Web App Follies

Keep Sites Running Smoothly By Avoiding These 10 Common ASP.NET Pitfalls

Jeff Prosise

This article discusses:

- Caching and forms authentication
- View state and session state
- Profile property serialization
- Thread pool saturation
- Impersonation and profiling

This article uses the following technologies:
.NET Framework, ASP.NET, Windows Server 2003

Contents

O
ne of the reasons ASP.NET is successful is that it lowers the bar for Web developers. You don’t need a Ph.D. in computer science to write ASP.NET code. Many of the ASP.NET people I encounter in my work are self-taught developers who wrote Microsoft® Excel® spreadsheets before they wrote C# or Visual Basic®.Now they’re writing Web applications and, in general, they’re doing a commendable job.

But with power comes responsibility, and even veteran ASP.NET developers aren’t immune to mistakes. Years of consulting on ASP.NET projects has shown me that certain mistakes have an uncanny predisposition to keep pitfalls occurring. Some of these mistakes affect performance. Others inhibit scalability. Still others cost development teams precious time tracking down bugs and unexpected behavior.

Here are 10 of the pitfalls that litter the path to releasing your production ASP.NET applications, and what you can do to avoid them. All of the examples draw from my experiences with real companies building real Web applications, and in some cases I provide background by describing some of the problems that the ASP.NET development team encountered along the way.

LoadControl and Output Caching

Rare is the ASP.NET application that doesn’t employ user controls. Before the advent of Master Pages, developers employed user controls to factor out common content, such as headers and footers. Even in ASP.NET 2.0, user controls provide an effective means for encapsulating content and behavior, and for dividing pages into regions whose ability to be cached can be controlled independently of the page as a whole—a special form of output caching known as fragment caching.

User controls can be loaded declaratively or imperatively. Imperative loading relies on Page.LoadControl, which

instantiates a user control and returns a Control reference. If the user control contains custom type members (for example, public properties), then you can cast that reference and access the custom members from your code. The user control in Figure 1 implements a property named BackColor. The following code loads the user control and assigns a value to BackColor:

```
protected void Page_Load(object sender, EventArgs e)
{
    // Load the user control and add it to the page
    Control control = LoadControl("~/MyUserControl.ascx");
    PlaceHolder1.Controls.Add(control);

    // Set its background color
    ((MyUserControl)control).BackColor = Color.Yellow;
}
```

This code, simple as it is, is a trap waiting to ensnare the unwary developer. Can you identify the flaw?

If you guessed that the problem is related to output caching, you are correct. These code samples compile and run fine as shown, but try adding the following (perfectly legal) statement to MyUserControl.ascx:

```
<%@ OutputCache Duration="5" VaryByParam="None" %>
```

Next time you run the page, you'll be greeted by an InvalidCastException (oh joy!) accompanied by the following error message:

```
"Unable to cast object of type 'System.Web.UI.PartialCachingControl' to type 'MyUserControl'."
```

So, here's code that works great without an OutputCache directive, but bombs when an OutputCache directive is added. ASP.NET isn't supposed to work this way. Pages (and controls) are supposed to be agnostic towards output caching. So what gives?

The problem is that when output caching is enabled for a user control, LoadControl no longer returns a reference to an instance of the control; instead, it returns a reference to an instance of PartialCachingControl that might or might not wrap a control instance, depending on whether the control's output is cached. Consequently, developers who call LoadControl to load a user control dynamically and who also cast the control reference in order to access control-specific methods and properties must take care in how they do it in order for the code to work with or without an OutputCache directive.

Figure 2 demonstrates the proper way to load user controls dynamically and cast the returned control references. Here's a synopsis of how it works:

- If the ASCX file lacks an OutputCache directive, LoadControl returns a MyUserControl reference. Page_Load casts the reference to MyUserControl and sets the control's BackColor property.
- If the ASCX file includes an OutputCache directive and the control's output isn't cached, LoadControl returns a reference to a PartialCachingControl whose CachedControl property contains a reference to the underlying MyUserControl. Page_Load casts PartialCachingControl.CachedControl to MyUserControl and sets the control's BackColor property.
- If the ASCX file includes an OutputCache directive and the control's output is cached, LoadControl returns a reference to a PartialCachingControl whose CachedControl property is null. Seeing this, Page_Load does nothing more. It's powerless to set the control's BackColor property because the control's output is delivered from the output cache. In other words, there is no MyUserControl on which to set a property.

