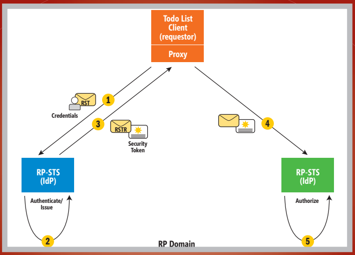
The Microsoft Journal for Developers

msdn® magazine

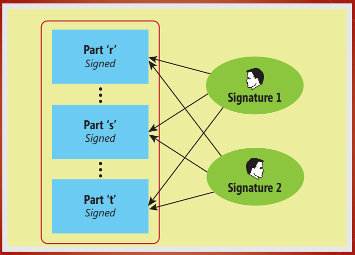


CLAIMS-BASED APPS WITH WIF

Michele Leroux Bustamante page 34

AD FS 2.0 IN IDENTITY SOLUTIONS

Zulfiqar Ahmed page 50



DIGITAL SIGNATURES

Jack Davis page 60

N-TIER APPS AND THE EF

Daniel Simmons page 68

THIS MONTH at msdn.microsoft.com/magazine:

WORKFLOW ESSENTIALS IN SHAREPOINT 2010
Paul Andrew

SANDBOXED SOLUTIONS IN SHAREPOINT 2010
Paul Stubbs

INSIDE MICROSOFT PATTERNS & PRACTICES: CUSTOMIZING WORK ITEMS
Chris Tavares

USABILITY IN PRACTICE: SEARCH IS KEY TO FINDABILITY
Charles B. Kreitzberg & Ambrose Little

NOVEMBER 2009 VOL 24 NO 11

msdn™

Microsoft®

COLUMNS

Claims-Based Apps ◆ AD FS 2.0 ◆ Digital Signatures ◆ *N*-Tier Apps and the EF   **MSDN Magazine**   **NOVEMBER 2009** Vol 24 No 11

Microsoft msdn magazine
NOVEMBER 2009 VOLUME 24 NUMBER 11

LUCINDA ROWLEY Director

EDITORIAL: mmeditor@microsoft.com
HOWARD DIERKING Editor-in-Chief

WEB SITE
MICHAEL RICHTER Webmaster

CONTRIBUTING EDITORS Don Box, Keith Brown, Dino Esposito, Juval Lowy, Dr. James McCaffrey, Fritz Onion, John Papa, Ted Pattison, Charles Petzold, Jeff Prosise, Jeffrey Richter, John Robbins, Aaron Skonnard, Stephen Toub

MSDN Magazine (ISSN # 1528-4859) is published monthly by TechWeb, a division of United Business Media LLC., 600 Community Drive, Manhasset, NY 11030 516-562-5000. Periodicals Postage Paid at Manhasset, NY and at additional mailing offices. Back issue rates: U.S. $10. All others: $12. Basic one-year subscription rates: U.S. $45. Canada and Mexico $55. Registered for GST as TechWeb, a division of United Business Media LLC., GST No. R13288078, Customer No. 2116057 Canada Post: Publications Mail Agreement #40612608. Canada Returns to be sent to Bleuchip International, P.O. Box 25542, London, ON N6C 6B2. All other foreign orders $70, must be prepaid in U.S. dollars drawn on a U.S. bank. Circulation Department, MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081, fax 847-763-9583. Subscribers may call from 8:00 AM to 4:30 PM CST M-F. In the U.S. and Canada 888-847-6188; all others 847-763-9606. U.K. subscribers may call Jill Sutcliffe at Parkway Gordon 01-49-1875-386. Manuscript submissions and all other correspondence should be sent to MSDN Magazine, 6th Floor, 1290 Avenue of the Americas, New York, NY 10104. Copyright © 2009 Microsoft Corporation. All rights reserved; reproduction in part or in whole without permission is prohibited.

POSTMASTER: Send address changes to MSDN Magazine, P.O. Box 1081, Skokie, IL 60076-8081

READERS: Order, renew, or make payment for subscriptions; order back issues; and submit customer service inquiries via the Web at http://msdn.microsoft.com/msdnmag/service.

PUBLISHER:
Jill Thiry jthiry@techweb.com
650-728-7368

United Business Media

BPA WORLDWIDE CONSUMER

ADVERTISING SALES:
Ed Day Western US Accounts/eday@techweb.com/785-838-7547
Brenner Fuller Eastern US Accounts/bfuller@techweb.com/603-746-3057
Michele Hurabiell Key Accounts/mhurabiell@techweb.com/415-378-3540
Wright's Reprints 877-652-5295

ONLINE SERIVCES:
Meagon Marshall Online Accounts & Programs/mmarshall@techweb.com/
785-838-7524

MANUFACTURING:
Pete C. Scibilia Production Manager/pscibili@ubm-us.com/516-562-5134

MARKETING:
Jon Guerringue Marketing/jguerringue@techweb.com/516-562-5957
Laura Robison Marketing/lrobison@techweb.com/415-947-6182

AUDIENCE AND SUBSCRIBERS SERVICES:
Karen McAleer Audience Development Director/kmcaleer@techweb.com/
516-562-7833
Andrew Athanasiou Audience Development Assistant/aathansiou@techweb.com/
516-562-7981
SUBSCRIBER SERVICES: 800-677-2452

TechWeb, a division of United Business Media LLC.–The Global Leader in Business Technology Media
Tony L. Uphoff CEO
Bob Evans SVP and Content Director
Eric Faurot SVP, Live Events Group
Joseph Braue SVP, Light Reading Communications Group
John Siefert VP and Publisher, InformationWeek and TechWeb Network
Scott Vaughan VP, Marketing Services
John Ecke VP, Financial Technology Group
Jill Thiry Publisher, MSDN Magazine and TechNet Magazine
John Dennehy General Manager
Fritz Nelson Executive Producer, TechWeb TV
Scott Popowitz Senior Group Director, Audience Development
Beth Rivera Senior Human Resources Manager

Printed in the USA

Beginnings and Endings

As the title indicates, this month's Editor's Note is about beginnings and endings. To begin with, I want to share with you just a few of the myriad changes that we have recently implemented across both the MSDN Web sites and the MSDN Subscriptions program. Starting with the Web sites, you can see that we have made some tremendous updates to the user experience not just with respect to aesthetics but also to some of the more fundamental metaphors for navigating and interacting with the site itself, the content and the people behind the content. Even the MSDN Library has been overhauled to include a more understandable and more responsive user experience option called "scriptless." Over time, expect to see many more improvements and innovations made in this arena.

The MSDN Subscriptions program is also going to have some really cool new benefits added to it. The benefit that I'm most excited about: Your MSDN Subscription will include a cloud computing sandbox environment on Windows Azure. This sandbox includes everything from the Azure computing platform to SQL Azure to the various elements of the .NET Services stack. For example, for MSDN Premium subscribers who sign up by June 30, 2010, your Azure sandbox would include the following for eight months:

Windows Azure	• 750 compute hours/month • 10GB storage • 1M storage transactions
SQL Azure	• Web Edition (1GB db)—3 databases
.NET Services	• 1M messages/month
Data Transfers	• 7GB in, 14GB out /month (Europe and North America) • 2.5GB in, 5GB out (Asia Pacific)

Like I said, I am excited to see this particular benefit added to the MSDN subscription program because I believe giving cloud computing tools equal visibility and access as their more established desktop and Web counterparts is an essential element in bringing the cloud into the mainstream of thinking about not just application deployment, but design and development as well.

Finally, in order to help you be successful with all of these new tools that you will have access to, the MSDN subscription will now also include a collection of Microsoft e-learning, which is about 20 hours worth of instruction. And naturally, you'll also continue to have access to *MSDN Magazine*!

And there's one more change to tell you about. I have greatly enjoyed my tenure as editor in chief for *MSDN Magazine*. I feel as though we've continued to make some forward progress in improving the way that people think about designing and constructing software. However, over the last several months, I have felt the pull to get back to my roots and get closer to the software development process, so I recently accepted a position in a more engineering-focused role.

Fortunately, Diego Dagum from *The Architecture Journal* is stepping up to run *MSDN Magazine*, and I'm confident that he will do a fantastic job. I wish both him and you all the best. I have been a huge fan of *MSDN Magazine* since long before I stepped into this role and will continue to be a fan long afterward.

So, in the tradition of my predecessor and friend Stephen Toub, I'll close with the following:

```
public static void UntilNextTime() {
    System.Windows.Forms.Application.Exit();
    System.Environment.Exit(0);
    System.Diagnostics.Process.GetCurrentProcess().Kill();
    System.Environment.FailFast("Thanks for the memories!");
}
```

[illegible]

Correction: In the October 2009 issue, John Papa's Data Points column had a disclaimer that the content was based on pre-release versions of the technology when it was actually based on RTM.

