and, because it is asynchronous, the call returns immediately and the method returns the ECB as type object.

Insert ASPEXTf0105.tif

Figure 1-5. Behind the scenes the first request is responsible for the creation of a new application

#### Handle the Request within the Runtime

The managed part is now alive and ISAPI is able to call it whenever there is something available for processing. In basic ASP.NET terms everything begins here. The AppDomain, the application life cycle and the first page cycle are born here and live for varying lengths of time. From now on, we’re back in the managed world. But there is a bit more you should know about the whole procedure. IIS is a multithreaded host and so is ISAPI. Each request is processed asynchronously, running in its own thread. Again, a look into the code using Reflector reveals what ASP.NET is doing here. The ProcessRequest method 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.

public int ProcessRequest(IntPtr ecb, int iWRType)

{

IntPtr zero = IntPtr.Zero;

if (iWRType == 2)

{

zero = ecb;

ecb = UnsafeNativeMethods.GetEcb(zero);

}

ISAPIWorkerRequest wr = null;

try

{

bool useOOP = iWRType == 1;

wr = ISAPIWorkerRequest.CreateWorkerRequest(ecb, useOOP);

wr.Initialize();

string appPathTranslated = wr.GetAppPathTranslated();

string appDomainAppPathInternal = HttpRuntime.AppDomainAppPathInternal;

if ((appDomainAppPathInternal == null)

|| StringUtil.EqualsIgnoreCase(appPathTranslated, appDomainAppPathInternal))

{

HttpRuntime.ProcessRequestNoDemand(wr);

return 0;

}

HttpRuntime.ShutdownAppDomain( ApplicationShutdownReason.PhysicalApplicationPathChanged,

SR.GetString("Hosting\_Phys\_Path\_Changed", new object[] { appDomainAppPathInternal, appPathTranslated }));

return 1;

}

catch (Exception exception)

{

// removed for sake of clarity

}

}

The code is just a part of the complete method. I have removed the catch block to focus on the working code. The method receives the pointer to an ECB data block and passes it to the CreateWorkerRequest method. This method is responsible for creating the ISAPIWorkerRequest object. From now on we have a Request object available that is able to talk directly to the ISAPI layer.

The ISAPIWorkerRequest (defined in the System.Web.Hosting namespace) class is an abstract subclass of HttpWorkerRequest. This class holds all the knowledge to talk HTTP. It’s a piece of cake, and although it would take too much time to go into here, it’s worth a closer look if you like. I’d encourage you to go on and find out how things work under the hood. Just open the method ProcessRequest in Reflector, click on ISAPIWorkerRequest and then on the HttpWorkerRequest subclass. At the end of the listing of signatures of either class click on the link Expand Methods and there you have it. Because we have an abstract class, eventually there must be some implementation elsewhere. The factory method that creates the type using CreateWorkerRequest method, is able to return one of the following four types available:

\* ISAPIWorkerRequestInProcForIIS7

* \* ISAPIWorkerRequestInProcForIIS6
* \* ISAPIWorkerRequestInProc

\* ISAPIWorkerRequestOutOfProc

It’s a Bit in the ECB that drives the decision. This means that finally the ISAPI module controls the way ASP.NET handles requests. Because ASP.NET is not limited to IIS, it’s simply the version of IIS that is responsible. ISAPIWorkerRequestInProc is for all IIS versions up to and including version 6 and IIS7 serves all versions from 7 and above. This sounds silly—there is no IIS8 in existence—but the ASP.NET engine would be robust against new versions as long as the ISAPI module is able to emulate IIS7 behavior.

Whenever you have trouble understanding what is going on in your application, whether it’s with creating and sending headers, receiving weird stuff, or getting wrong data down the wire, it’s worth taking a closer look here. HttpWorkerRequest is meant to provide a high level abstraction around the low level interfaces, regardless of the source of the data. However, reading and understanding are two different things. Let’s take an example of how to deal with all the internals. The next listing shows you how the well known QueryString is retrieved from the ECB. Moving down the class hierarchy, we find that it’s partly implemented in the base class and partly in the ISAPIWorkerRequestInProc class. For IIS6 and IIS7 there is no distinct change here, so we can skip these classes for now.

internal override int GetQueryStringRawBytesCore(byte[] buffer, int size)

{

if (base.\_ecb == IntPtr.Zero)

{

return 0;

}

return UnsafeNativeMethods.EcbGetQueryStringRawBytes(base.\_ecb, buffer, size);

}

[DllImport("webengine.dll")]

internal static extern int EcbGetQueryStringRawBytes(IntPtr pECB, byte[] buffer,   
 int size);

Again, there is nothing thrilling here. These classes are all very thin wrappers around native method calls. In fact, ASP.NET does not provide any significant overhead to the request procedure. That means ASP.NET is extremely fast — as fast as IIS with native code. If you feel (or know) that your application is slow, then you know that in most cases it will be a problem in your code. Let’s focus on the managed side of the world and learn more about what’s going on here.

### HttpContext and HttpApplication

You’re probably already familiar with the HttpContext and HttpApplication classes. Instances of these classes are created whenever a request hits the engine. At a glance, the steps are:

\* Create a new HttpContext instance for the request

* \* Retrieve an HttpApplication instance (or create one, if it’s the first request)
* \* Call HttpApplication.Init to set up pipeline events

\* Call HttpApplication.ResumeProcessing to start the ASP.NET pipeline processing

#### Wrapping the Request: HttpContext

The context of an HTTP request is available throughout the lifetime of the request. For ease of use it’s always accessible through the static HttpContext.Current property. Because every request usually starts a new thread, your code runs in that thread. In chapter 2 I’ll explain in more detail the thread and processing behavior. For now, this isn’t needed to understand the request processing on the context level. The current context has a one-to-one relation to the request for that thread. This context object is also the place where all other objects required to process the request are stored: Request, Response, Application, Server, and Cache. At any time during request processing HttpContext.Current gives you access to all these. There are several shortcuts, so you may use the Page’s property Context instead, but it’s exactly the same object.

You may already use several collections available in the Application, Session, and Cache objects. Even the HttpContext class contains a useful collection that provides a store for request specific data. The big difference to the other data collections is the life time. The HttpContext represents exactly the current request. If the processing is finished and the data sent to the server, then the collections are also disposed of. However, the Context exists before the page’s life span begins and ends after it finishes. This data storage is therefore a bit more powerful than simple page members.

The following sample shows how to use Context.Items to get information about the request processing elapsed time. At the BeginRequest event a timestamp is stored and retrieved later at EndRequest.

Listing 1-5. Get request information using events in global.ascx

Application\_BeginRequest(object sender, EventArgs e)

{

if (Settings.Default.Logging)

{

Context.Items.Add("LogTime", DateTime.Now);

}

}

protected void Application\_EndRequest(object sender, EventArgs e)

{

if (Settings.Default.Logging)

{

DateTime end = DateTime.Now;

TimeSpan span = end.Subtract((DateTime) Context.Items["LogTime"]);

System.Diagnostics.Debug.WriteLine(span.TotalMilliseconds, "RequestTime");

}

}

This sample uses two events defined in the global.ascx code behind file. You might be thinking of how to put the results into a database or log file, but for now the output to a Visual Studio console window will do.

Insert ASPEXTf0106.tif

Figure 1-6. Get time information about the request while debugging

If you feel that the values shown in the figure are too high, you need to be aware that I’m running the entire environment in a Virtual PC console, and that the timestamps shown are expressed in milliseconds.

#### The most basic type: HttpApplication

The HttpApplicationFactory is responsible for creating an appropriate number of HttpApplication objects. The load and the number of threads required affect how it handles incoming requests. The size of the pool is limited to the MaxWorkerThreads setting in the ProcessModel key in the machine.config, which by default is 20. Since the days of .NET 2.0 it’s still an auto-configured entry:

<processModel autoConfig="true" />

Change this to the following:

<processModel autoConfig="false" maxWorkerThreads="30" />

You can find more information about threading in chapter 2.

The pool starts out with a smaller number— usually one — and it then grows as multiple simultaneous requests are processed. The pool is monitored so that under load it grows to a maximum number of instances. Later the pool is scaled back to a smaller number as the load drops. Once we have the right number of instances of HttpApplication, we have the real entry point within the managed world for an incoming request. HttpApplication is like an outer container for the whole application. For easy access the events related to the application are mapped in the global.asax file, which is the declarative expression of this class. The definition of global.asax in global.asax.cs reveals:

public class Global : System.Web.HttpApplication

The pipeline the request is going through fires several events to let you control and intercept the various states of the application’s life cycle. The following list shows all of them, even if the majority are never used in an application. In the next few sections I’ll dig deeper into the world of application events.

Listing 1-2.Events available on application level (pulled from HttpApplication)

public event EventHandler AcquireRequestState;

public event EventHandler AuthenticateRequest;

public event EventHandler AuthorizeRequest;

public event EventHandler BeginRequest;

public event EventHandler Disposed;

public event EventHandler EndRequest;

public event EventHandler Error;

public event EventHandler LogRequest;

public event EventHandler MapRequestHandler;

public event EventHandler PostAcquireRequestState;

public event EventHandler PostAuthenticateRequest;

public event EventHandler PostAuthorizeRequest;

public event EventHandler PostLogRequest;

public event EventHandler PostMapRequestHandler;

public event EventHandler PostReleaseRequestState;

public event EventHandler PostRequestHandlerExecute;

public event EventHandler PostResolveRequestCache;

public event EventHandler PostUpdateRequestCache;

public event EventHandler PreRequestHandlerExecute;

public event EventHandler PreSendRequestContent;

public event EventHandler PreSendRequestHeaders;

public event EventHandler ReleaseRequestState;

public event EventHandler ResolveRequestCache;

public event EventHandler UpdateRequestCache;

By writing something like the following into the code behind file, you attach a handler:

protected void Application\_BeginRequest(object sender, EventArgs e)

Avoiding an explicit delegate definition is just for the sake of convenience.