The code in Figure 2 will work with or without an OutputCache directive in the .ascx file. It's not pretty, but it averts nasty surprises. Simpler doesn't always equate to more maintainable.

Sessions and Output Caching

Speaking of output caching, there's a potential issue with ASP.NET 1.1 and ASP.NET 2.0 that affects output-cached pages on servers running on Windows Server™ 2003 and IIS 6.0. I've personally seen this manifest itself on production ASP.NET servers twice, and both times it was resolved by turning off output caching. I later learned there's a better solution that doesn't require output caching to be disabled. Here's how I first encountered the problem.

The saga began when a dot-com—let's call it Contoso.com—that runs a public e-commerce app on a small ASP.NET Web farm contacted my team and complained that they were experiencing "cross-threading" errors. Every now and then, a customer using the Contoso.com Web site would suddenly lose the data they had entered and instead would see data corresponding to another user. A bit of sleuthing revealed that cross-threading wasn't an accurate description; "cross-session" errors was more like it. It seems that Contoso.com was storing data in session state, and for some reason users were occasionally—and randomly—being connected to other users' sessions.

One of my team members wrote a diagnostic tool to log key elements of each HTTP request and response, including cookie headers. Then he installed it on Contoso.com's Web servers and let it run for a few days. The results were remarkable. Roughly once in every 100,000 requests, ASP.NET was correctly assigning a session ID to a brand new session and returning the session ID in a Set-Cookie header. It would then return the same session ID (that is, the same Set-Cookie header) in the very next request, even if that request was already associated with a valid session and was correctly submitting the session ID in a cookie. In effect, ASP.NET was randomly switching users away from their own sessions and connecting them to other sessions.

Astonished, we began to look for causes. We first examined Contoso.com's source code and satisfied ourselves that the problem lay elsewhere. Next, just to be sure the problem wasn't related to the fact that the application was hosted on a Web farm, we turned off all the servers but one. The problem persisted, which wasn't surprising since our logs showed that the matching Set-Cookie headers never came from two different servers. It wasn't credible that ASP.NET accidentally generated duplicate session IDs because it uses the .NET Framework RNGCryptoServiceProvider class to generate those IDs, and session IDs are of sufficient length to ensure that the same one will never be generated twice (not in the next trillion years, anyway). Besides, even if RNGCryptoServiceProvider was erroneously generating duplicate random numbers, that wouldn't explain why ASP.NET mysteriously replaced valid session IDs with new (and non-unique) ones.

On a hunch, we decided to look at output caching. When OutputCacheModule caches HTTP responses, it must be careful not to cache Set-Cookie headers; otherwise, a cached response containing a new session ID would connect all recipients of the cached response (as well as the user whose request generated the cached response) to the same session. We checked the source code; Contoso.com had output caching enabled in two pages. We turned it off. Lo and behold, the application ran for days without a single cross-session incident. It has run without error for more than two years since. And we saw the exact same scenario play out at a different company with a different application and a different set of Web servers. As at Contoso.com, eliminating output caching made the problem go away.

Microsoft has since confirmed that this behavior stems from a problem in OutputCacheModule. (There may be an update available by the time you read this.) When ASP.NET is paired with IIS 6.0 and kernel-mode caching is enabled, OutputCacheModule sometimes fails to strip Set-Cookie headers from the cached responses it passes to Http.sys. Here's the specific sequence of events that causes the bug to manifest itself:

1. A user who hasn't visited the site recently (and therefore doesn't have a corresponding session) requests a page for which output caching is enabled, but whose output isn't currently available in the cache.
2. The request executes code that accesses the user's newly created session, causing a session ID cookie to be returned in a Set-Cookie header in the response.
3. OutputCacheModule provides the output to Http.sys, but fails to strip the Set-Cookie header from the response.
4. Http.sys returns the cached response in subsequent requests, inadvertently connecting other users to the session.

The moral of the story? Session state and kernel-mode output caching don't mix. If you use session state in a page that has output caching enabled, and if the application runs on IIS 6.0, then you need to turn off kernel-mode output caching. You'll still get the benefit of output caching, but because kernel-mode output caching is substantially faster than ordinary output caching, the caching won't be as effective. For more information on this issue, see support.microsoft.com/kb/917072.

You can turn off kernel-mode output caching for individual pages by including VaryByParam="*" attributes in the page's OutputCache directives, although doing so can cause memory requirements to explode. The safer alternative is to turn off kernel-mode caching for the entire application by including the following element in web.config:

```
<httpRuntime enableKernelOutputCache="false" />
```

You can also disable kernel-mode output caching globally—that is, for entire servers—with a registry setting. For details, see support.microsoft.com/kb/820129.