Visit us at msdn.microsoft.com/magazine. Questions, comments or suggestions for *MSDN Magazine*? Send them to the editor: mmeditor@microsoft.com.

Database Documentation, API for Pre- and Post-Conditions, Blogs and More

One-Click Database Documentation

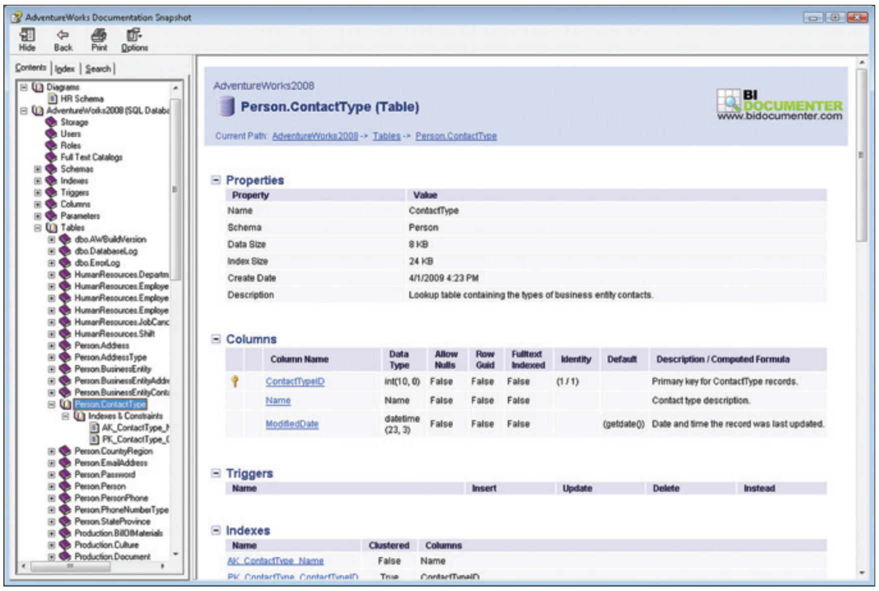
Over the course of my career I have seen a variety of techniques used for documenting the structure and purpose of a database. Most developers use Microsoft SQL Server's Database Diagram tool to generate a pictorial representation, and then call it a day. While this is a good first step, it is rarely sufficient. For starters, sharing database diagrams with other stakeholders can be difficult at best. (I had one client e-mail me dozens of screenshots that, stitched together, comprised one large database diagram.) And such diagrams are often less than ideal for large databases; a diagram displaying hundreds of tables is difficult to read and understand.

While a database diagram is certainly useful, there are much better tools for documenting your database. One such tool is **BI Documenter** (version 3.0), a user-friendly application for auto-generating documentation for Microsoft SQL Server databases. To get started, launch BI Documenter and create a Solution file. The Solution file specifies the databases to document and maintains a history of snapshots. A snapshot is a description of the database's structure at a particular point in time and is generated for you by BI Documenter. Over the lifespan of a project, a database's schema usually changes, sometimes significantly—new tables get added, existing columns get modified or removed, indexes may be added or dropped. These snapshots ensure that the documentation you create today can be faithfully reproduced in the future. When generating the documentation, you can either create a new snapshot of the current database schema or you can regenerate the documentation from a previous snapshot.

After selecting the database to document and creating the snapshot, all that remains is to configure the documentation options. BI Documenter can create a Compiled HTML Help file (.chm) or a series of HTML pages. There are settings to customize the colors used in the documentation as well as the ability to add a logo to each page. You can optionally select what database object types to document and which ones to omit. What's more, BI Documenter includes a built-in database diagramming tool you can use to create and add database diagrams to the documentation. Plus, you can import your own Microsoft Word and image files.

The generated documentation includes a detailed list of the specified database objects, which includes users, roles, indexes, triggers, tables, and more. Viewing information about a particular table lists that table's columns, triggers, indexes, constraints, dependencies, extended properties, and the SQL statements needed to create the table. The table's columns, triggers, and other information are displayed as links that, when clicked, load a page with further details.

BI Documenter is available in three editions: Database, Enterprise, and Workgroup. The Database Edition can only document Microsoft SQL Server databases, while the Enterprise and Workgroup Editions can also document Analysis Services databases, Integration Services packages, and



AdventureWorks2008

Person.ContactType (Table)

Current Path: AdventureWorks2008 -> Tables -> Person.ContactType

Properties

Property	Value
Name	ContactType
Schema	Person
Data Size	8 KB
Index Size	24 KB
Create Date	4/1/2009 4:23 PM
Description	Lookup table containing the types of business entity contacts.

Columns

		Column Name	Data Type	Allow Nulls	Row Guid	Fulltext Indexed	Identity	Default	Description / Computed Formula
🔑		ContactTypeID	int(10, 0)	False	False	False	(1 / 1)		Primary key for ContactType records.
		Name	Name	False	False	False			Contact type description.
		ModifiedDate	datetime (23, 3)	False	False	False		(getdate())	Date and time the record was last updated.

Triggers

Name	Insert	Update	Delete	Instead

Indexes

Name	Clustered	Columns
AK_ContactType_Name	False	Name
PK_ContactType_ContactTypeID	True	ContactTypeID

BI Documenter

All prices confirmed at press time are subject to change. The opinions expressed in this column are solely those of the author and do not necessarily reflect the opinions at Microsoft.

Send your questions and comments for Scott to toolsmm@microsoft.com.

Reporting Services servers. The Database and Professional Editions enable a single user to create the documentation, while the Workgroup Edition allows multiple users to collaborate.

Price: $195 for the Database Edition, $395 for the Enterprise Edition, $495 for the Workgroup Edition

bidocumenter.com

Blogs of Note

I recently worked on a project that involved adding new functionality to an existing line-of-business Windows Presentation Foundation (WPF) application. During my time on that project I found Beth Massi's blog to be an indispensible resource. Beth is a Program Manager for the Visual Studio Community Team at Microsoft. She has created numerous tips, tricks, videos, articles and tutorials on her blog and on Microsoft's Channel 9 Web site (channel9.msdn.com) that explore topics like LINQ, Entity Framework, WPF, ADO.NET Data Services, and Microsoft Office development.

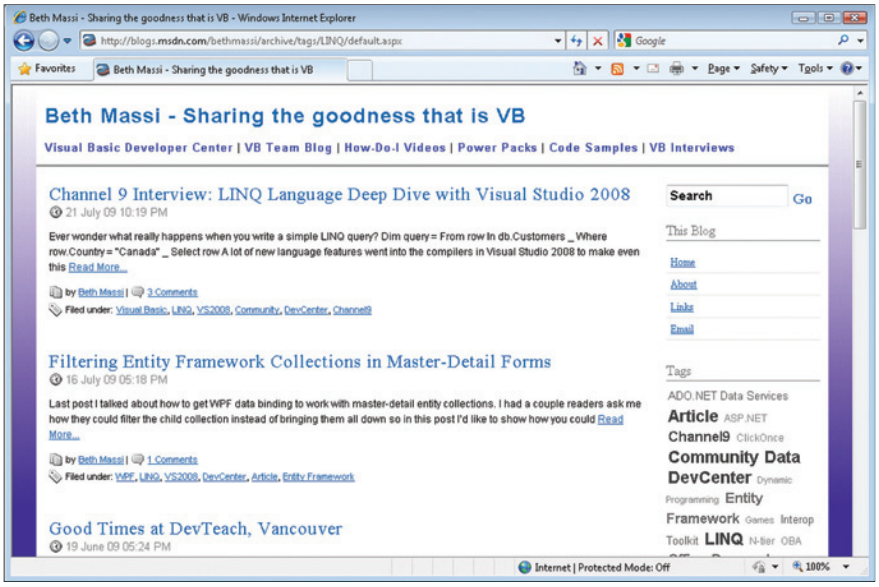
Most of Beth's blog entries describe a specific programming scenario and then show how to solve it with lucid step-by-step instructions, numerous screen shots and code snippets, and links to blog posts and articles with more information. For example, the blog entry titled "Master-Detail Data Binding in WPF with Entity Framework" starts by walking the reader through creating the Entity Data Model. Beth then shows different ways to use LINQ to pull back the appropriate detail records. Other blog entries written in a similar style include "Using TableAdapters To Insert Related Data Into An MS Access Database" and "Tally Rows In A DataSet That Match A Condition," among many others.