The whole handling of AppDomain, HttpApplication, and threads might look confusing. Remember that the server has to handle multiple incoming requests simultaneously. Each ASP.NET application runs in its own AppDomain, where several instances of HttpApplication can run in parallel, supplied from the pool managed by the factory. To get a better understanding of this, let’s examine the information we can extract from related objects.

Listing 1-5. Retrieving information about the application

protected void Page\_Load(object sender, EventArgs e)

{

Guid appId = ((Global) Context.ApplicationInstance).AppId;

this.appId.Text = appId.ToString();

this.threadId.Text = Thread.CurrentThread.ManagedThreadId.ToString();

this.domainId.Text = AppDomain.CurrentDomain.FriendlyName;

this.threadInfo.Text = Thread.CurrentThread.IsThreadPoolThread ? ⮰

"Pool Thread" : "No Thread";

this.threadApart.Text = Thread.CurrentThread.GetApartmentState().ToString();

Thread.Sleep(4000);

}

The Sleep call allows you to open several browser instances and hit F5 to refresh the page within the four second period in order to see how it works. This simulates the behavior when several requests come in while previous requests are still running.

An application by definition has no internal id. To create one, add the following code to the global.asax code behind:

internal Guid AppId = Guid.NewGuid();

Figure 1-7 shows three screens made within the four second period using three different browser windows.

Insert ASPEXTf0107.tif

Figure 1-7. The application’s id changes, as the thread id does

You can see that the application id and the thread id change. If you remove the Sleep command and request the pages one by one, the application id is always the same. Usually the thread is recycled from the pool and the id is also unchanged.

The abbreviation ‘MTA’ for the thread’s apartment model stands for Multi Threaded Apartment. 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 with a safe environment in which to run. 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. As long as there you have multi threaded COM component in your application only, you can simply ignore this. For single threaded components some additional work is required. Usually a well developed COM component should support multiple threads.

It is a simple fact that all HttpApplication objects are in the same AppDomain. This has, however, some influence on their behavior. If you change something in the page’s code or web.config, the application restarts. To be more precise, it’s actually the AppDomain which is being shut down and restarted. This ensures that all currently existing HttpApplication instances are also shut down and recreated. To see this behavior in action, open the example shown in Listing 1-5 and launch it. Hit F5 several times and notice that the Domain ID value is still the same. Now change something in the web.config file. This forces a shutdown and therefore the creation of a new AppDomain. The Domain ID value is now different.

Currently running requests are being processed properly, even if the new AppDomain is already up and running. To avoid old threads blocking resources, the request has a specific time out value. Once timed out, the threads are shut down and the request’s “life” ends.

### Flowing through the ASP.NET Pipeline

You now have a good overview firstly of what happens when a request comes in, and secondly of how the objects start up as they wait to be used in your code. There are several steps following this startup sequence worth investigating further. In each step you’ll find events and callbacks to intercept the process and customize or optimize its behavior.

Figure 1-8 illustrates what happens during the various states. If you look into the code using Reflector, you can find the steps in subclass ApplicationStepManager and the additional steps for IIS7 in PipelineStepManager.

Insert ASPEXTf0108.tif

Figure 1-8. Events that build the pipeline

The most important information here is that modules precede handlers. They are loaded and executed at the particular state of the processing pipeline, if configured properly in web.config file. The figure shows only the minimum number required to execute a page. You’ll find a closer look at all these events in the next section. For now, it’s enough to know when and why the modules and handlers get executed. This is the major part of the pipeline business, and I’ll go into more detail in the next section.

### Modules and Handlers using HttpModule and HttpHandler

HttpApplication and HttpContext are merely containers for incoming messages. They build the pipeline by forming a chain of events that commence in a defined order, and they hold together data related to the request, in order to give other instances easy access to the information involved in the process. However, the real work is done in other processing units, particularly the modules and handlers. Modules are built on top of the HttpModule class. Handlers are built on top of the HttpHandlers class. Both are abstract classes and need concrete implementations. In addition, both are highly configurable and act as a chain of instances. That means that you can attach as many modules and handlers as you like and the request will flow through all these instances.

Several tasks could be handled in both levels — however, there is a different intention behind them. Modules tend to control tasks on a lower level; their nature is more basic than that of handlers. Think of modules as the right place to prepare data being processed by handlers or modify data subsequently when the regular treatment by a handler is completed. IIS programming aficionados can think of modules as ISAPI filters. The good news is that programming ASP.NET modules using .NET is much easier than developing filters for IIS. Moving on from IIS, let’s focus on the amazing things that become possible as we extend ASP.NET.

The natural order of modules and handlers is:

1. Use module to pre-process a request

2. Use handler to process the request

3. Use module again to post-process the request

ASP.NET comes with several default handlers, such as the very basic ones for \*.aspx pages or web services. Some default modules exist that are responsible for simple tasks like authentication and caching. In an ideal world, the modules are transparent to the handlers. This means that handlers don’t know about the modules that work before or after their own processing. ASP.NET allows several easy ways to create your own modules and handlers. This is usually the first approach of extending the standard behavior of ASP.NET. Let’s take a closer look at this technique.

Insert ASPEXTf0109.tif

Figure 1-9. Modules and Handlers fit closely in the ASP.NET model

In Chapter 3, we’ll take a closer look at the process of developing custom modules and custom handlers with some clever examples. By now it’s enough to know that several modules process the request one after another on both, the request and the reponse path. Whereas only one handler processes the request finally, either an internal handler or a custom one.

## The Life Cycles

The life cycle is a term used in almost every introduction to ASP.NET. This term is absolutely important for a complete understanding and the correct usage of ASP.NET.

To break things down we can divide the different life cycles into three parts:

The application’s life cycle

The page’s life cycle

The control’s life cycle

The following sections covers this in-depth. The behavior of the cycles is slightly different between IIS versions but the overall description is the same. The following explanations focus on IIS7 and Framework 3.5 behavior, which is valid for .NET 2.0 up to the upcoming 4.0.

IIS7 can run in two different modes. The IIS integrated mode is the new native one, whereas the classic mode mimics the behavior of IIS5 or IIS6. The integrated mode has several advantages and should be preferred if there is no explicit requirement to step down.

## The IIS7 integrated pipeline

The IIS7 integrated pipeline is a unified request processing pipeline. Each incoming request is handled by this pipeline and routed through the internal parts of IIS. The pipeline supports both managed and native code modules. You may already know about creating managed modules based on the IHttpModule interface. Once implemented and hooked into the pipeline, the module receives all events used to interact with the request passing through the pipe.

The term “unified request processing pipeline” needs some more investigation. IIS6 provided two different pipelines: one for native and one for managed code. This is obviously for historical reasons, because the managed world came after the IIS world. In IIS 7 both pipelines united to become the unified request processing pipeline. For ASP.NET developers this has several benefits:

\* The integrated pipeline raises all exposed events, which enables existing ASP.NET modules to work in the integrated mode.

* \* Both native-code and managed-code modules can be configured at the Web server, Web site, or Web application level.
* \* Managed-code modules can be invoked at any stage in the pipeline.

\* Modules are registered and enabled or disabled through an application’s Web.config file.

The configuration of modules includes the built-in ASP.NET managed-code modules for session state, forms authentication, profiles, and role management. Furthermore, managed-code modules can be enabled or disabled for all requests, regardless of whether the request is for an ASP.NET resource like an .aspx file.

Invoking modules at any stage means that this may happen before any server processing occurs for the request, after all server processing has occurred, or anywhere in between.

## The Application’s Pipeline

In more generic terms the pipeline describes the flow through several instances within the IIS. Each step processes the request in a distinct way and fires appropriate events. This forces interactions with the modules and applications attached by configuration. The variety of actions you can perform ranges from none to total control. However, doing nothing lets the built-in modules accomplish their job and handle the basic page processing. From the perspective of the extensibility of ASP.NET, “total control” is what we’re looking for.

D:\UserData\jkrause\Documents\Meine Daten\Bücher\Fachbuchprojekte\ASP.NET Extensibility\Images\ASPEXTf0110.tif

Figure 1-10. Application’s life cycle events

Two states are new in IIS7: Map Handler and Log Request. This gives developers much more control over the request process. Some modules are managed whereas others are written in native code. You can add your own modules in either way. However, as a .NET developer, you’d probably prefer writing in managed code and indeed I would strongly recommend this. (Native code is an option if you need to create a high performance module which performs simple functions.)

### The Request Arrives

The application life cycle begins when a request is made for an application resource sent by a client application, such as a browser, to the Web server. In integrated mode, the unified pipeline handles all requests. When the pipeline receives a request, it’s being passed through stages that are common to all requests. These stages are represented internally by the RequestNotification enumeration. All request stages can be configured to allow developers to take advantage of ASP.NET functionality. That functionality is encapsulated in managed-code modules that have access to the request pipeline. For instance, even though the .htm filename extension is not mapped by default to ASP.NET, a request for an HTML page still invokes ASP.NET modules. This enables you to take advantage of ASP.NET authentication and authorization for all resources.