Whenever I hear about inexplicable things happening with sessions, I ask the customer if they're using output caching in any of their pages. If the answer is yes, and if the host OS is Windows Server 2003, then I advise them to disable kernel-mode output caching. The problem usually goes away. If it doesn't, then the bug is in their code. Be warned!

Forms Authentication Ticket Lifetime

Can you spot the problem with this code:

```
FormsAuthentication.RedirectFromLoginPage(username, true);
```

As innocuous as it seems, this code should never be used in an ASP.NET 1.*x* app unless there is mitigating code elsewhere in the application to counteract this statement's debilitating effects. If you're not sure why, then read on.

FormsAuthentication.RedirectFromLoginPage performs two tasks. First, it redirects a user to the page they originally requested when they were redirected by FormsAuthenticationModule to the login page. Second, it issues an authentication ticket—typically carried in a cookie, and always carried in a cookie in ASP.NET 1.*x*—that allows the user to remain authenticated for a predetermined period of time.

The problem is the period of time. In ASP.NET 1.*x*, passing RedirectFromLoginPage a second parameter equal to false issues a temporary authentication ticket that expires, by default, after 30 minutes. (You can change the time-out period using a timeout attribute in web.config's <forms> element.) Passing a second parameter equal to true, however, issues a persistent authentication ticket that's valid for—get this—50 years! That's an accident waiting to happen, because if someone steals that authentication ticket, they can access the Web site using the victim's identity for the life of the ticket. There's no shortage of ways to swipe authentication tickets—sniffing unencrypted traffic at public wireless access points, cross-site scripting, gaining physical access to a victim's PC, and so on—so passing true to RedirectFromLoginPage is little better than disabling security on your Web site. Fortunately, this problem was fixed in ASP.NET 2.0. Today's RedirectFromLoginPage honors the timeout specified in web.config for temporary and persistent authentication tickets alike.

One solution is to never pass true in RedirectFromLoginPage's second parameter in ASP.NET 1.*x* apps. But that's not very practical because login pages typically feature a "Keep me signed in" box that users can check to receive persistent rather than temporary authentication cookies. An alternate solution is a snippet of code in Global.asax (or, if you prefer, an HTTP module) that modifies cookies containing persistent authentication tickets before they go back to the browser.

Figure 3 contains one such snippet. If present in Global.asax, this code modifies the Expires property of outgoing persistent forms authentication cookies so that the cookies expire after 24 hours. By modifying the line commented "New expiration date," you can set the timeout to whatever you like.

You may find it curious that the Application_EndRequest method calls a local helper method (GetCookieFromResponse) to check outgoing responses for authentication cookies. The helper method is a work-around for another bug in ASP.NET 1.1 that causes a bogus cookie to be added to the response if you check for a nonexistent cookie using HttpCookieCollection's string indexer. Using the integer indexer as GetCookieFromResponse circumvents the problem.

View State: The Silent Perf Killer

In some ways, view state is the greatest thing since sliced bread. After all, it's view state that allows pages and controls to persist state across postbacks. That's why you don't have to write code to keep the text in a TextBox from disappearing when a button is clicked as you did in classic ASP, or requery a database and rebind a DataGrid following a postback.

But view state has a dark side, too: when it grows too large, it's a silent performance killer. Some controls, such as TextBoxes, are judicious with view state. Others, notably DataGrids and GridViews, emit view state in proportion to the amount of information displayed. I cringe when I see a GridView displaying 200 or 300 rows of data. Even though ASP.NET 2.0 view state is roughly half the size of ASP.NET 1.*x* view state, one lousy GridView can easily cut the effective bandwidth of a connection between a browser and a Web server by 50 percent or more.

You can turn off view state for individual controls by setting EnableViewState to false, but some controls, particularly DataGrids, lose some of their functionality when denied the freedom to use view state. A much better solution to taming view state is keeping it on the server. In ASP.NET 1.*x*, you can override a page's LoadPageStateFromPersistenceMedium and SavePageStateToPersistenceMedium methods and handle view state however you like. The overrides shown in the code in Figure 4 prevent view state from being persisted in a hidden field and persist it in session state instead. Storing view state in session state is particularly effective when paired with the default session state process model—that is, when session state is stored in memory in the ASP.NET worker process. If session state is stored in a database instead, then only testing will show whether keeping view state in session state improves or degrades performance.