What makes Beth's blog stand out is her passion for Visual Basic. All of Beth's code examples are in VB and she frequently posts about upcoming language features, such as support for collection initializers in Visual Basic 10. She's also interviewed several Microsoft MVPs who are using VB, asking them about the applications they're building, their favorite language features, and so on. These interviews (and others like them) can be seen at Channel 9 (channel9.msdn.com/tags/MVP) and at the "I'm a VB" Web site, imavb.net.

blogs.msdn.com/bethmassi

A Fluent API for Pre- and Post-Conditions

When a method is called, it expects its environment to be in a certain state prior to its execution; it may also assume certain conditions hold once execution completes. These assumptions are called pre-conditions and post-conditions. Pre-conditions commonly apply to a method's input parameters. For


Beth Massi - Sharing the goodness that is VB - Windows Internet Explorer
http://blogs.msdn.com/bethmassi/archive/tags/LINQ/default.aspx
Google
Favorites
Beth Massi - Sharing the goodness that is VB
Page
Safety
Tools
Search
Go
This Blog
Home
About
Links
Email
Tags
ADO.NET Data Services Article ASP.NET Channel9 ClickOnce Community Data DevCenter Dynamic Programming Entity Framework Games Interop Toolkit LINQ N-tier OBA
Internet | Protected Mode: Off
100%

Beth Massi - Sharing the goodness that is VB

Visual Basic Developer Center | VB Team Blog | How-Do-I Videos | Power Packs | Code Samples | VB Interviews

Channel 9 Interview: LINQ Language Deep Dive with Visual Studio 2008
21 July 09 10:19 PM

Ever wonder what really happens when you write a simple LINQ query? Dim query = From row In db.Customers _ Where row.Country = "Canada" _ Select row A lot of new language features went into the compilers in Visual Studio 2008 to make even this Read More...

by Beth Massi | 3 Comments
Filed under: Visual Basic, LINQ, VS2008, Community, DevCenter, Channel9

Filtering Entity Framework Collections in Master-Detail Forms
16 July 09 05:18 PM

Last post I talked about how to get WPF data binding to work with master-detail entity collections. I had a couple readers ask me how they could filter the child collection instead of bringing them all down so in this post I'd like to show how you could Read More...

by Beth Massi | 1 Comments
Filed under: WPF, LINQ, VS2008, DevCenter, Article, Entity Framework

Good Times at DevTeach, Vancouver
19 June 09 05:24 PM

Beth Massi's blog

Figure 1 **Pre- and Post-Conditions Implemented Using Conditionals**

```
public string ReadAllText(string path)
{
    // Pre-conditions...
    if (path == null)
        throw new ArgumentNullException(...);
    if (path.Length == 0)
        throw new ArgumentException(...);
    if (IsOnlyWhitespace(path) ||
        ContainsInvalidCharacters(path))
        throw new ArgumentException(...);
    if (path.Length >
        GetMaximumFilePathLength())
        throw new PathTooLongException(...);

    // Open file and read and return contents
       as string...
    object contents = GetFileContents(path);

    // Post-conditions
    if (!contents is string)
        throw
            new InvalidFileTypeException(...);
    if (string.IsNullOrEmpty(
        (string) contents))
        throw new EmptyFileException(...);
}
```

example, in the .NET Framework the File class's ReadAllText method accepts the path of a file as input and returns the contents of the file as a string. The inputted file path cannot be a zero-length string, contain only white space characters, or include any invalid file-path characters; it cannot be null; and its length cannot exceed the system-defined maximum file-path length. If any of these preconditions are not met, the ReadAllText method throws an exception.

Pre- and post-conditions are typically implemented using a series of conditional statements as shown in **Figure 1**. CuttingEdge.Conditions (version 1.0), an open source project started by Steven van Deursen, provides a fluent interface for specifying pre- and post-conditions. (A fluent interface is an API design style that maximizes readability through the use of descriptive names and method chaining.)

To apply a pre-condition on an input parameter, property, or variable, use the Requires extension method; for post-conditions, use the Ensures extension method. Both methods return a Validator object, which has a host of methods available for applying rule checks, such as IsNotNull, IsNotEmpty, and IsEqualTo, among many others. **Figure 2** shows the same method as **Figure 1**, but uses the CuttingEdge.Conditions API to implement the pre- and post-conditions.

Figure 2 **Pre- and Post-Conditions Implemented Using CuttingEdge.Conditions**

```
public string ReadAllText(string path)
{
  // Pre-conditions...
  path.Requires()
    .IsNotNull()
    .IsNotEmpty()
    .Evaluate(!IsOnlyWhitespace(path)) &&
      !ContainsInvalidCharacters(path),
      "path contains whitespace only or
invalid characters")
    .Evaluate(p => p.Length <=
      GetMaximumFilePathLength());

  // Open file and read and return contents
  // as string...
  object contents = GetFileContents(path);

  // Post-conditions
  contents.Ensures()
   .IsOfType(typeof(string))
   .IsNotNull()
   .IsNotEmpty();
}
```

Each Validator method call—IsNotNull, IsNotEmpty, and so on—throws an exception if the condition is not met. For example, if path is null, the IsNotNull method call will throw an ArgumentNullException. And you can optionally provide a string to use as the exception's message.

Sometimes a pre- or post-condition cannot be expressed using one of the Validator class's built-in methods. In such cases you have two options. You can either create an extension method on the Validator class or you can use the Evaluate method, which lets you specify a Boolean or lambda expression to evaluate. If the expression returns true, processing continues; if it returns false, an exception is thrown. **Figure 2** illustrates using both forms of the Evaluate method in the pre-conditions section.

Price: Free, open source
conditions.codeplex.com

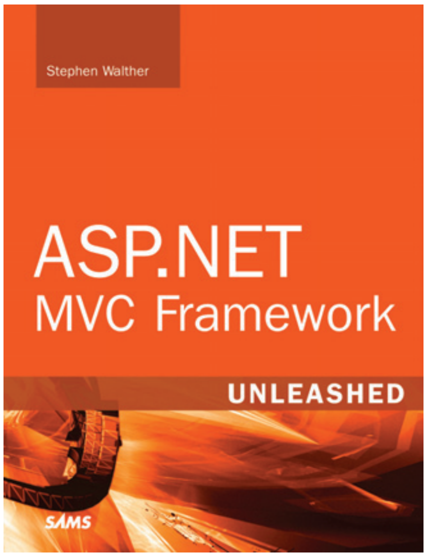
The Bookshelf

ASP.NET MVC is a relatively new framework added to the ASP.NET stack that enables developers to create Web applications using a Model-View-Controller (MVC) pattern. It differs from the traditional Web Forms development model in many ways. For starters, ASP.NET MVC affords much more control over the rendered markup and provides a more distinct separation of concerns. Web pages are accessed using terse, SEO-friendly URLs. And the MVC architecture allows for better testability. For a more in-depth look at the differences between ASP.NET MVC and Web Forms, refer to Dino Esposito's article in the July 2009 issue (msdn.microsoft.com/magazine/dd942833.aspx).

Getting started with ASP.NET MVC involves a bit of a learning curve, even for experienced ASP.NET developers, because of the numerous differences between the two frameworks. For example, when creating an ASP.NET MVC application in Visual Studio, you are prompted to create a unit test project. With ASP.NET MVC, you design your Web pages using HTML along with a few helper classes—there are no Web controls to drag and drop onto the page. And unlike Web Forms, there are no baked-in postback or Web control event models. In short, there are a lot of new techniques to learn when moving from Web Forms to ASP.NET MVC.

If you are an intermediate to experienced ASP.NET developer who wants to learn ASP.NET MVC, check out Stephen Walther's latest book, "ASP.NET MVC Framework Unleashed" (Sams, 2009). Walther does an excellent job introducing new concepts and showing how they're used—without overwhelming the reader with an avalanche of information.

The book starts with a short overview of ASP.NET MVC—the motivation behind the new framework, its design goals,



ASP.NET MVC Framework Unleashed

and benefits. The next 500 pages walk the reader through the key aspects of ASP.NET MVC, one at a time. First, the reader learns how to create a new ASP.NET MVC application in Visual Studio, how to add a database, how models, views and controllers are added to the project, and so forth. The next 100 pages explore models, views and controllers in detail. Each concept is clearly described and is accompanied by screenshots and code samples in both C# and Visual Basic. Later chapters look at the HTML helpers available; explore techniques for validating form data; and dissect ASP.NET MVC's URL routing feature. There are also chapters on authentication, AJAX, jQuery and deployment.

There are a lot of new techniques to learn when moving from Web Forms to ASP.NET MVC.

After examining the core pieces of ASP.NET MVC, in separate chapters, in detail, and with the aid of several simple exercises, the book finishes with an end-to-end example of building a real-world ASP.NET MVC Web application. This final project, spread over 150 pages, cements the concepts explored in the earlier chapters and highlights many of the benefits of the ASP.NET MVC framework.