#### The unified pipeline receives the first request

When the unified pipeline receives the first request for any resource in an application, an instance of the ApplicationManager class is created. This builds the application domain (AppDomain) in which the request is processed. Application domains provide isolation between applications for global variables and enable each application to be unloaded separately. In the application domain, an instance of the HostingEnvironment class is created, which provides access to information about the application, such as the name of the folder where the application is stored.

Tip: When an application operates in the context of Visual Studio while you run a debug session, the application starts with the launch of the integrated web server. When the debug session ends and you launch the application again using the F5 key, the integrated web server will run the same application domain. This means that between the debug sessions no new application life cycle is being started. To force restarting the application you can right click the notification icon in the system tray and stop the web server.

The first request has several additional tasks. You will notice this because the elapsed time for the page to be delivered is longer than for all subsequent requests. During the first request, top-level items in the application are compiled, if required. This includes application code in the project. By the way, since the days of Visual Studio 2008, there is no longer a special code folder named App\_Code. Even if you name a folder this, it will have no special function. Code that resides in any folder at any level is compiled in one assembly per project.

#### Response objects are created for each request

After the application domain has been created and the HostingEnvironment object has been instantiated, application objects such as HttpContext, HttpRequest, and HttpResponse are created and initialized. These objects exist throughout the lifetime of the request and give developers full access to all related data.

The HttpContext class contains objects that are specific to the current application request, such as the HttpRequest and HttpResponse objects. The HttpRequest object contains information about the current request, which includes cookies and browser information. The HttpResponse object contains the response that is about to be send to the client, which includes all rendered output and header data, such as cookies. HttpContext.Current gives you permanent access to the current context through a static method. For easy access the Page class has properties like Request and Response that return instances of the very same related objects.

#### Differences between IIS6 and IIS7

We’re currently in the transition period from IIS6 to IIS7. While IIS7 is the best choice for a web server on Microsoft platforms, there are still many servers with IIS6 in production. Because you may already have a deep understanding of processing ASP.NET based on IIS6, the following explanation focuses on the key differences between IIS6 and IIS7, running in both Integrated mode and with the .NET Framework 3.5 or later. The following properties are specific to IIS7 with Integrated mode. You must run IIS7 to take advantage of these.

The HttpResponse object has a new property SubStatusCode. This is useful for setting codes for tracking failed requests. The Headers property of the HttpResponse object provides access to response headers for the response. Two properties of the HttpContext object, IsPostNotification and CurrentNotification, are used when one event handler handles several HttpApplication events. Both the Headers and ServerVariables property of the HttpRequest object are write-enabled.

#### An HttpApplication object is assigned to the request

After all application objects have been initialized, the application is started by creating an instance of the HttpApplication class. If the application has a Global.asax file, ASP.NET instead creates an instance of the Global.asax class that is derived from the HttpApplication class. It then uses the derived class to represent the application. This way you get access to the application events simply by overwriting the handlers within the global.asax code portion.

Note: The first time that an ASP.NET page or process is requested in an application, a new instance of the HttpApplication class is created. However, to maximize performance, HttpApplication instances might be reused for multiple requests.

#### The request is processed by the pipeline

The HttpApplication class performs several tasks while the request is being processed. Each step fires a specific request to allow you to run code when the event is raised. Without any further action, all incoming requests will fire all events and the attached modules will be invoked in sequence. To take advantage of extensibility, some interfaces come into the scope. IHttpModule is the basic interface for custom modules. In IIS7’s integrated mode you can use the module’s Init method to attach the required events.

One step you’re supposed to perform is the request validation. Imagine hackers trying to send malicious markup or other intrusions. The following list of events shows all steps you can run private code in. Validating the request should take place as early as possible, according to the following list in BeginRequest.

Raise the BeginRequest event.

Raise the AuthenticateRequest event.

Raise the PostAuthenticateRequest event.

Raise the AuthorizeRequest event.

Raise the PostAuthorizeRequest event.

Raise the ResolveRequestCache event.

Raise the PostResolveRequestCache event.

Raise the MapRequestHandler event. An appropriate handler is selected based on the filename extension of the requested resource. The handler can be a native code module such as the IIS7 StaticFileModule or a managed module such as the PageHandlerFactory class.

Raise the PostMapRequestHandler event.

Raise the AcquireRequestState event.

Raise the PostAcquireRequestState event.

Raise the PreRequestHandlerExecute event. Call either the ProcessRequest method for synchronous calls or the asynchronous version IHttpAsyncHandler.BeginProcessRequest of the appropriate IHttpHandler class.

Raise the PostRequestHandlerExecute event.

Raise the ReleaseRequestState event.

Raise the PostReleaseRequestState event.

Perform response filtering if the Filter property is defined.

Raise the UpdateRequestCache event.

Raise the PostUpdateRequestCache event.

Raise the LogRequest event.

Raise the PostLogRequest event.

Raise the EndRequest event.

Raise the PreSendRequestHeaders event.

Raise the PreSendRequestContent event.

## The Page’s Life Cycle

After understanding the application’s life cycle, you’re now ready to move further along the path taken by the request. The request is now being processed until the resources are prepared and ready to be served. For static resources it’s easy; the data is all ready to send to the client. (Examples of static resources include images, JavaScript files, and embedded objects.) The more exciting information is how dynamic resources are processed.

The basic steps in the page’s life cycle include initialization, instantiating controls, restoring and maintaining state, running event handler code, and rendering. As you saw in the application’s life cycle, you will be able to interact with the processing and change the handling for your intended effects. Some steps during the page’s life cycle are more complex and require a closer look in order to take full advantage of their customization potential. One major portion is the view state handling. I’ll dedicate a whole section to explaining view state later in this chapter.

It is also important for you to understand the page life cycle. As a developer creating custom controls, for instance, you must be familiar with the life cycle in order to correctly initialize controls, populate control properties with view state data, and run any control behavior code. You may have heard of a control life cycle, too. You may already know that a page in ASP.NET is a sort of specialized control and therefore a control’s life cycle is very similar. However, pages have more events available.

#### What about Master Page Life Cycle?

You might miss information regarding master pages here. The master page itself does not have any stages, because it’s not an object created and run standalone. The master page hierarchy is used to resolve the final construct of the page requested and then the page’s life cycle begins. Therefore, the master page is treated like a control, with some events related to the page’s life cycle events, but again, not with its own life cycle.

### Page Request

The page request occurs before the page life cycle begins. When the page is requested by a client, ASP.NET determines whether the page needs to be parsed and compiled, or whether a cached version of the page can be sent in response without executing the page. In each case the “life” of the page begins.

#### Start

In this step, page properties such as Request and Response are set. These properties give you access to the current HttpRequest and HttpResponse instances created through the application’s life cycle.

You can access all information related to the request using this.Request. (In later stages you can modify the response created and stored in this.Response.) At this stage, the page also determines whether the request is a postback or a new request and sets the IsPostBack property. This involves two checks: whether the request is done as a POST, and whether it comes from the same ASP.NET page. Additionally, during the Start step, the page’s UICulture property is set. At this point you can perform any action based on the automatically recognized page culture. Alternatively, you can force the culture property, so that the correct resources are obtained. (Forcing the culture doesn’t work in other steps).

#### Page initialization

During page initialization step, controls on the page become available and each control’s UniqueID property is set. The UniqueID is used later on postbacks to assign control state in order to make forms sticky, for instance. Any themes are also applied to the page. If the current request is a postback, the postback data has not yet been loaded and control property values have not been restored to the values from view state.

#### Load

During load, if the current request is a postback, control properties are loaded with information recovered from view state and control state. For a regular request via GET, view state and control state are set to their respective default states.

#### Validation

During this step the Validate method of all validator controls is called. The IsValid property of all validator controls is set and as a summary the page’s IsValid property is set, too.

#### Postback Event Handling

If the request is a postback, any event handlers are called, such as button click events.

#### Rendering

Before rendering, view state is saved for the page and all controls. Rendering is the process that creates the HTML that makes the page visible in a browser. Each control is responsible for rendering itself. This feature and the extensible design time support is a major opportunity for third party control vendors. Their controls can encapsulate all the clever design time experience and run time functionality within each control. That means during the rendering phase, the page calls the Render method for each control, providing a TextWriter instance that writes its output to the OutputStream of the page’s Response property.

#### Unload

Unload is called after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and any cleanup is performed.

#### Events Fired Within the Life Cycle

Within each stage of the life cycle of a page, the page raises events that you can handle to run your own code. For control events, you bind the event handler to the event, either declaratively using attributes such as onclick, or in code.

Pages also support automatic event wire-up, meaning that ASP.NET looks for methods with particular names and automatically runs those methods when certain events are raised. If the AutoEventWireup attribute of the @Page directive is set to true, page events are automatically bound to methods that use the naming convention that follow the pattern Page\_<event>, such as Page\_Load and Page\_Init.

The following table lists all the page events. The ”Typical Use” column indicates the events you usually handle when developing pages and controls. The other events are less important but give you more control from the extensibility perspective, for instance when you create custom controls that require a very specific behavior.