The same technique works in ASP.NET 2.0, but ASP.NET 2.0 offers a simpler means for persisting view state in session state. You begin by defining a custom page adapter whose GetStatePersister method returns an instance of the .NET Framework SessionPageStatePersister class:

```
public class SessionPageStateAdapter :
    System.Web.UI.Adapters.PageAdapter
{
    public override PageStatePersister GetStatePersister ()
    {
        return new SessionPageStatePersister(this.Page);
    }
}
```

Then you register the custom page adapter as the default page adapter by dropping an App.browsers file like the following into the application's App_Browsers folder:

```
<browsers>
  <browser refID="Default">
    <controlAdapters>
      <adapter controlType="System.Web.UI.Page"
        adapterType="SessionPageStateAdapter" />
    </controlAdapters>
  </browser>
</browsers>
```

(You can name the file anything you like as long as it has a .browsers extension.) Thereafter, ASP.NET will load the page adapter and use the returned SessionPageStatePersister to persist all page state, including view state.

One downside to using a custom page adapter is that it acts globally for every page in the application. If you'd prefer to persist view state in session state for some pages but not for others, use the technique shown in Figure 4. Additionally, you can run into issues with this technique if a user creates multiple browser windows within the same session.

SQL Server Session State: Another Perf Killer

ASP.NET makes it easy to store session state in databases: just flip a switch in web.config and session state magically moves to a back-end database. This is a critical feature for applications that run on Web farms because it allows every server in the farm to share a common repository for session state. The added database activity slows the performance of individual requests, but the loss in performance is offset by an increase in scalability.

This is all fine and good until you take a moment to ponder the following points:

- Even in an application that uses session state, most pages do not use session state.
- By default, the ASP.NET session state manager performs two accesses—one read access and one write access—to the session data store in every request, regardless of whether the page requested uses session state.

In other words, when you use the SQL Server™ session state option, you pay the price (two database accesses) in every request—even in requests for pages that do nothing with session state. This directly (and adversely) affects the throughput of the entire site.

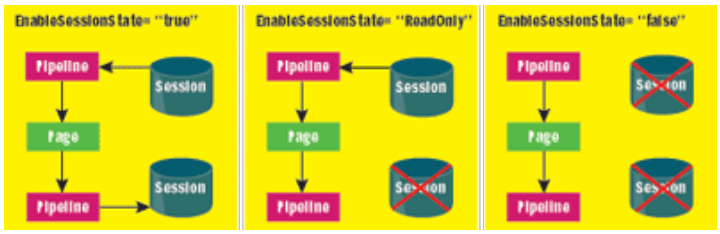

Figure 5 Eliminating Unnecessary Session State Database Accesses

So what do you do about it? Simple: you disable session state in pages that don't use it. It's always a good idea to do so, but it's especially important when session state is stored in a database. Figure 5 shows how to disable it. If a page doesn't use session state at all, include an EnableSessionState="false" in its Page directive, like so:

```
<%@ Page EnableSessionState="false" ... %>
```

This directive prevents the session state manager from reading and writing the session state database in each request. If a page reads data from session state but doesn't write it (that is, doesn't modify the contents of the user's session), then set EnableSessionState to ReadOnly, as shown here:

```
<%@ Page EnableSessionState="ReadOnly" ... %>
```

Finally, if a page requires read/write access to session state, then either omit the EnableSessionState attribute or set it to true:

```
<%@ Page EnableSessionState="true" ... %>
```

By taming session state in this way, you'll ensure that ASP.NET only accesses the session state database when it really needs to. Eliminating unnecessary database accesses is the first step toward building high-performance applications.

Incidentally, the EnableSessionState attribute is no secret. It's been documented since ASP.NET 1.0, yet I rarely see developers take advantage of it. Perhaps this is because it's not terribly important with the default in-memory session state model. But it's critical with the SQL Server model.

Uncached Roles

The following statement is frequently found in the web.config files of ASP.NET 2.0 applications and in samples introducing the ASP.NET 2.0 role manager:

```
<roleManager enabled="true" />
```

As presented, however, this statement can have a demonstrably negative impact on performance. Do you know why?