Price: $49.99
samspublishing.com

SCOTT MITCHELL, *author of numerous books and founder of 4GuysFromRolla.com, is an MVP who has been working with Microsoft Web technologies since 1998. Mitchell is an independent consultant, trainer and writer. Reach him at Mitchell@4guysfromrolla.com or via his blog at ScottOnWriting.net.*

ANDREW DAI

Exploring the .NET Framework 4 Security Model

The .NET Framework 4 introduces many updates to the .NET security model that make it much easier to host, secure and provide services to partially trusted code. We've overhauled the complicated Code Access Security (CAS) policy system, which was powerful but difficult to use and even more difficult to get right. We've also improved upon the Security Transparency model, bringing much of the Silverlight improvements (which I talked about last October: msdn.microsoft.com/magazine/cc765416.aspx) in security enforcement to the desktop framework. Finally, we've introduced some new features that give host and library developers more flexibility over where their services are exposed. With these changes, the .NET Framework 4 boasts a simpler, improved security model that makes it easy for hosts and libraries to sandbox code and libraries to safely expose services.

A Background in .NET Framework Security

Before diving into any particular feature, it helps to have a little background on how security in the .NET Framework works. Partially trusted code is restricted by the permissions it has, and different APIs will require different permissions to successfully be called. The goal of CAS is to make sure that untrusted code runs with appropriate permissions and can't do anything beyond its permissions without authorization.

We can think of the .NET security model as comprising three parts. Specific improvements have been made in each area, and the rest of this article is organized according to these three fundamental sections:

- Policy—Security policy determines which permissions to give a particular untrusted assembly or application. Managed code has Evidence objects associated with it, which can describe where the code is loaded from, who published it and so on. Evidence can be used to determine which permissions are appropriate; the result is called a permission grant set. The .NET Framework has traditionally used CAS policy as a machine-wide mechanism to govern this. As mentioned before, CAS policy has been overhauled in favor of giving hosts more flexibility over their own security policy and unhosted code parity with native code.
- Sandboxing—Sandboxing is the actual process of restricting assemblies or applications to a given permission grant set. The preferred way to sandbox is to create a sandboxing application domain containing a permission grant set for loaded assemblies and an exemption list for specific library assemblies (these are given full trust). With the obsoletion of CAS policy, partial trust code always gets sandboxed this way in .NET Framework 4.
- Enforcement—Enforcement refers to the mechanism that keeps untrusted code restricted to its sandbox. Proper use of enforcement APIs prevents one untrusted assembly from simply calling an API in a different, more-trusted assembly and exercising greater permissions that way. It also allows host and library developers to expose controlled, limited access to elevated behavior and provide meaningful services to partially trusted code. The Level 2 Security Transparency model makes it much easier to safely do this.

Send your questions and comments to clrinout@microsoft.com.

The .NET security model has always been of particular importance to host and library developers (who often go hand in hand). Examples of the .NET security model are ASP.NET and SQL CLR, which both host managed code within controlled environments and restricted contexts. When a host like these wants to load a partially trusted assembly, it creates a sandboxed application domain with the appropriate permission grant set. The assembly is then loaded into this sandboxed domain. The host also provides library APIs that are fully trusted but callable from the hosted code. The libraries are also loaded into the sandboxed domain, but are explicitly placed on the exemption list mentioned earlier. They rely on the .NET Framework's enforcement mechanisms to ensure that access to its elevated abilities is tightly controlled.

For most managed application developers, this is all magic that is happening at the framework level—even developers writing code that will be run in a sandbox don't need to know all the details of how the security model works. The framework ensures sandboxed code is limited to using APIs and abilities that the host provides. The .NET security model and CAS have long been the realm of enterprise administrators and host and library developers; for them, we've made things easier than ever.

Policy

CAS policy has been provided since the beginning of the .NET Framework to give machine and enterprise administrators a way to fine-tune what the runtime considered trusted or untrusted. While

CAS policy was very powerful and allowed for very granular controls, it was extremely difficult to get right and could hinder more than help. Machine administrators could lock certain applications out of needed permissions (described in the next major section, Sandboxing), and many people wondered why their applications suddenly stopped working once they decided to put them on a network share. Furthermore, CAS policy settings didn't move forward from one version of the runtime to another, so the elaborate custom CAS policy that someone set up in .NET Framework 1.1 had to be redone by hand for .NET Framework 2.0.

Security policy could be split into two scenarios: security policy for hosted code and security policy for the machine or enterprise. Regarding machine policy, the Common Language Runtime security team has decided that the runtime was the wrong place to govern it, as native code was obviously not subject to its restrictions. While it makes sense for hosts to be able to determine what their hosted code can do, unhosted exes that are simply clicked or run from the command line should behave like their native counterparts (especially since they look identical to the users running them).

The correct place for global security policy is at the operating system level, where such a policy would apply to native and managed code equally. Therefore, we're encouraging machine administrators to look at solutions like Windows Software Restriction Policies and disabling machine-wide CAS policy resolution by default. The other scenario, security policy for hosted code, is still very much valid in the managed code world. Host security policy is now easier to govern, as it will no longer clash with an arbitrary machine policy.

What This Means for You

For one, all unhosted managed code runs as fully trusted by default. If you run an .EXE from your hard drive or a network share, your app will have all the abilities a native app running from the same place would have. Hosted code, however, is still subject to the security decisions of the host.(Note that all the ways that code can arrive via the Internet are hosted scenarios—ClickOnce applications, for example—so this does not mean that code running over the Internet is fully trusted.)

For many applications, these changes are mostly in the background and will have no perceived effect. Those that are affected by the change may run into two issues. The first is that certain CAS policy-related APIs are deprecated, many having to do with assembly loads (so read on if you do this at all). Second, and affecting fewer people (primarily hosts), will be the fact that heterogeneous application domains (which are described in the Sandboxing section) aren't available by default.

But I'm not doing any of this! How do I just make it work?

Perhaps you've run into an error or obsoletion message that looked something like this:

```
This method [explicitly/implicitly] uses CAS policy, which has been
obsoleted by the .NET Framework. In order to enable CAS policy for
compatibility reasons, please use the NetFx40_LegacySecurityPolicy
configuration switch. Please see [link to MSDN documentation]for more
information.
```

For compatibility reasons, we've provided a configuration switch that allows a process to enable CAS policy resolution on it. You may enable CAS policy by placing the following in your project's app.config file:

```
<configuration>
   <runtime>
      <!-- enables legacy CAS policy for this process -->
      <NetFx40_LegacySecurityPolicyenabled="true" />
   </runtime>
</configuration>
```

The following section describes where to start looking for migration, if the exception is being thrown from your own code. If it isn't, then the configuration switch is the way to go and the following section shouldn't apply directly to you.

Affected APIs

Affected APIs can be divided into two groups: those that are explicitly using CAS policy and those that are implicitly using it. Explicit usages are obvious—they tend to reside in the System.Security.Policy.SecurityManager class and look something like SecurityManager.ResolvePolicy. These APIs directly call or modify the machine's CAS policy settings, and they have all been deprecated.

Implicit usages are less obvious—these tend to be assembly loads or application domain creations that take evidence. CAS policy is resolved on this evidence, and the assembly is loaded with the resulting permission grant set. Since CAS policy is off by default, it doesn't make sense to try to resolve it on this evidence. An example of such an API is Assembly.Load(AssemblyName assemblyRef,Evidence assemblySecurity).

There are a couple of reasons why such an API would be called:

1. Sandboxing—Perhaps you know that calling that Assembly.Load overload with zone evidence from the Internet will result in that assembly being loaded with the Internet named permission set (unless, that is, an administrator changed that evidence mapping for this particular machine or user!).
2. Other parameters on the overload—Maybe you just wanted to get to a specific parameter that existed only on this overload. In this case, you might've simply passed null or Assembly.GetExecutingAssembly().Evidence for the evidence parameter.

If you're trying to sandbox, the Sandboxing section describes how to create a sandboxed application domain restricted to the Internet named permission set.. Your assembly could then be loaded into that domain and be guaranteed to have the permissions you intended (that is, not subject to the whims of an administrator).

In the second scenario, we've added overloads to each of these APIs that expose all necessary parameters but don't expose an evidence parameter. Migration is a simple matter of cutting out the evidence argument to your calls. (Note that passing null Evidence into an obsolete API still works as well, as it doesn't result in CAS policy evaluation.)

One additional thing to note is that if you're doing an assembly load from a remote location (that is, Assembly.LoadFrom("http://...")), you'll initially get a FileLoadException unless the following configuration switch is set. This was done because this call would've sandboxed the assembly in the past. With CAS policy gone, it is fully trusted!

```
<configuration>
   <runtime>
      <!-- WARNING: will load assemblies from remote locations as fully
         trusted! -->
      <loadFromRemoteSources enabled="true" />
   </runtime>
</configuration>
```

Another way to do this, without turning this switch on for the whole process, is to use the new Assembly.UnsafeLoadFrom API, which acts like LoadFrom with the switch set. This is useful if you only want to enable remote loads in certain places or you don't own the primary application.

With machine-wide CAS policy out of the picture, all examination of assembly evidence and decisions regarding appropriate permission sets is left up to hosts of managed code. Without a complicated system on top of it interfering with its security decisions (aside from any OS security policy), a host is free to assign its own permissions. Now it's time to assign those permissions to partial trust assemblies.