Table 1-2. Usage of page events

|  |  |
| --- | --- |
| Page Event | Typical Use |
| PreInit | Use this event to check the IsPostBack property to determine whether this is the first time the page is being processed. Create or re-create dynamic controls. You may also set master pages and the Theme property dynamically. Read or set any profile property values. |
| Init | Event is raised after all controls have been initialized and any skin settings have been applied. Use this event to read or initialize control properties. |
| InitComplete | This event is raised by the Page object. Use this event for processing tasks that require all initialization be complete. |
| PreLoad | Use this event if you need to perform processing on your page or control before the Load event. Before the Page instance raises this event, it loads view state for itself and all controls, and then processes any post-back data included with the Request instance. |
| Load | The Page calls the OnLoad event method on the Page, then recursively does the same for each child control, which does the same for each of its child controls until the page and all controls are loaded. Use the overridden OnLoad event method to set properties in controls and establish database connections. |
| Control events | Use these events to handle specific control events, such as a Button control's Click event or a TextBox control’s Text-Changed event. |
| LoadComplete | Use this event for tasks that require that all other controls on the page be loaded. |
| PreRender | Before this event occurs, the Page object calls EnsureChildControls for each control and for the page itself. Each data bound control whose DataSourceID property is set calls its DataBind method. The PreRender event occurs for each control on the page. Use the event to make final changes to the contents of the page or its controls. |
| SaveStateComplete | Before this event occurs, view state has been saved for the page and for all controls. Any changes to the page or controls at this point will be ignored. See the section about view state later in this chapter to learn more about its specific behavior. Use this event to perform tasks that require view state to be saved, but that do not make any changes to controls. |
| Unload | This event occurs for each control and, when all control events are done, for the page. In controls, use this event to do final cleanup for specific actions, such as closing database connections. For the page itself, use this event to do final cleanup work, such as closing open files and database connections, finishing up logging or other request-specific tasks. |

If the request is a postback, the values of the controls have not yet been restored from view state during PreInit state. If you set a control property at this stage, its value might be overwritten in the next event.

Again, in a postback request during Load event, if the page contains validator controls, check the IsValid property of the Page and of individual validation controls before performing any processing.

During the Unload stage, the page and its controls have been rendered, so you cannot make further changes to the response stream. If you attempt to call a method such as the Response.Write method, the page will throw an exception. The render process itself does not have an event; it happens after PreRender and before UnLoad and can be intercepted by overwriting the Render method.

#### Additional Page Life Cycle Considerations

Server controls have their own life cycle that is similar to the page life cycle. For example, a control’s Init and Load events occur during the corresponding page events. Although both Init and Load recursively occur on each control, they happen in reverse order. The Init and also the Unload event for each child control occur before the corresponding event is raised for its container. This is clearly the expected behavior, because the Init event on page level indicates that all subsequent Init events are completed. You may read the event as “Is Initialized”. However, the Load event for a container occurs before the Load events for its child controls, which is the opposite behavior. This is so that you can have access to the process before any other processing takes place. From perspective of Init event it doesn’t make sense to do anything before it appears, because there is nothing you can access in your code.

You can customize the appearance or content of a control by handling the events for the control, such as the Click event for the Button control, and the SelectedIndexChanged event for the ListBox control.

When inheriting a class from the Page class, in addition to handling events raised by the page, you can override methods from the page’s base class. It’s a common technique to create your own base page classes that derive from Page and to handle common tasks there. For example, you can override the page’s InitializeCulture method to dynamically set culture information. Overriding event handlers or attaching events are two different methods to perform almost the same action. If you use the Page\_<event> syntax you don’t need to handle the base implementation. However, if you override the method you must handle the base implementation and therefore call the base method. You can do this at the beginning, at the end, or anywhere in between.

## Dynamic Controls and Data Binding Events

The page and control life cycles define a chain of events to give you maximum control over the creation and runtime of the control. However, the control handling has more possibilities than you can achieve with such a straightforward model. You will see a different behavior when adding controls dynamically. Furthermore data-bound controls work internally other than this way.

### Dynamic Control Events

The page life cycle and control life cycle are very similar because of the nature of a Page as a class derived from Control. However, you can add controls dynamically—a feature which doesn’t make sense for pages. You might assume that adding controls is not very common in ASP.NET. However, imagine that declaratively authored controls within templates of data-bound controls are treated like dynamically added controls. Their events are not initially synchronized with the other controls on the page. The lifetime of such controls begins with their instantiation; they raise the life cycle events one after the other until the control has caught up to the event during which it was added to the Controls collection.

Usually this has no implications for developers. Unless you have nested data-bound controls, you don’t need to be concerned about this. For nested data-bound controls, this behavior is a bit different. If a child control has been data bound, but its container control has not yet been bound, the data in the child control and the data in its container control can be out of sync.

Imagine that you have a GridView control and that each row contains a bound DropDownList control. Furthermore, assume that the DropDownList’s data properties, such as DataSourceID, are set declaratively. The DropDownList will now bind to its datasource when the DataBinding event of the containing GridView row occurs. However, the GridView might not yet have raised its RowDataBound event. In that case the DropDownList and containing control are out of sync. To avoid this you simply put the data source control—the one you have set in the DataSourceID property—within the same template. Additionally, remove the declarative assignment and set it programmatically during the RowDataBound event. In this event the container’s data are bound and both controls can stay in sync.

### Data Binding Events for Data-Bound Controls

To help you understand the relationship between the page life cycle and data binding events, the following table lists data-related events in data-bound controls. Remember that not all controls support all events shown in the table.

Note: In the description of data-bound control’s behavior I often refer to the term “containing control”. If there is no containing control this assumes that the container is the page.

Table 1-3. Relationship between life cycle events and data-bound events

|  |  |  |
| --- | --- | --- |
| Event | Typical behavior | Usage hints |
| DataBinding | Raised before PreRender of containing control. This is the beginning of binding procedure. | Open database connections here, if required. |
| RowCreated | Raised after each row gets bound. | Manipulate content that does not depend on data binding. |
| ItemCreated | Raised after each item gets bound. | Manipulate content that does not depend on data binding. |
| RowDataBound | Bound data is now available for the row. | Format data, get and filter child rows or related data. |
| ItemDataBound | Bound data is now available for the item. | Format data, pull and filter child rows or related data. |
| DataBound | Marks the end of binding operation. All rows are bound now. | Do any action that requires all data available. |

### Login Control Events

The login controls, which handle login and authentication, are highly sophisticated and powerful. In most scenarios, the controls work well out of the box or just need some configuration using the web.config file. However, if you need to change or customize the behavior or extend the available features, you’ll need to know the events fired during their life cycle.

Table 1-4. Relationship between life cycle events and logging-in events

|  |  |  |
| --- | --- | --- |
| Event | Typical behavior | Usage hints |
| LoggingIn | Raised during postback after LoadComplete. | Tasks required before the login procedure, such as opening a database connection, if required. |
| Authenticate | Raised next after LoggingIn. | Customize the authentication behavior itself. |
| LoggedIn | Raised after authentication. | Action required after successful authentication, such as redirect to another page. |
| LoginError | Raised after failed authentication. | Do any action to handle failed authentication, such as displaying instructions. |

## View State

The ASP.NET view state is a mechanism to track changes of the state of pages from one postback to the next. We call it postback whenever a page sends the contents of a form back to the server. The name comes from the underlying HTTP command POST that we use and the way that it takes the contents back to the server. View state is turned on by default and most developers don’t care about it. It contains the values of properties of controls changed since the last page cycle and is stored in a serialized and encrypted format within the page. Each postback sends all the information stored in it back to the server. As you may know, using view state in the default manner can lead to huge pages that consume a lot of bandwidth. It’s not easy to understand view state completely, and simply disabling it leads to controls not working or having fewer features.

The real internal behavior is confusing and hard to understand. As long as you deal with simple applications, a basic ideamight be enough. But when it comes to your own controls, complex pages, or using AJAX, a good understanding is a prerequisite.

## Why Understanding View State is Important

Misunderstanding view state leads to leaking of sensitive data, being exploited by view state attacks, poor page performance, poor scalability, and headaches. Whatever you want to achieve with your application, the Figure 1-11 shows the source of a page which we never want to see.

Insert ASPEXTf0113.tif

Figure 1-11.The view state’s form field could grow

Simply disabling view state will not help, however, because you will probably lose some features that you need and like. In fact, there is nothing wrong with view state; it’s just the wrong or inappropriate usage of it that causes problems. Now let’s move on by starting back at the beginning. What does view state do?

1. Stores values of control’s properties by keys, similar to a hash table

2. Tracks changes to initial values

3. Serializes and deserializes a saved dictionary in a hidden form field

4. Restores stored values into control’s properties

These features are well known. Much more important is what the view state does not do:

1. Retain the state of class members

2. Remember state information across page loads (page reloads by GET request)

3. Avoid the need to repopulate data on every request

4. Hold data in controls that are posted back—the so-called sticky form behavior

The last bullet point is probably the most confusing here. Many developers think that this is the only task that the view state has to do.Although the view state is involved in the process of holding the control’s visible data; it is not the originator of the process (that would be the control state).

#### What View State is Supposed to Do

Before we start looking into what view state does exactly, some basic features should be reviewed.

View state stores values

* View state tracks changes
* View state serializes and deserializes data

View state restores data

View state is held in a hidden field with the name \_\_VIEWSTATE. “Hidden” means that the browser does not render the field, but the field itself and its content are visible in the page’s source. It needs the bandwidth twice—the first time when the page is sent to the browser and the second time when the browser sends the page back (that is, when a postback happens). A postback occurs very frequently when a user navigates within the application; unless you use hyperlinks, almost all actions are based on a form being sent back.

## The Page Cycle from the View State Perspective

The page cycle is very important to understanding view state completely as several levels of the cycle treat view state explicitly. You can refer to the previous section to get an idea of the page cycle in a nutshell. In this section I will repeat the major steps that actually handle view state.

Each time the server receives a request for an .aspx page it will be processed by the ASP.NET engine through a specific number of steps. These include checking access security and restoring the session state. At the end of the process the engine creates a class that consists of the markup and code-behind. This generates an instance of that class which becomes its ProcessRequest method.