By default, the ASP.NET 2.0 role manager doesn't cache roles data. Instead, it consults the roles data store each time it needs to determine which roles, if any, a user belongs to. This means that once a user is authenticated, any page that utilizes role data—for example, pages that use site maps with security trimming enabled, and pages to which access is restricted using role-based URL directives in web.config—causes the role manager to query the roles data store. If roles are stored in a database, that's one more database access per request that you can easily do without. The solution is to configure the role manager to cache roles data in cookies:

```
<roleManager enabled="true" cacheRolesInCookie="true" />
```

You can use other <roleManager> attributes to control the characteristics of role cookies—for example, how long the cookies should remain valid (and consequently how frequently the role manager will go back to the roles database). Role cookies are signed and encrypted by default, so the security risk, while not zero, is mitigated.

Profile Property Serialization

The ASP.NET 2.0 profile service provides a ready-made solution to the problem of persisting per-user state, such as personalization preferences and language preferences. To use the profile service, you define an XML profile containing the properties you want to persist on behalf of individual users. ASP.NET then compiles a class containing the same properties and provides strongly typed access to class instances via the profile property added to the page.

Profiles are flexible enough to allow even custom data types to be used as profile properties. But therein lies the problem—one that I've personally seen trip developers up. Figure 6 contains a simple class named Posts, as well as a profile definition that uses Posts as a profile property. However, this class and this profile produce unexpected behavior at run time. Can you determine why?

The problem is that Posts contains a private field named _count that must be serialized and deserialized to fully hydrate and rehydrate class instances. But _count doesn't get serialized and deserialized because it's private and, by default, the ASP.NET profile manager uses XML serialization to serialize and deserialize custom types. The XML serializer ignores nonpublic members. Therefore, instances of Posts will get serialized and deserialized, but each time a class instance is deserialized, _count is reset to 0.

One solution is to make _count public rather than private. Another is to wrap _count with a public read/write property. The best solution, and one that preserves the design of the class itself, is to mark Posts as serializable (using the SerializableAttribute) and to configure the profile manager to use the .NET Framework binary serializer to serialize and deserialize class instances. The binary serializer, unlike the XML serializer, serializes fields, regardless of accessibility. Figure 7 shows a fixed version of the Posts class and the accompanying profile definition with changes highlighted.

The thing you should remember is that if you use a custom data type as a profile property, and if that data type has nonpublic data members that must be serialized in order to fully serialize type instances, then use a serializeAs="Binary" attribute in the property declaration and make sure the type itself is serializable. Otherwise, complete serialization will not occur and you'll waste time trying to determine why the profile isn't working.

Thread Pool Saturation

I am often amazed at the number of real-life ASP.NET pages I see that perform database queries and wait 15 seconds or more for the queries to return. (I've also seen queries that take 15 minutes!) Sometimes the delay is an unavoidable by-product of the volume of data returned; other times it's the result of poor database design. But whatever the reason, long database queries or lengthy I/O operations of any kind are throughput killers in an ASP.NET app.

I've written about this problem at length before, so I won't belabor the point here. Suffice it to say that ASP.NET

relies on a finite thread pool to process requests, and if all the threads are occupied waiting for database queries, Web service calls, or other I/O operations to complete, additional requests must be queued up until an operation completes and a thread is freed. Performance drops off precipitously when requests are queued. And if the queue fills up, ASP.NET fails subsequent requests with HTTP 503 errors. This is not a situation a production app on a production Web server ever wants to find itself in.

The solution, of course, is asynchronous pages—one of the best but lesser-known features of ASP.NET 2.0. A request for an asynchronous page begins its life on one thread, but returns that thread and an IAsyncResult interface to ASP.NET when it begins an I/O operation. When the operation completes, the request signals ASP.NET through IAsyncResult and ASP.NET pulls another thread from the pool and finishes processing the request. Significantly, no thread pool threads are consumed while the I/O operation takes place. This can dramatically improve throughput by preventing requests for other pages—pages that don't perform lengthy I/O operations—from waiting in a queue.

You can read all about asynchronous pages in the October 2005 issue of *MSDN®Magazine*. Any page that is I/O-bound rather than compute-bound and that takes more than a few seconds to execute is a candidate to be an asynchronous page.

When I tell developers about asynchronous pages, they often reply, "That's cool, but I don't really need them in my app." To which I reply, "Do any of your pages query a database? Do any of them call a Web service? Have you checked the ASP.NET performance counters for stats regarding queued requests and average wait times? Even if your app runs fine today, is it possible that the load on it will increase as your customer base grows?"

The reality is that few real-world ASP.NET apps have no need whatsoever for asynchronous pages. Write it down!