Sandboxing

Via a host's security policy, we can determine the correct permission grant set to give to a partial trust assembly. Now we need a simple, effective way to load that assembly into an environment that is restricted to that particular grant set. Sandboxing, particularly using the simple sandboxing CreateDomain overload, does just that.

Sandboxing in the Past

With the old CAS policy model, it was possible to create a heterogeneous application domain, where every assembly in the domain had its own permission set. An assembly load with Internet zone evidence could result in two or more assemblies at different partial trust levels being loaded into the same domain as the full trust assembly doing the loading. Furthermore, the application domain could have its own evidence, giving it its own permission set.

There are several issues with this model:

- The permission set granted to an assembly is dependent on CAS policy, as several policy levels are intersected to compute the final permission set. Therefore, it is possible to end up with fewer permissions than intended.
- Similar to the previous point, evidence evaluation on an assembly is done by CAS policy, which could differ across machines, users and even versions of the runtime (CAS policy settings didn't move forward with new versions of the runtime). Therefore, it wasn't always obvious what permission grant set an assembly was getting.
- Partial trust assemblies are not usually examined for security hardening, making "middle trust" assemblies vulnerable to "lowest trust" assemblies. Assemblies are freely and easily able to call each other, so having many of them with different abilities becomes problematic from a security perspective. Can someone be certain that every combination of calls from assemblies at different trust levels is secure? Is it absolutely safe to cache information from a middle trust layer?

Because of these issues, we introduced the concept of a homogeneous application domain, which contains only two permission grant sets (partial trust and full trust) and is extremely simple to create and reason about. Homogeneous domains, and how to create them, are described later in this section.

Another popular mechanism for sandboxing was the use of PermitOnly and Deny, which are stack walk modifiers that list specific allowed permissions (and nothing more) and disallow specific permissions, respectively. It seemed useful to say, "I only want callers with permissions x and y to be able to call this API," or, "as an API, I want to deny permission to all my callers." However, these modifiers *did not actually change the permission grant set of a particular assembly*, which meant that they could be asserted away because *all they did was intercept demands*. An example of this in action is shown in **Figure 1**.

Figure 1 **A Call Stack Representing an Attempt at Sandboxing with Deny**

Method Called	Assembly's Grant Set	Stack walk op/modifier
Host.Main	FullTrust	
Host.LoadUntrusted	FullTrust	Deny X permission
Untrusted.DoEvil	FullTrust	Assert X permission
File.FormatDisk	FullTrust	Demand X permission

Without the red Assert, the demand hits the Deny and the stack walk is terminated. When the red Assert is active, however, the Deny is never hit, as Untrusted has asserted the demand away. (Notes: Call stack is growing down. APIs do not represent actual APIs in the framework.) For this reason, Deny is deprecated in the .NET Framework 4, because using it is always a security hole (PermitOnly is still around because it can be legitimately used in a few corner cases, but is generally discouraged). Note that it can be reactivated using the NetFx40_LegacySecurityPolicy switch, mentioned in the policy section above.

Sandboxing Today

For the .NET Framework, the unit of isolation we use is the application domain. Each partial trust application domain has a single permission grant set that all assemblies loaded into it get, except for the ones specifically listed on the full trust exemption list or loaded from the Global Assembly Cache. Creating this domain is very simple — the .NET Framework provides a simple sandboxing API that takes in everything you need to create the domain:

```
AppDomain.CreateDomain( string friendlyName,
                        Evidence securityInfo,
                        AppDomainSetup info,
                        PermissionSet grantSet,
                        params StrongName[] fullTrustAssemblies);
```

Where the parameters are:

- friendlyName—The friendly name of the application domain.

- securityInfo—Evidence associated with the application domain. This isn't used for CAS policy resolution, obviously, but can be used to store things like publisher information.
- info—Application domain initialization information. This must include, at minimum, an ApplicationBase, representing the store where partial trust assemblies reside.
- grantSet—The permission grant set of all loaded assemblies in this domain, except for those on the full trust list or in the Global Assembly Cache.
- fullTrustAssemblies—A list of StrongNames of assemblies that are granted full trust (exempt from partial trust).

Once the domain is created, you can call AppDomain.CreateInstanceAndUnwrap on a MarshalByRefObject in your partial trust assembly and then call into its entry point method to kick it off, as shown in **Figure 2**.

That's it! With several lines of code, we now have a sandbox with partial trust code running in it.

This CreateDomain API was actually added in .NET Framework 2.0, so it's not new. However, it's worth mentioning as it's now the only truly supported way to sandbox code. As you can see, the permission set is passed directly, so no evidence has to be evaluated in loading assemblies into this domain; you know exactly what each loaded assembly is going to get. Furthermore, you're using a real isolation boundary to contain partial trust code, which is extremely helpful in making security assumptions. With the simple sandboxing CreateDomain API, sandboxes become more obvious, consistent and secure—all things that help make dealing with untrusted code easier.

Enforcement

At this point, we have an appropriate permission grant set for our partial trust assembly and have loaded the assembly into a proper sandbox. Great! However, what if we actually want to expose some elevated functionality to partially trusted code? For example, I may not want to give full file system access to an Internet application, but I don't mind if it reads and write from a known temporary folder.

Figure 2 Sandbox with Partial Trust Code Running Inside

```
PermissionSetpermset = newPermissionSet(PermissionState.None);
ps.AddPermission(newSecurityPermission(
   SecurityPermissionFlag.Execution));
AppDomainSetup ptInfo = new AppDomainSetup();
ptInfo.ApplicationBase = ptAssemblyStore;


AppDomainsandboxedDomain = AppDomain.CreateDomain(
   "Sandbox",
   AppDomain.CurrentDomain.Evidence,
   ptInfo,
   permset);

ExampleTypeet = sandboxedDomain.CreateInstanceAndUnwrap(
   typeof(ExampleType).Assembly.FullName,
   typeof(ExampleType).FullName)
   as PartialTrustType;

et.EntryPoint();
```

Those of you who read last year's column on Silverlight security (msdn.microsoft.com/magazine/cc765416.aspx) know exactly how this issue is addressed in that platform—through the Security Transparency model that neatly divides code into three buckets. I'm happy to say that Silverlight's advancement of the model is now in effect on .NET Framework 4. This means that the benefits of the simpler model enjoyed by the Silverlight platform libraries are now available to non-Microsoft developers of partial trust libraries. Before I go into that and other improvements in the enforcement space, though, I'll discuss our primary enforcement mechanisms from before.

Enforcement in the Past

I mentioned last year that Security Transparency was actually introduced in .NET Framework 2.0, but served primarily as an audit mechanism rather than an enforcement one (the new Security Transparency model is both). In the older model, or Level 1 Security Transparency, violations did not manifest themselves as hard failures—many of them (like p/invoking into native code) resulted in permission demands. If your transparent assembly happened to have UnmanagedCode in its grant set, it could still go ahead and do what it was doing (violating the Transparency rules in the process). Furthermore, Transparency checks stopped at the assembly boundary, further reducing its enforcement effectiveness.

True enforcement in the .NET Framework 2.0 came in the form of LinkDemands— JIT time checks that checked if the grant set of the calling assembly contained the specified permission. That was all well and good, but this model essentially required library developers to use two different mechanisms for audit and enforcement, which is redundant. The Silverlight model, which consolidated and simplified these two concepts, was a natural progression from this state and became what is now Level 2 Security Transparency.

Level 2 Security Transparency

Level 2 Security Transparency is an enforcement mechanism that separates code that is safe to execute in low-trust environments and code that isn't. In a nutshell, it draws a barrier between code that can do security-sensitive things (Critical), like file operations, and code that can't (Transparent).

The Security Transparency model separates code into three buckets: Transparent, Safe Critical and Critical.. The following diagram, **Figure 3**, describes these buckets. (Note: Green arrows represent calls that are allowed; red arrows represent those that aren't. Self-loops are valid as well, but not shown.)

For typical desktop applications, the Level 2 Transparency model has no noticeable effect—code that does not have any security annotations and is not sandboxed is assumed to be Critical, so it is unrestricted. However, since it is Critical, it is off-limits to partial trust callers. Therefore, developers who don't have partial trust scenarios won't have to worry about exposing anything to partial trust.

For sandboxed applications, the opposite is true—any assembly loaded into a sandboxed application domain is assumed to be completely Transparent (even if it has annotations specifying otherwise). This ensures that partial trust code cannot attempt to elevate via asserting for permissions or calling into native code (a Full Trust equivalent action).