The page life cycle begins with the call to this method. It follows the initialization of the controls of the page and life cycle phases of those controls begin. This includes the processing of the view state, handling of postback data and related events and finally the processing of HTML content.

The life cycle of the page ends when the HTML is completely rendered and handed over to the web server. The web server is responsible for sending the data over the wire to the browser. While the web server is sending the data, the garbage collector is already freeing up the released memory. This is true for all requests without any interaction or custom code required. However, within this cycle, view state has an extraordinary function and it makes your life as a control developer easier if you understand life cycles and their relation to view state.

### Step One—Instantiation

The instantiation is the beginning of the life cycle of the page. The class, which represents the page, is instantiated and launched. But what actually happens and where does the page’s code get stored? ASP.NET pages consist of HTML markup with controls and code. Firstly, the ASP.NET engine converts the text portion of the page and all markup into web controls. That means the page is now completely made of code. Most parts that consist of HTML are replaced by LiteralControl controls. Literals contain text that doesn’t need any further processing. This is required in order to avoid the extensive recognition process. The ASP.NET engine is able to recognize changes made and re-runs the process, if required. The features required for a page are inherited from the Page class (System.Web.UI.Page). If code behind is being used, one more step is required. Using the partial keyword this class is married with the code-behind class that is associated with the aspx page.

When compilation is done with SDK components, the final class is stored under this path:

WINDOWS\Microsoft.NET\Framework\<Version>\Temporary ASP.NET Files

This is same if IIS is calling a web project that is not pre-compiled and the ASP.NET engine is forced to compile the code on the fly.

Using Visual Studio 2008, the path is quite different:

<UserHome>\AppData\Local\Temp\Temporary ASP.NET Files\root\

Under either path, you can find folders for projects and solutions. However, Visual Studio uses some weird codes to name the folders instead of the project’s real name. I suggest sorting the explorer view by date to recognize the right folder based on last change date. The language—either C# or Visual Basic—depends on the setting of the page and the code behind used. Reading the generated pages can occasionally be helpful, especially when researching odd errors. If you want to go deeper into the internal page processing, it’s worth a look. The generated material includes all fragments read from pages, code behind, user controls and master pages.

If the code is compiled directly you may not find the generated classes but the final assembly only. In that case I recommend using a tool like Red Gate’s Reflector to examine the finally generated code.

#### Building the Hierarchy of Controls

The page is the top level of the hierarchy of controls. Because the Page class itself inherits from Control, it’s a control, too. Each control has exactly one parent and none, one, or many children. The elements placed declaratively in the page are children of the highest level. There is no limit to nesting levels for the control hierarchy. This leads to vast trees made of complex controls.

One important control is HtmlForm (full name is System.Web.UI.HtmlControls.HtmlForm). It creates the <form> tag and determines the content sent during postback. The view state information is necessarily placed here. You can see this in the sample shown in Listing 1-4.

Listing 1-4. Simple ASPX page with controls

<html xmlns="http://www.w3.org/1999/xhtml">

<head runat="server">

<title></title>

</head>

<body>

<h1>

Apress - ViewState</h1>

<form id="form1" runat="server">

<div>

<asp:TextBox runat="server" ID="txtName" />

<br />

Are you called a

<asp:DropDownList runat="server" ID="ddlWhat">

<asp:ListItem Value="G" Selected="True">Geek</asp:ListItem>

<asp:ListItem Value="N">Nerd</asp:ListItem>

<asp:ListItem Value="W">Don’t know</asp:ListItem>

</asp:DropDownList>

<br />

<asp:Button runat="server" ID="btnSend" Text="Send!" />

</div>

</form>

</body>

</html>

Figure 1-14 shows how it will look like in the Visual Studio designer.

D:\UserData\jkrause\Documents\Meine Daten\Bücher\Fachbuchprojekte\ASP.NET Extensibility\Images\ASPEXTf0112.tif

Insert Figure ASPEXTf0112.tif

Figure 1-14. Simple page in Visual Studio Designer

The ASP.NET engine generates the output code, a fragment of which is shown in Listing 1-5:

Listing 1-5. Simple ASPX page from last listing compiled into code

public class default\_aspx : \_Default, IRequiresSessionState, IHttpHandler

{

private static object \_\_fileDependencies;

private static bool \_\_initialized;

[DebuggerNonUserCode]

public default\_aspx()

{

base.AppRelativeVirtualPath = "~/Default.aspx";

if (!\_\_initialized)

{

string[] virtualFileDependencies = new string[] { "~/Default.aspx" };

\_\_fileDependencies = ⮰

base.GetWrappedFileDependencies(virtualFileDependencies);

\_\_initialized = true;

}

}

[DebuggerNonUserCode]

private HtmlTitle \_\_BuildControl\_\_control2()

{

return new HtmlTitle();

}

[DebuggerNonUserCode]

private void \_\_BuildControl\_\_control3(ListItemCollection \_\_ctrl)

{

ListItem item = this.\_\_BuildControl\_\_control4();

\_\_ctrl.Add(item);

ListItem item2 = this.\_\_BuildControl\_\_control5();

\_\_ctrl.Add(item2);

ListItem item3 = this.\_\_BuildControl\_\_control6();

\_\_ctrl.Add(item3);

}

[DebuggerNonUserCode]

private ListItem \_\_BuildControl\_\_control4()

{

ListItem item = new ListItem();

item.Value = "G";

item.Selected = true;

item.Text = "Geek";

return item;

}

[DebuggerNonUserCode]

private ListItem \_\_BuildControl\_\_control5()

{

ListItem item = new ListItem();

item.Value = "N";

item.Text = "Nerd";

return item;

}

[DebuggerNonUserCode]

private ListItem \_\_BuildControl\_\_control6()

{

ListItem item = new ListItem();

item.Value = "W";

item.Text = "Don’t know";

return item;

}

[DebuggerNonUserCode]

private Button \_\_BuildControlbtnSend()

{

Button button = new Button();

base.btnSend = button;

button.ApplyStyleSheetSkin(this);

button.ID = "btnSend";

button.Text = "Send!";

return button;

}

[DebuggerNonUserCode]

private DropDownList \_\_BuildControlddlWhat()

{

DropDownList list = new DropDownList();

base.ddlWhat = list;

list.ApplyStyleSheetSkin(this);

list.ID = "ddlWhat";

this.\_\_BuildControl\_\_control3(list.Items);

return list;

}

[DebuggerNonUserCode]

private HtmlForm \_\_BuildControlform1()

{

HtmlForm form = new HtmlForm();

base.form1 = form;

form.ID = "form1";

IParserAccessor accessor = form;

accessor.AddParsedSubObject( ⮰

new LiteralControl("\r\n <div>\r\n "));

TextBox box = this.\_\_BuildControltxtName();

accessor.AddParsedSubObject(box);

accessor.AddParsedSubObject( ⮰

new LiteralControl("\r\n <br />\r\n ⮰

Are you called a \r\n "));

DropDownList list = this.\_\_BuildControlddlWhat();

accessor.AddParsedSubObject(list);

accessor.AddParsedSubObject( ⮰

new LiteralControl("\r\n <br />\r\n "));

Button button = this.\_\_BuildControlbtnSend();

accessor.AddParsedSubObject(button);

accessor.AddParsedSubObject( ⮰

new LiteralControl("\r\n </div>\r\n "));

return form;

}

[DebuggerNonUserCode]

private HtmlHead \_\_BuildControlHead1()

{

HtmlHead head = new HtmlHead("head");

base.Head1 = head;

head.ID = "Head1";

HtmlTitle title = this.\_\_BuildControl\_\_control2();

IParserAccessor accessor = head;

accessor.AddParsedSubObject(title);

return head;

}

[DebuggerNonUserCode]

private void \_\_BuildControlTree(default\_aspx \_\_ctrl)

{

this.InitializeCulture();

IParserAccessor accessor = \_\_ctrl;

accessor.AddParsedSubObject( ⮰

new LiteralControl("\r\n\r\n<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\r\n\r\n<html xmlns=\"http://www.w3.org/1999/xhtml\">\r\n"));

HtmlHead head = this.\_\_BuildControlHead1();

accessor.AddParsedSubObject(head);

accessor.AddParsedSubObject(new LiteralControl("\r\n<body>\r\n <h1>\r\n Apress - ViewState</h1>\r\n "));

HtmlForm form = this.\_\_BuildControlform1();

accessor.AddParsedSubObject(form);

accessor.AddParsedSubObject(⮰

new LiteralControl("\r\n</body>\r\n</html>\r\n"));

}

[DebuggerNonUserCode]

private TextBox \_\_BuildControltxtName()

{

TextBox box = new TextBox();

base.txtName = box;

box.ApplyStyleSheetSkin(this);

box.ID = "txtName";

return box;

}

[DebuggerNonUserCode]

protected override void FrameworkInitialize()

{

base.FrameworkInitialize();

this.\_\_BuildControlTree(this);

base.AddWrappedFileDependencies(\_\_fileDependencies);

base.Request.ValidateInput();

}

[DebuggerNonUserCode]

public override int GetTypeHashCode()

{

return -1678387491;

}

[DebuggerNonUserCode]

public override void ProcessRequest(HttpContext context)

{

base.ProcessRequest(context);

}

// Properties

protected HttpApplication ApplicationInstance

{

get

{

return this.Context.ApplicationInstance;

}

}

protected DefaultProfile Profile

{

get

{

return (DefaultProfile) this.Context.Profile;

}

}

}

The code is not very readable, but this is what generated code is supposed to look like, right? There is no need to do anything special with it, but rather to learn what is going on internally. Note two things here. Firstly, the declarative assignment of attributes is being replaced with simple property assignments. See the following markup:

<asp:Button text="Foobar">

It becomes this in the generated class:

button1.Text = "Foobar"

Secondly, all literal stuff is placed in strings and copied into the final page “as is”, just like shown in the snippet below:

new LiteralControl("\r\n</body>\r\n</html>\r\n"))

Once the class is complete, the engine proceeds to the next step.