Impersonation and ACL Authorization

Here's a simple configuration directive, but one that raises my eyebrows whenever I see it in web.config:

```
<identity impersonate="true" />
```

This directive enables client impersonation in an ASP.NET app. It attaches access tokens representing clients to the threads that process requests so that any security checks performed by the operating system act against the client's identity rather than the worker process's identity. Impersonation is rarely necessary in an ASP.NET app; my experience is that when developers enable it, they usually do so for the wrong reasons. Here's why.

Too often, developers enable impersonation in ASP.NET apps so they can use file system permissions to restrict access to pages. If Bob doesn't have permission to view Salaries.aspx, then the devs enable impersonation so they can prevent Bob from viewing Salaries.aspx by setting the access control list (ACL) to deny Bob read permission. But here's the kicker: impersonation isn't necessary for ACL authorization. When Windows authentication is enabled in an ASP.NET app, ASP.NET automatically checks the ACL for each .aspx page requested and denies the request if the caller lacks permission to read the file. It does so even if impersonation is disabled.

There are times when impersonation is justified. But you can usually avoid it with good design. For example, suppose Salaries.aspx queries a database for salary info that should only be available to managers. By impersonating, you can use database permissions to deny non-managers the ability to query for salary data. Or you can forget about impersonation and restrict access to salary data by setting the ACL for Salaries.aspx so that non-managers lack read permission. The latter approach provides better performance because it avoids impersonation altogether. It eliminates unnecessary database accesses, too. Why query a database only to be denied for security reasons?

Incidentally, I once helped troubleshoot a classic ASP app that periodically rebooted due to unconstrained memory consumption. A junior developer had turned a targeted SELECT statement into a SELECT *, not considering the fact that the table being queried contained images—large images, and lots of them. The problem was exacerbated by an undetected memory leak. (My kingdom for managed code!) Suddenly, an app that had run fine for years began grinding to a halt because SELECT statements that previously returned a kilobyte or two of data now returned several megabytes. Coupled with inadequate version control, it made life pretty exciting for the development team—exciting, that is, if you consider sleeping at night and seeing your kids' soccer games boring.

In theory, classic memory leaks can't happen in ASP.NET applications composed entirely of managed code. But inefficient memory usage impacts performance by forcing garbage collections to occur more frequently. Even in ASP.NET apps, beware of SELECT *!

Don't Just Trust It—Profile Your Database!

As a consultant, I'm often called in when an app isn't performing the way it should. Recently my team was asked to determine why an ASP.NET app was only achieving about 1/100th of the throughput (requests per second) that the requirements document called for. What we discovered was typical of what we find in underperforming Web apps—and a lesson that all of us can take to heart.

We ran SQL Server Profiler and watched the interactions between the app and the database on the back end. In one of the more extreme cases, a simple button click produced more than 1,500 trips to the database. You can't build a high-performance application that way. Good architecture always begins with good database design. No matter how efficient your code is, it's hamstrung if saddled with a poorly written database.

Bad data access architectures typically stem from one or more of the following:

- Poor database design (usually designed by developers, not database administrators).
- Use of DataSets and DataAdapters—in particular, DataAdapter.Update, which is great for Windows Forms apps and other thick clients, but usually not ideal for Web apps.
- Poorly designed data access layers (DALs) with poor factorization and too many CPU cycles expended performing relatively simple operations.

Problems must be identified before they can be treated. And the way to identify data access problems is to run SQL Server Profiler or an equivalent tool to see what's going on behind the scenes. Performance tuning isn't complete until you've examined the traffic between the app and the database. Try it—you may be surprised by what you find.

Conclusion

Now you've seen some of the problems and solutions you're likely to encounter in the process of building production ASP.NET apps. The next step is to take a close look at your own code and try to avoid some of the issues I've outlined here. ASP.NET may lower the bar for Web developers, but there's no reason your apps can't be slick, solid, and speedy. Put on your thinking cap and avoid making those rookie mistakes.

Figure 8 provides a short checklist you can use to avoid the pitfalls described in this article. You can build a similar checklist for security pitfalls. For example:

- Have you encrypted configuration sections containing sensitive data?
- Are you checking and validating input used in database operations, and are you HTML-encoding input used as output?
- Do any of your virtual directories contain files with unprotected extensions?

These are important questions to ask if you value the integrity of your Web site, the servers that host your site, and the back-end resources upon which these rely.

Jeff Prosise is a contributing editor to MSDN Magazine and the author of several books, including Programming Microsoft .NET (Microsoft Press, 2002). He is also a cofounder of Wintellect, a software consulting and education firm.