Libraries exposed to partial trust callers, unlike desktop or sandboxed apps, must be keenly aware of their security requirements and have much more flexibility over their abilities and what they expose. A typical partial trust-callable library should be primarily Transparent and Critical code with a minimal set of Safe Critical APIs. Critical code, while unrestricted, is known to be inaccessible from partial trust code. Transparent code is callable from partial trust code but is safe. Safe Critical code is extremely dangerous, as it provides elevated functionality, and utmost care must be taken to make sure its caller is validated before transitioning over to Critical code.

The Security Transparency attributes and their behaviors are listed and described in **Figure 4**. Keep in mind that the highest-scoped attribute applies for all introduced APIs under it, regardless of whether those APIs have their own annotations. AllowPartiallyTrustedCallers is different in that it defers to and honors lower-level attributes. (Note: This table describes the attributes and their behavior when applied at the assembly, type or member level. The attributes apply only to introduced APIs, which means subclasses and overrides are subject to the inheritance rules and may be at different Transparency levels.)

Those of you who remember last October's article will probably notice that the attributes work, more or less, the same way they do in Silverlight. You might also remember that there were specific inheritance rules associated with the different types of code. Those are also in effect in the desktop. For more details on the inheritance rules and other aspects of Level 2 Transparency, take a look at last year's article, "Security in Silverlight 2" (msdn.microsoft.com/magazine/cc765416.aspx).

Conditional AllowPartiallyTrustedCallers

The AllowPartiallyTrustedCallers attribute (APTCA) indicates that an assembly is a library that may expose security-sensitive functionality to partial trust. APTCA library assemblies are often written in conjunction with hosts, since the hosts typically want to expose specific functionality to their hosting environments. One major example is ASP.NET, which exposes the System.Web namespace to its hosted code, which may be at various trust levels.

However, putting APTCA on an assembly means it's available to partial trust in any host that decides to load it, which can be a liability if the assembly author doesn't know how that assembly will behave in different hosts. Therefore, host developers sometimes want their libraries to be available to partial trust only when loaded in their own domains. ASP.NET does exactly this, and in earlier versions has had to use LinkDemands for special permissions on their APIs. While this works, it causes everyone building on top of them to have to satisfy that LinkDemand, preventing those up-stack assemblies from being transparent.

To solve this, we introduced the Conditional APTCA feature, which allows libraries to expose APIs to partial trust callers only in an enabling host (via a list).

The specific roles of the host and library are:

- The library simply qualifies the AllowPartiallyTrustedCallers attribute with a parameter, the PartialTrustVisibilityLevel enum. For example:

```
[assembly: AllowPartiallyTrustedCallers(PartialT
rustVisibilityLevel=
PartialTrustVisibilityLevel.NotVisibleByDefault)]
```

  This attribute basically says that the library is not callable from partial trust unless the host has it on its allow-list, mentioned below. A value of VisibleToAllHosts would make the library callable from partial trust in all hosts.
- The host specifies partial trust visible assemblies, per application domain, via an allow list. This list is typically populated via a configuration file supplied to the host. An important thing to keep in mind is that unconditional APTCA assemblies, like the basic framework libraries, do not have to be added to this list.(Also important to keep in mind is that if you're enabling a Conditional APTCA assembly, you should enable its transitive closure of dependent Conditional APTCA assemblies as well. Otherwise, you might end up with odd behavior, as your original assembly tries to call APIs that it assumes are accessible but really aren't.)

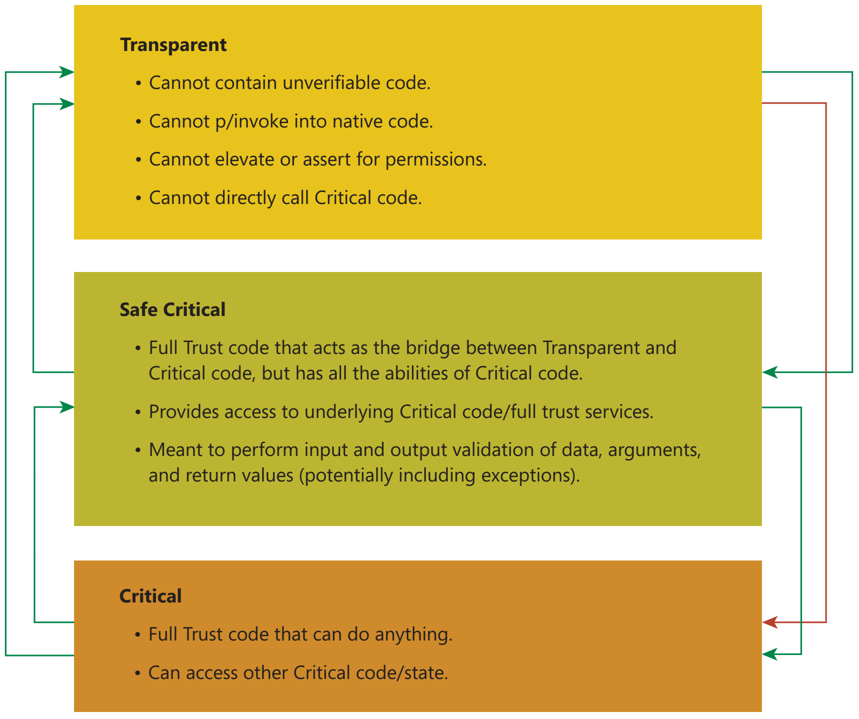

Transparent
• Cannot contain unverifiable code.
• Cannot p/invoke into native code.
• Cannot elevate or assert for permissions.
• Cannot directly call Critical code.
Safe Critical
• Full Trust code that acts as the bridge between Transparent and Critical code, but has all the abilities of Critical code.
• Provides access to underlying Critical code/full trust services.
• Meant to perform input and output validation of data, arguments, and return values (potentially including exceptions).
Critical
• Full Trust code that can do anything.
• Can access other Critical code/state.

Figure 3 **Security Transparency Model**

Easier to Secure

Much has happened in the security model for the .NET Framework 4. CAS policy has been disabled by default, leaving all security policy decisions up to the host and granting unhosted managed exes behavioral parity with native exes. Disabling CAS policy has also disabled heterogeneous application domains, finally making the efficient simple-sandboxing CreateDomain overload the primary supported sandboxing mechanism for partial trust assemblies. Silverlight's improvements to the Security Transparency model, described last October, have also come to the desktop, providing partial trust library developers with the same efficiency and cleanliness benefits that were provided to the Silverlight platform.

We've crafted these changes in such a way that most applications will continue to work as they have before, but the host and library developers out there will find a simpler model to work with—one that is more deterministic, simpler to use and, therefore, easier to secure. ■

ANDREW DAI *is a program manager on the CLR security team. For more, in-depth information on how to use the features mentioned in this article, please visit the CLR team blog (blogs.msdn.com/clrteam) and Shawn Farkas' .NET security blog (blogs.msdn.com/shawnfa).*

THANKS to the following technical experts for reviewing this article: *Cristian Eigel, Shawn Farkas, Joshua Goodman, Mike Rousos and Mueez Siddiqui.*

Figure 4 **Security Transparency Attributes and Their Behaviors**

	Assembly Level	Type Level	Member Level
SecurityTransparent	All introduced types and members are Transparent.	N/A	N/A
SecuritySafeCritical	N/A	Type and all introduced members are Safe Critical.	Member is Safe Critical.
SecurityCritical	All introduced types and members are Critical.	Type and all introduced members are Critical.	Member is Critical.
AllowPartiallyTrustedCallers	All code is Transparent unless annotated otherwise.	N/A	N/A

Conditional Rendering in ASP.NET AJAX 4.0

Client-side rendering is by far the most exciting, and long-awaited, new feature you'll find in ASP.NET AJAX 4. Client-side rendering allows you to use HTML templates to define your desired layout and supplies a text-based syntax for placeholders of run-time data. It all looks like server-side data binding, except that it takes place within the client browser with external data downloaded via Web services.

Last month, I covered the basics of the new DataView client control and the binding techniques that will be most commonly used. In this article, I'll go one step further and cover conditional template rendering.

Conditional Template Rendering

In ASP.NET AJAX 4, an HTML template for data binding is a piece of markup that may contain ASP.NET markup, HTML literals and some placeholders for run-time data. The rendering algorithm is fairly simple: bound to such templates, the DataView control fetches some data and uses that to fill up the template. The resulting markup, with actual data substituted for placeholders, is then displayed in lieu of the original HTML template.

There are a couple of ways in which you can create an instance of the DataView control: declaratively or programmatically. However, the algorithm used to generate the markup remains the same. This is where we left the matter in last month's article.

Going forward, a question springs up naturally. What if some logic is required to render the template? What if you need conditional rendering that produces a different markup based on different run-time conditions? Some client-side code must be intertwined with markup to check the values of data items being bound as well as the state of other global JavaScript objects.

ASP.NET AJAX 4 defines a special set of namespaced attributes through which you attach custom behaviors to the HTML template. These behaviors are JavaScript expressions to be evaluated and executed at very specific stages of the rendering process. **Figure 1** lists the predefined code attributes for conditional template rendering.