### Step Two—Initializing

After creating the control hierarchy, all the controls of the page are moved into the initialization state. This becomes apparent by raising the Init event. In this phase not only are the controls ready but they also having their static, declarative properties assigned.

Note: “Declarative” is everything that’s written within the markup and not in code fragments.

Whether or not a property has been declaratively assigned, you still can change the value any time before the render process is complete. However, because I’m talking about view state, the initialization step plays a special role regarding property values.

#### View State Tracks Changes

View state tracks changes of properties. This is a special function of the StateBag class, which holds the data behind the scenes. Tracking starts during the Initializing step. In order to recognize the default value and the changed one, the StateBag compares the initialized value set by declarative markup with the current value set anytime later programmatically. Even if the results look the same, the view state treats the data in a different way.

#### View State and Dynamic Controls

Using dynamic controls, you’ll know that technically it’s possible to add controls any time before the page render process is complete. View state, is not that flexible. Adding controls using Controls.Add method must take place in the initializing phase. Otherwise the StateBag class does not recognize the Init event at the right time and does not start tracking changes. You still can add controls anytime, but for those controls the view state will not work. You might find problems here occasionally as the majority of controls being added dynamically—such as the Label control—don’t require property value tracking.

The golden rule states that in order to “take care of the view state, whether it’s required or not, add dynamic controls during the Initializing step”.

### Step Three—Loading the View State

View state data is saved into a hidden field called \_\_VIEWSTATE. It’s transmitted back to the server only during a POST operation, which forces a postback.

Note: You can completely ignore page requests that use GET for this step. Such requests do not interact with view state at all.

In this step the engine decodes values from the hidden fields and assigns them to the controls, looping recursively through the control hierarchy. Also in this phase the validity of the view state is checked. There are several reasons why the view state could become invalid. One reason is that the control hierarchy has been changed during postback and the control is not at the expected place, or removed completely.

If everything is fine, the engine proceeds to the next step.

### Step Four—Loading Postback Data

After loading and restoring the view state, the form data is processed. Not all controls can return data. To let the engine know whether it has to look for data sent back, the IPostBackDataHandler interface is recognized. Each control implementing this interface might send form data back. Form data is sent back in the HTTP protocol as id−value pairs:

Myclientid=value

The page class looks for a control with a ClientID that equals “Myclientid”. When found, it checks whether the control implements the IPostBackDataHandler interface. If it does this, the page class calls the only method defined there—LoadPostData. That means that the control itself manages the loading of data, which opens up a great way of changing behavior and adding custom code.

Concerning view state, this behavior is very significant. The ability to get the values of controls back between postbacks is called “sticky form” behavior. It’s one of the best features of ASP.NET. Let me explain this in more detail using the example of a TextBox control. The TextBox has a property Text. It also implements the IPostBackDataHandler interface. Once a page with the textbox control is posted back, the LoadPostData method reads the value of the control out of the view state and writes it into the Text property. The default value—that was probably set declaratively—is overwritten. You can easily check this with the following definition:

<asp:TextBox runat="server" ID="txtName" Text="" />

The render process transforms this markup into HTML form:

<input type="text" id="txtName" name="txtName" />

If the user now enters some text here, like “It’s a geek”, the form transmitted back to the server contains at least this line:

txtName=It’s a geek

This data pair becomes part of the Request.Form collection. The key is “txtName” and the value is “It’s a geek”. The page class hands this data pair to the LoadPostData method. Internally there is nothing surprising here. Using Reflector, you can look into the relevant code inside System.Web.dll.

Listing 1-6. Disassembled code of the LoadPostData method

protected virtual bool LoadPostData(string postDataKey, ⮰

NameValueCollection postCollection)

{

base.ValidateEvent(postDataKey);

string text = this.Text;

string str2 = postCollection[postDataKey];

if (!this.ReadOnly && !text.Equals(str2, StringComparison.Ordinal))

{

this.Text = str2;

return true;

}

return false;

}

This code snippet is from TextBox, and the one and only property filled by a postback is Text. The control might have been set read-only and therefore this is checked first (this.ReadOnly). Also, as the code compares the old value and the new one, it’s written only if the value has been changed. Why is this additional check required? You might assume that comparing takes more time than assigning a property—even if there is nothing to render at this stage. Well, the reason is view state. To understand this, take a look into the code for the Text property:

Listing 1-7. Disassembled code of Text method of the TextBox class

public virtual string Text

{

get

{

string str = (string) this.ViewState["Text"];

if (str != null)

{

return str;

}

return string.Empty;

}

set

{

this.ViewState["Text"] = value;

}

}

The value is not stored in a private field but only in view state. The tracking feature of the underlying StateBag class handles changes as well. If nothing has changed, it’s not recommended to assign the value to view state, or the StateBag class will start tracking. If there is nothing to track, it doesn’t make sense to do so, and the LoadPostData method is clever enough to know.

One major aspect many developers overlook is the role of view state in the handling of postback data. The previous code snippets showed that the postback and form data loading procedures do not involve view state. More than that, they try to avoid any interaction with view state. The form data handling and stickiness behavior has nothing to do with the view state. The stickiness is a feature that comes with the IPostBackDataHandler interface. Just for clarity, here it is in its entire glory:

Listing 1-8. Code of IPostBackDataHandler interface

public interface IPostBackDataHandler

{

bool LoadPostData(string postDataKey, ⮰

NameValueCollection postCollection);

void RaisePostDataChangedEvent();

}

You know now that view state tracks changes and stores these changes. You know, too, that the stickiness of the form has nothing to do with view state.

Now, all the controls have been initialized, properly filled with declarative data, over-written with data posted back from the browser, and are ready to enter the next state.

### Step Five—Load Step

Most descriptions of ASP.NET put user code into the Load event. In the explanation of life cycles you read about this important step. I as it signals to us that the control is now “ready”. Even if you don’t want to do anything else with the control, it’s ready to be rendered. This is why the best step for adding custom code and modifying the behavior of the page is the load step. Internally, it’s much easier, because there is nothing to do. Anything that is supposed to happen here is up to you.

### Step Six—Raising Postback Events

Several controls can fire events, depending on the form data posted back. After the loading step, the events are fired one by one. For example, the Button control can fire a Click event, whereas the DropDownList fires SelectedIndexChanged. This is another major feature that makes the dynamic portion of an ASP.NET page so powerful and easy to program.

There are two kinds of event fired during the page’s postback.

One kind is responsible for changes, indicated by the suffix \_Change. The implementation of IPostBackDataHandler is responsible for recognizing this and firing the appropriate event by calling the RaisePostDataChangedEvent method. In the case of the TextBox example, the event exposed by the class is OnTextChanged.

The other kind of event is the trigger event. For example, in the case of a Button this is a Click event. There is no special state required and no data to compare. To handle trigger events the IPostBackEventHandler interface is used:

Listing 1-9. Code of IPostBackEventHandler interface

public interface IPostBackEventHandler

{

void RaisePostBackEvent(string eventArgument);

}

To understand what happens internally, we’ll look into the RaisePostBackEvent method. Listing 1-10 is for a Button control.

Listing 1-10. Code of typical implementation of IPostBackEventHandler interface

protected virtual void RaisePostBackEvent(string eventArgument)

{

base.ValidateEvent(this.UniqueID, eventArgument);

if (this.CausesValidation)

{

this.Page.Validate(this.ValidationGroup);

}

this.OnClick(EventArgs.Empty);

this.OnCommand(new CommandEventArgs(

this.CommandName, this.CommandArgument));

}

As you can see this method raises more than one event in a predefined and hard-coded order. Several other actions take place first—such as validation—but the events being fired are unconditional.

Again, all these events can only happen during a postback. Calling the page using GET will do nothing. If you have a DropDownlist, and if you change the current index programmatically, nothing will happen unless the page is posted back. This is different from the windows programming model, where the controls can fire events immediately and independently of a particular state.

### Step Seven—Storing the View State

After all events have been fired, the current state of each control’s property changes needs to be stored. The tracked changes are retrieved recursively through the controls hierarchy. To get the value back, the SaveViewState method of each control is called. The collection of data is then serialized and encoded using Base64.

In the next step the string is saved to the hidden field named \_\_VIEWSTATE.

### Step Eight—Render the Page

The render process runs through all the controls and allows them to render. This is done by a recursive call to all RenderControl methods. The HtmlForm control mentioned at the beginning is responsible for creating the \_\_VIEWSTATE hidden field, which is the container for view state data.

## The True Role of View State

The eight states of the life cycle are virtually all important for view state. There are several more steps the various life cycles could run through, but these steps don’t affect view state and we can exclude them for now. To understand the view state, it’s crucial to know what the purpose of view state is. As the name implies, view state stores status information. But what kind of status is it?

The hierarchy of controls and the default values of properties are defined in the declarative part of the page. Take a look at this markup:

<asp:Label runat="server" Text="We learn viewstate" Font-Bold="true" />

Neither the text “We learn viewstate” nor the value Bold of the Font property is stored in view state. The values are assigned during the initializing phase. View state, in contrast, tracks changes made *programmatically*. This leads to the first definition about view state.

Note: View state becomes important only if the page contains custom code.

However, if a page contains custom code, this does not necessarily mean that view state is required. Let’s look into another code snippet. The following example has two buttons. Both cause a postback, but only one has an OnClick event attached and handled to access properties in code.

Listing 1-11. Markup of the view state test

<asp:Label ID="LabelMessage" runat="server" Text="We learn viewstate"></asp:Label>

<br />

<asp:Button ID="ButtonSubmit" runat="server" onclick="ButtonSubmit\_Click"

Text="Change Text" Width="150px" />

<br />

<asp:Button ID="ButtonEmpty" runat="server" Text="No Change"

Width="150px" />

The code of the click handler is simple, too.

Listing 1-12. Code of the Click handler

protected void ButtonSubmit\_Click(object sender, EventArgs e)

{

LabelMessage.Text = "Hello Geek!";

}

At the first call of the page the ASPX markup is processed and all values of properties are set, especially the text “We learn viewstate” for the Label. View state stores nothing and therefore contains only internal information without custom data:

<input type="hidden" name="\_\_VIEWSTATE" id="\_\_VIEWSTATE" ⮰  
 value="/wEPDwULLTExNjMzNDIxNjRkZGn5amjBsOOap6CvRbpUM5D9Mlgo" />

Now click on the button “No Change”. This forces the page to postback and reload, but nothing more happens, because no custom code is involved. Now click on the other button “Change Text”. Again a postback occurs, but now the handler processes the OnClick event. In the code, the text of the label gets a new value. View state is changed to this:

<input type="hidden" name="\_\_VIEWSTATE" id="\_\_VIEWSTATE" va-lue="/wEPDwULLTExNjMzNDIxNjQPZBYCAgMPZBYCAgEPDxYCHgRUZXh0BQtIYWxsbyBHZWVrIWRkZHTeN11LiTv5BJ0xSdey0L0Qsnk8" />

As you see, the coded part is bigger and obviously contains more information. This is no surprise. If you click now on the other button “No Change”, the page loads as expected and no custom code is run. However, the text of the label is still “Hello geek!” There is no custom code—the initializing phase has obviously passed and has set the declaratively assigned values. But the change made programmatically in an earlier postback is still there—it’s persistent.

And that’s really the purpose of view state. During postbacks, it makes programmatic changes to control properties persistent. To check whether this behavior is really managed by view state, just disable it using the @Page directive:

<%@ Page Language="C#" EnableViewState="false" …

The first click on “Change text” still works as expected. The new text appears. However, when you click on “No change”, the postback forces the page to reload and the declarative part of the markup reloads the default values.

Imagine that the “no view state” behavior described above is the intended behavior in most cases. If you have a label, you usually want it to show its default text. In case of an exception you might want to display a different text. But subsequent reloads usually store the state of such an exception by either overwriting the text again or letting the default text appear.

Globally disabling view state is not an option either, because several complex controls use it to handle their behavior properly, such as the GridView control. You need to understand the view state to prevent overuse in case you don’t need it. In case you really need the special behavior the view state provides you use it correctly, then.

## View State Anti-Patterns

Improper use of view state is easily avoided once you really understand how it works.To make things clear, I’ll explain the most frequent misuse scenarios. Examples of how *not* to use view state will help you understand how to use view state correctly:

1. Forcing a default.

2. Persisting static data.

3. Persisting “cheap” data.

4. Initializing child controls.

5. Adding controls dynamically.

6. Initializing dynamically created controls.

I’ll explain each of these scenarios to make things as clear as possible.

### Forcing a Default

Forcing a default vlue is a very common misuse. Fortunately, it’s even easier to fix. It’s also a good demonstration of the KISS[[4]](#footnote-4) principle. The developers of ASP.NET did a tremendous job to give us a powerful and well-designed toolkit. The whole framework is an infrastructure thing and provides stable and reliable blocks of code for basic stuff. Doing too much infrastructure work is a signal that something is going wrong.

Assume that you want to store some private data used in a user control. The following code uses the view state, which, in principle, is not a bad idea.

Listing 1-13. DON’T TRY THIS AT HOME: a wrongly written user control

public class MyControl : WebControl

{

public string MyData

{

get { return ViewState["MyData"] as string; }

set { ViewState["MyData"] = value; }

}

protected override OnLoad(EventArgs e)

{

if (!IsPostBack)

{

this. MyData = Session["Control MyData"] as string;

}

base.OnLoad();

}

}

You may have written code like this before. It’s not a bad style; it’s simply wrong. To understand my harsh judgment, let’s examine the intent of the control. The developer wrote a user control with the public property MyData. This gives other developers access to it in order to put the control into a defined state using markup like this:

<alias:mycontrol runat="server" Text="Show this label" id="myControl1" />

This compiles and runs well without any exceptions. However, it doesn’t work as expected because the MyData property is never set. Instead, the view state of the page starts growing because the view state field stores this private information. From that perspective, it’s doing well; it holds some data. Also, remember the life cycle events. It’s boring to repeat them again and again, but everything in ASP.NET is based on the life cycle. In Listing 1-13 the code is written into the OnLoad method. But the Load stage is too late. View state already tracks changes and now it’s being overwritten. That’s why it doesn’t make sense to write the data there. Let’s improve our example:

Listing 1-16. Save to be used at home: A well written control

public class MyControl : WebControl

{

public string MyData

{

get {

if (ViewState["MyData"] == null)

return Session["Control MyData"] as string;

else

return ViewState["MyData"] as string;

}

set { ViewState["MyData"] = value; }

}

}

This involves less code and it’s working well. The default value in the session is now used without touching view state. Simply putting the control onto the page does not increase the size of view state. OnLoad is out of the way here. The assignment of properties happens in the Initializing phase—early enough to set values properly. In case the user of the control sets the MyData property programmatically, view state is in full cry and stores the changes silently in the hidden field.

### Persisting constant data

A lot of data used in a page’s code never changes during the life of the page or during the user’s session. Assume your application has some “My” section that is customizable by the user. After the user is logged on, his or her name appears at the top of each page. Assume further that a user control is being written to achieve this.

<asp:label id=”lblUser” runat=”server”/>

In the code behind portion the name is set:

public class MyControl : WebControl

{

protected override OnLoad(EventArgs e)

{

lbluser.Text = CurrentUser.Name;

base.OnLoad();

}

}

This is code that would work well. However, it’s doing something under the hood that we don’t want. The label has its own view state and, as a child control of the user control, it will store its changed properties in the parent’s view state. The user control is going to recognize this and store the change of view state. Storing the label’s text value in view state doesn’t make sense as, each time it’s loaded, the value is pulled from the CurrentUser class. The default behavior of controls is an implicit usage of view state. Either you assign it once, and hold data in view state, or you need to prevent it from using view state. The solution I would recommend looks very easy:

<asp:label id="lblUser" runat="server" EnableViewstate="false" />

Disable view state on the control level and it will work like a charm.

### Persisting cheap data

Sometimes you use data that changes frequently, depending on user action or external conditions. Assume you want to show a list of data on your page. It is a small list and it doesn’t change frequently. But it could change or grow at any time. Your site is already using a database; it’s fast and there is nothing wrong with pulling the data from a table.

<asp:dropdownlist runat="server" id="ddlMyData" DataTextField="Name" DataValueField="ID"/>

The (fictitious) code-behind portion could look like this:

public class MyControl : WebControl

{

protected override OnLoad(EventArgs e)

{

if (!IsPostBack)

{

ddlMyData.DataSource = DAL.QueryDdlData();

ddlMyData.DataBind();

}

base.OnLoad();

}

}

This is, again, working well and nobody would see any issues here. However, there is something here that is easily overlooked. Databound controls need to remember their state, for instance, to hold the last selected option of a dropdownlist after postback. I discussed this previously and called this smart behavior “stickiness”. We don’t want to lose the form’s sticky behavior. You might therefore assume that switching view state off is not a solution either. But stickiness is part of the control state; it’s made by checking the form data. It’s not related to view state, as some developers think. The reason we run into trouble with view state is that it grows each time the page reloads.

This time the solution is not so simple. Switching off view state is only the half of the answer:

<asp:dropdownlist runat="server" id=" ddlMyData" DataTextField="Name" DataValueField="ID" EnableViewState="false" />

The code behind is almost the same, except we pull the data every time the control loads (note the missing if statement):

Public class MyControl : WebControl

{

protected override OnLoad(EventArgs e)

{

ddlMyData.DataSource = DAL. QueryDdlData ();

ddlMyData.DataBind();

base.OnLoad();

}

}

Now view state is as clean as possible. But the sticky form behavior has gone, too. Trying this, you might think you’ve found a bug in this book and that the author is wrong about view state.

The fault is not view state itself. It still has no function because the control state is restoring the data from postback values. However, we overwrite this because the OnLoad event is fired after all states are restored. This is, again, a life cycle issue. Understanding the life cycle leads us to the solution. Move the last code fragment as is to another step in the life cycle—OnInit:

public class MyControl : WebControl

{

protected override OnInit(EventArgs e)

{

ddlMyData.DataSource = DAL. QueryDdlData ();

ddlMyData.DataBind();

base.OnInit();

}

}

This solution is both clean and simple. OnInit fires before the control state restores the values. It pulls static data from a fast database, puts it directly into the control, forces the render process before the control restores its state, and everything is fine. The render process is an important step here, made by calling the DataBind method.