Code attributes are recognized by the template builder and their content is appropriately used in the rendering process. Attributes in **Figure 1** can be attached to any DOM elements used in an ASP.NET AJAX template.

Note that attributes in **Figure 1** were scoped in the code: namespace up until Preview 4 of the ASP.NET AJAX 4 library and Visual Studio 2010 beta 1. Starting with Preview 5, the ASP.NET team eliminated the code: namespace, and it is using only the sys: namespace for everything.

This column is based on a pre-release version of ASP.NET 4.0.

Send your questions and comments for Dino to cutting@microsoft.com.

Conditional Rendering in Action

The sys:if attribute is assigned a Boolean expression. If the expression returns true, then the element is rendered; otherwise, the algorithm proceeds with the next step. Here's a trivial example, just to quickly illustrate the point:

```
<div sys:if="false">
:
</div>
```

While processing this markup, the builder just ignores the DIV tag and all of its content. In a certain way, the sys:if attribute can be used to comment out parts of the template at development time. All in all, assigning a constant value of false to the sys:if attribute is not much different from doing the following in C#:

```
if (false)
{
   :
}
```

Setting sys:if to a false value, doesn't exactly hide a HTML element. It should be noted that any ASP.NET AJAX template is initially treated like plain HTML by the browser. This means that any

Figure 1 **Attributes for Conditional Rendering of DOM Elements Within a Template**

Attribute	Description
sys:codeif	The attribute evaluates to a Boolean expression that controls the rendering of the HTML element. If the expression returns false, the element is not rendered.
sys:codebefore	The attribute is expected to contain an arbitrary piece of JavaScript code that executes before the HTML element is rendered.
sys:codeafter	The attribute is expected to contain an arbitrary piece of JavaScript code to be executed right after the rendering of the HTML element.

Figure 2 **Pseudo-Variables Supported by the ASP.NET AJAX Template Builder**

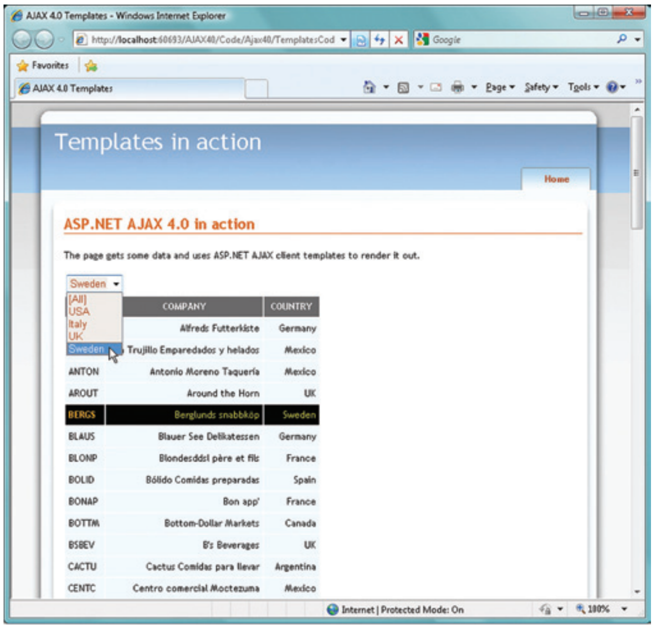
Variable	Description
$component	Added in Preview 5, it returns the last component that was created via sys:attach.
$context	Added in Preview 5, it returns is an instance of Sys.UI.TemplateContext, so you have all the information available to you from that (e.g., $context.parentContext, $context.dataItem, $context.data, others). Another noteworthy addition is $context.data, which is equal to the entire dataset that the dataItem is from. For example, you can obtain $dataItem as follows: $dataItem == $context.data[$index].
$dataItem	Returns the current data object being currently bound to the template. For example, if the template is bound to an array of MyCustomer objects, $dataItem returns the MyCustomer object being bound in the current iteration.
$element	Returns the DOM element that is being rendered in the template. If used in, say, a sys:codebefore attribute attached to a TD element, $element references the current TD object.
$id("yourID")	Returns a unique ID based on the specified string. You use this variable in a template when you need to assign a unique ID to a given element within each instance of the template. The ID is typically obtained by concatenating the $index value to the specified string. However, the primary goal of $id is just isolating you from the details of the ID function being used.
$index	Returns the 0-based index of the current iteration.

template is fully processed to a document object model (DOM) tree. However, as an ASP.NET AJAX template is decorated with the sys-template attribute, nothing of the DOM tree shows up when the page is displayed. In fact, the sys-template attribute is a CSS class that contains the following:

```
.sys-template { display:none; visibility:hidden; }
```

The sys:if attribute keeps the HTML element off the actual markup for the template. The sys:if attribute is ignored if attached to a HTML element outside any ASP.NET AJAX templates. The sys:if attribute is not currently associated with an else branch.

If defined, the sys:codebefore and sys:codeafter attributes execute before and after the rendering of the HTML element. The two attributes can be used together or individually as it best suits you. The content of the attributes must be a piece of executable JavaScript code.


AJAX 4.0 Templates - Windows Internet Explorer
http://localhost:60693/AJAX40/Code/Ajax40/TemplatesCod
Google
Favorites
AJAX 4.0 Templates
Page
Safety
Tools
Home
Internet | Protected Mode: On
100%

Templates in action

ASP.NET AJAX 4.0 in action

The page gets some data and uses ASP.NET AJAX client templates to render it out.

Sweden
[All]
USA
Italy
UK
Sweden

	COMPANY	COUNTRY
	Alfreds Futterkiste	Germany
	Trujillo Emparedados y helados	Mexico
ANTON	Antonio Moreno Taquería	Mexico
AROUT	Around the Horn	UK
BERGS	Berglunds snabbköp	Sweden
BLAUS	Blauer See Delikatessen	Germany
BLONP	Blondesddsl père et fils	France
BOLID	Bólido Comidas preparadas	Spain
BONAP	Bon app'	France
BOTTM	Bottom-Dollar Markets	Canada
BSBEV	B's Beverages	UK
CACTU	Cactus Comidas para llevar	Argentina
CENTC	Centro comercial Moctezuma	Mexico

Figure 3 **A Sample ASP.NET AJAX Page with Conditional Template Rendering**

Altogether, the code attributes give you enough power to deal with nearly every possible scenario, even though not always with a straightforward solution. Let's consider a less trivial example of sys:if and sys:codebefore.

By the way, you may have noticed a few weird $-prefixed variables in the preceding code. Let me briefly introduce them just before introducing the example.

Template Pseudo-Variables

In the custom code you use within the code attributes of a template, you have access to the full set of public properties of the data item. In addition, you can access some extra variables defined and exposed by the template builder for your convenience.

Currently, the documentation refers to them as "pseudo-columns" but I personally like the term "pseudo-variable." **Figure 2** lists them all.

These pseudo-variables provide a glimpse of the internal state of the rendering engine as it is working. You can use any of such variables as you would use any JavaScript variable in the ASP.NET AJAX template.

Coloring Matching Rows in a Table

As an example, let's consider a page that shows a list of customers plus a drop-down list to pick up a country. Whenever a new country is selected, the list of customers refreshes to render customers from that country in a different style (see **Figure 3**).

Figure 4 shows the markup for the sample page. As you can see, the page is a content page associated with a master page. The required Sys namespace is declared in the Body tag defined in the master page.

You should note that Preview 5 of ASP.NET AJAX requires you to override the MicrosoftAjax.js file that comes with the ScriptManager control and beta 1. This is a temporary fix that will no longer be necessary as assemblies are updated to beta 2 and then to release to manufacturing.

Before coming to grips with ASP.NET AJAX templates, let me focus on the markup code for the drop-down list control.