Tip: You’ll probably want to avoid heavy database access whenever possible. Instead, you’ll probably store small data portions in XML files and maintain changes there. I would recommend using a database anyway. Modern implementations, like Microsoft SQL Server™, hold frequently requested data in memory. The amount of data requested is small, and the connection is usually 1GB or 10GB Ethernet. A powerful data access layer will also cache the data locally, so that in reality nothing is transferred between the servers. The slowest part of the connection is from the web server to the user. He or she will have ISDN, DSL, or a dial-up modem. Storing the small portion of data in view state sends the data three times over the wire: once when the user loads the page (as visible data in the dropdown control), once in view state, and the third time when the form is sent back (as view state is part of the form’s field collection). Calling a database frequently is not torture; this is the reason for a database.

### Initializing child controls

One paradigm you might hear frequently about ASP.NET is to do things declaratively. Whatever you can do directly within the aspx page, do it! However, this has some implications and limitations, and eventually you must start doing some work programmatically. The trouble is that initializing controls programmatically is not straightforward. As shown before, you can do this in OnLoad, but this could cause the view state to grow and, in any case, it’s not necessary to use view state when merely setting defaults. Even OnInit is not the best solution because view state will still track changes and catch the settings. If you need view state and can’t disable it, you’ll have to live with this behavior. Because the control’s view state tracks changes from the bottom up—say, from the last leaf control in the control hierarchy up to the tree’s root—the OnInit for child controls is done when the current OnInit is called.

<asp:label id="lblDate" runat="server"/>

In the code behind portion we set the text by assigning the current date:

public class MyControl : WebControl

{

protected override OnInit(EventArgs e)

{

lblDate.Text = DateTime.Now.ToLongDate();

base.OnInit();

}

}

Even if this is the earliest event possible, it’s already too late. The label is initialized before the user control, and view state is already tracking changes. This means thatthe date persists in view state, which doesn’t make sense—this is the same situation we faced in the current user name example.

Disabling view state has been discussed before, but this is not always a solution. Perhaps you have tried this:

<asp:label id="lblDate" runat="server" Text="<% = DateTime.Now.ToLongDate() %>"/>

This is not possible, because ASP.NET doesn’t allow the initializing of properties in that way. Using databinding syntax <%# %> is no solution, either.

Let’s assume that for some reason you want to assign the value by code. And you want to use view state for some other reason as well. You may think about OnPreInit, but this event isn’t recursive and appears only on the page level. So, what’s going on here? It’s interesting that the OnInit event behaves slightly differently depending on how you get access to it. The obvious way is to override the event handler. This is easy, well supported in IntelliSense, and the most common solution. You can attach events declaratively, too. As I said before, this is one of the major paradigms of ASP.NET.

<asp:label id="lblDate" runat="server" OnInit="lblDate\_Init" />

This handler is fired before the internal initialization takes place—before the TrackViewState method is called and the view state starts tracking changes. Just set the text in the code behind as you would before:

public class MyControl : WebControl

{

public void lblDate\_Init(object sender, EventArgs e)

{

lblDate.Text = DateTime.Now.ToLongDate();

}

}

Tip: If you’re wondering whether to use declarative events or just wire up to code—remember the basic rule: “Whatever we can do easily and safely the declarative way is our primary programming technique”.

Another solution comes to mind if you’re an experienced developer of object oriented software. Each object starts its life with the call of the object’s constructor. Usually the constructor is the place to initialize the object. However, in ASP.NET the life of controls begins some time later, in the initialization phase. You may subclass the control and override the constructor to invoke code there, but child controls are not yet present. The various events required are fired some time later. With a custom control it would work. This is indeed another powerful solution. By implementing your own label you can access the constructor without needing to do anything elsewhere:

public class MyLabel : Label

{

public MyLabel()

{

this.Text = DateTime.Now.ToLongDate();

}

}

This is a slightly modified label that simply initializes itself at the right time without any side effects. The constructor call assures that any tracking begins later.

Note: The internal “building steps” of an object, like the constructor call, are made before the life time of the object begins. By “life time”, I mean the life of an object within the control’s or page’s life cycle—not the life time of an object within the runtime.

### Attaching Dynamic controls

All controls have a collection of child controls, represented by the property Controls. Some controls, such as Label, have an empty collection, because they can’t render children. All controls inherit the collection from Control base class. This is the common way to access the hierarchy of controls. The class behind the collection is defined:

public class ControlCollection : ICollection, IEnumerable

There are no obvious limits preventing you from adding new controls at any time. With view state, it’s not that easy. View state can track values only for existing controls. To handle dynamic controls, you need to add them on any page load, whether it’s a postback or regular page load cycle. Attaching dynamic controls must happen in the initializing phase (OnInit event), because, at an earlier state, there is no hierarchy of controls to which you could attach anything. View state of those dynamically attached controls is then tracked automatically. Remember that this is done recursively through the whole hierarchy, so the StateBag class will never miss anything. However, this point is critical. You might experience the following exception loading the page after postback:

[Failed loading view state]

Failed to load view state.  The control tree into which view state is being loaded must match the control tree that was used to save view state during the previous request.  For example, when adding controls dynamically, the controls added during a post-back must match the type and position of the controls added during the initial request.

Let’s create an example so that we can figure out what is happenning internally:

protected void Page\_Init(object sender, EventArgs e)

{

if (!IsPostBack)

{

Button myButton = new Button();

form1.Controls.Add(myButton);

myButton.Text = "Click here";

}

else

{

Label label = new Label();

form1.Controls.Add(label);

}

}

The intent of this code is obvious. The user has a Button to invoke some action, and, when the action is completed, the button is replaced by a Label control. You might argue that this is not good practice, but let’s examine it for the moment. View state tries to recognize the elements solely by their index. In this example the control at index [0] is the Button. After postback it has been replaced by the Label, using the same index [0]. This does not lead to the exception shown above, because the restoring code is fairly stupid. It just looks for the right property, and both Button and Label share the same Text property. While this works, the label now shows the text “Click here”. Why? The exception appears even if this less than ideal method fails, probably because the control does not provide the expected property.

The first solution is simple. Turn off view state for the button control. Since you throw away the button anyway, view state is not required at all. If the view state is being saved, it can’t disturb the next cycle—problem solved. Here is the solution in all its glory:

protected void Page\_Init(object sender, EventArgs e)

{

if (!IsPostBack)

{

Button myButton = new Button();

myButton.EnableViewState = false;

form1.Controls.Add(myButton);

myButton.Text = "Click here";

}

else

{

Label label = new Label();

form1.Controls.Add(label);

}

}

Had you been using view state in the sample above, the problem would resolve itself—that is, if you had recreated the control after postback, which is one of the basic rules of dynamic controls creation. The ultimate solution looks like this:

protected void Page\_Init(object sender, EventArgs e)

{

Button myButton = new Button();

myButton.EnableViewState = false;

form1.Controls.Add(myButton);

myButton.Text = "Click here";

if (IsPostBack)

{

Label label = new Label();

form1.Controls.Add(label);

myButton.Visible = false;

}

}

Here we are close to the best practice. Allow all controls on the page to remain untouched and switch the visibility on and off, as required, via code. The render method is smart enough to not render invisible controls. But that’s another topic and would lead us away from view state.

### Initializing dynamically created controls

I discussed dynamic controls before and, reading the text, you might feel that there are some issues with them. This is indeed something we need to approach carefully, but there are no real issues.

The problem is very similar to the one described before. Because you create the control when you choose, you have more influence and this makes your life easier. It does, however, run against the paradigm, because we leave the world of declarative definitions completely. In any case, let’s look how to handle another view state issue correctly.

public class MyCustomControl : Control

{

protected override void CreateChildControls()

{

Label l = new Label();

Controls.Add(l);

l.Text = DateTime.Now.ToLongDate();

}

}

You can create child controls any time, but the CreateChildControls method is the best opportunity for fitting into the event sequence. The initialize event is called and the control doesn’t miss the tracking of view state. The secret is the behavior of the Controls.Add call. This isn’t just a collection; it does much more when the Add method is invoked. Even if all events of the parent hierarchy are complete, the control begins its regular life cycle and all events involved here are fired correctly. That means that the control starts tracking view state immediately after it’s added to the collection. CreateChildControls always seems too late, as even though it is called at different points in time, it’s always later than OnInit. Just for completeness, you need to know that it’s based on EnsureChildControls call, which happens in OnPreRender at the latest. For some reasons the call might come much earlier.

Now take a closer look at the solution:

public class MyCustomControl : Control

{

protected override void CreateChildControls()

{

Label l = new Label();

l.Text = DateTime.Now.ToLongDate();

Controls.Add(l);

}

}

There is only a subtle difference. The control’s properties are set before the control is added, which means that it’s initialized before the OnInit is fired, and that subsequently it has not yet begun to track the view state.

You can also databind controls before they have been added to the parent control’s control hierarchy. This is very powerful and flexible. So real developers write their own custom controls and know how they work internally. Because this is a good topic for extensibility I’ll cover this in this book, too.

# Summary

In this chapter you learned about the internal processing of ASP.NET. Especially we took a look into the processing pipeline, which is running a single request and performs the steps required to create the content send to the client finally. The pipeline forms the life cycle of application, pages, and controls. During the life cycle several states store current processing step’s data. To rescue data from one request to another, the view state is used. You learned what the view state is for and how to overcome the quirks and traps the implemention has.

However, Web applications handle usually more than one request at a time. The next chapter extends the description by looking into threading, thread pools, and other stuff required to make an ASP.NET project fast and reliable even if it comes under pressure from multiple requests.

1. The old assemblies and namespaces are still supported, of course. [↑](#footnote-ref-1)
2. API means Application Programming Interface [↑](#footnote-ref-2)
3. Component Object Model, an interface standard for software componentry [↑](#footnote-ref-3)
4. Keep it Simple, Stupid [↑](#footnote-ref-4)