Figure 4 **Code Attributes in Action**

```
<asp:Content ContentPlaceHolderID="PH_Body" runat="server">

   <asp:ScriptManagerProxy runat="server" ID="ScriptManagerProxy1">
      <Scripts>
        <asp:ScriptReference Name="MicrosoftAjax.js"
                             Path="~/MicrosoftAjax.js" />
        <asp:ScriptReference Path="~/MicrosoftAjaxTemplates.js" />
      </Scripts>
   </asp:ScriptManagerProxy>

    <div>
        <asp:DropDownList ID="listOfCountries" runat="server"
            ClientIDMode="Static"
            onchange="listOfCountries_onchange()">
        </asp:DropDownList>


        <table id="gridLayout">
        <tr>
           <th>ID</th>
           <th>COMPANY</th>
           <th>COUNTRY</th>
        </tr>
        <tbody id="grid" class="sys-template">

        <tr sys:if="$dataItem.Country != currentCountry">
            <td align="left">{{ ID }}</td>
            <td align="right">{{ CompanyName }}</td>
            <td align="right">{{ Country }}</td>
        </tr>
        <tr sys:if="$dataItem.Country == currentCountry"
            class="highlight">
            <td align="left"
                sys:codebefore="if($dataItem.Country == 'USA') {
                                   $element.style.color = 'orange';
                                   $element.style.fontWeight=700;
                                }">
                {{ ID }}
            </td>
            <td align="right">{{ CompanyName }}</td>
            <td align="right">{{ Country }}</td>
        </tr>
        </tbody>
    </table>

    </div>
</asp:Content>
```

Setting up the Drop-Down List

The code for the drop-down list for countries is as follows:

```
<asp:DropDownList ID="listOfCountries" runat="server"
     ClientIDMode="Static"
     onchange="listOfCountries_onchange()">
</asp:DropDownList>
```

As you can see, the control assigns a value to the new ClientIDMode property and provides a client-side handler for the DOM-level onchange event. The control is bound to its data on the server, precisely in the classic Page_Load method:

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      // Fetch data
      string[] countries = new string [] {"[All]", "USA", ... "};

      // Serialize data to a string
      string countriesAsString = "'[All]', 'USA', ...'";

      // Emit the array in the page
      this.ClientScript.RegisterArrayDeclaration(
          "theCountries", countriesAsString);

      // Bind data to the drop-down list
      listOfCountries.DataSource = countries;
      listOfCountries.DataBind();
   }
}
```

The binding procedure is articulated in two steps. First, a JavaScript array is emitted in the response that contains the same data bound programmatically to the DropDownList control. Next, the classic server-side data binding takes place.

This technique is known as Dual-Side Templating and is a variation of the standard client-side data binding pattern. The difference consists in the fact that data to bind is fetched on the server the first time the page is accessed and served to the client as an embedded JavaScript array.

Further client-side data binding that proves necessary can then take place using the embedded array. In this way, you basically save an extra roundtrip to get the data. This variation of the classic client data binding is helpful when you display static data that don't change during the user interaction. In the example, I used this technique only for getting the list of countries; the list of customers, instead, is fetched from the Web server using a Web service.

When you use a server-side control to emit HTML markup, you may have little control over the actual ID if you're using a master page. In ASP.NET AJAX 4, the new ClientIDMode property gives you more flexible ways to deal with the issue.

In particular, if you set ClientIDMode to Static as in the example, then the client ID of the HTML element will match exactly the server ID. This trick is not useful when you're going to repeat that server control in the context of a data-bound templated control.

The following script code handles the change of selection event in the drop-down list:

```
<script language="javascript" type="text/javascript">
    var currentCountry = "";

    function listOfCountries_onchange() {
        // Pick up the currently selected item
        var dd = $get("listOfCountries");
        currentCountry = dd.options[dd.selectedIndex].value;

        // Refresh the template
        refreshTemplate();
    }

    function refreshTemplate() {
        var theDataView = $find("grid");
        theDataView.refresh();
    }
</script>
```

Note that this code will raise a JavaScript error if you don't set the ClientIDMode property of the DropDownList control to Static. This is because of the ID-mangling work that ASP.NET usually does to ensure that when master pages are used each produced HTML element has a unique ID.

The preceding onchange event handler saves the name of the currently selected country to a global JavaScript variable and then refreshes the ASP.NET AJAX template. Let's focus on templates now.

Conditional Templates

The template is created and populated programmatically as below:

```
<script language="javascript" type="text/javascript">
   function pageLoad()
   {
      dv = $create(Sys.UI.DataView,
            {
               autoFetch: true,
               dataProvider: "/ajax40/mydataservice.asmx",
               fetchOperation: "LookupAllCustomers"
            },
            {},
            {},
            $get("grid")
      );
   }
</script>
```

The DataView client control makes a call to the specified Web service, performs the given fetch operation and uses any returned data to fill the ASP.NET AJAX template rooted in the DOM element named "grid."

The overall template is rendered using a DataView instance bound to the return value of the LookupAllCustomers method on the sample Web service. The method returns a collection of Customer objects with properties such as ID, CompanyName and Country.

The template will stay bound to its data for the entire lifetime of the page regardless of the changes that may occur to the data. What if, instead, you just want to modify the rendering of the template—no data refresh whatsoever—as certain run-time conditions change? To get this, you need to insert code attributes in the template.

What you really need here is a truly conditional rendering that renders the template in one way if a given condition is verified and otherwise if the condition is not verified. As mentioned, the sys: if attribute doesn't support "if-then-else" semantics, and all it does is rendering, or ignoring, its parent element based on the value of the Boolean guard.

A possible workaround to simulate the two branches of a condition consists in using two mutually exclusive portions of template, each controlled by a different Boolean expression. Also shown in **Figure 4**, the code follows the schema below:

```
<tr sys:if="$dataItem.Country != currentCountry">
  :
</tr>
<tr sys:if="$dataItem.Country == currentCountry"
    class="highlight">
  :
</tr>
```

The variable currentCountry is a global JavaScript variable that contains the name of the currently selected country. The variable is updated every time the onchange event is raised by the HTML markup for the server-side DropDownList control.

In the preceding code snippet, the former TR element is rendered conditionally based on the value of the Country property of the data item being bound. If the variable matches the selected country, the former TR is skipped. This behavior relies on the fact that the global variable is initialized to the empty string and doesn't subsequently match any value. As a result, the table row template is initially rendered for any returned customer.

As the user makes a selection in the drop-down list, the global currentCountry variable is updated. However, this action doesn't automatically trigger any refresh on the template as you see in **Figure 3**. The refresh of the template must be explicitly commanded in the onchange event handler. Here's a possible way of doing that:

```
var theDataView = $find("grid");
theDataView.refresh();
```

The $find function is shorthand for a lookup function that in the Microsoft AJAX library retrieves instances of components. To use $find (or $get), you must have a ScriptManager control at work and configured in a way that references the MicrosoftAjax.js core library. Once you have retrieved the DataView instance associated with the "grid" template, you just invoke its refresh method. Internally, the method just recompiles the template and updates the DOM. Note that you don't strictly need to retrieve the DataView instance from the list of registered components. You can also save the DataView instance to a global variable as you create it upon page loading:

```
var theDataView = null;
function pageLoad()
{
   theDataView = $create(Sys.UI.DataView, ...);
   :
}
```

Next, in the onchange handler you just call the refresh method on the global instance:

```
theDataView.refresh();
```

In this first example, I used a drop-down list to render the portion of the user interface responsible for triggering changes on the rest of page. The drop-down list element is particular because it incorporates the logic to raise a change event when one of its elements is selected.

ASP.NET AJAX, however, provides a more general mechanism to trigger change/notification events that result in page-specific operations. Let's rework the previous example using a plain handmade list instead of a drop-down list.

The sys:command Attribute

The list of countries is now generated using HTML tags for an unordered bulleted list. The template is as follows:

```
<fieldset>
   <legend><b>Countries</b></legend>
   <ul id="listOfCountries" class="sys-template">
       <li>
           {{ $dataItem }}
       </li>
   </ul>
</fieldset>
```

The template is programmatically attached to a DataView control for rendering purposes. Data to fill up the template is provided via an embedded JavaScript array. The JavaScript array that contains the list of countries is emitted from the server using the services of the ClientScript object on the Page class. Unlike the previous example, the Page_Load code doesn't include server-side binding operations:

```
protected void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
      string[] countries = new string [] {"[All]", "USA", ... "};
      string countriesAsString = "'[All]', 'USA', ...'";
      this.ClientScript.RegisterArrayDeclaration(
           "theCountries", countriesAsString);
   }
}
```

A second DataView control is instantiated on the client as the page is loaded. Here's the modified code for the JavaScript pageLoad function, as shown in **Figure 5**.

As you can see, the second DataView used to bind countries to a UL-based template has quite a different structure than the other.

The first difference is that the data property is used to import data. This is the correct procedure when embedded data is being used.

When the data source is an array of user-defined objects, you perform binding via the {{expression}} syntax. The content of the expression is typically the name of a public property exposed by the data item. In this example, instead, the source of data binding is a plain array of strings. Subsequently, the data item is a string with no public properties to refer to in the binding expression. In this case, you resort to the following:

```
<ul>
  <li>{{ $dataItem }}</li>
</ul>
```

The initialSelectedIndex and selectedItemClass properties configure the expected behavior of the DataView as far the selection of a displayed item is concerned.

The DataView can attach the template the built-in behavior of selection. The item at the position specified by initialSelectedIndex will be styled according to the CSS class set via the selectedItemClass property. You set initialSelectedIndex to -1 if you don't want any selection made on first display.

The list that results from the template is a plain UL list and, as such, it doesn't natively incorporate any logic to handle selection, as you see here:

```
<ul>
  <li>[All]</li>
  <li>USA</li>
  :
</ul>
```

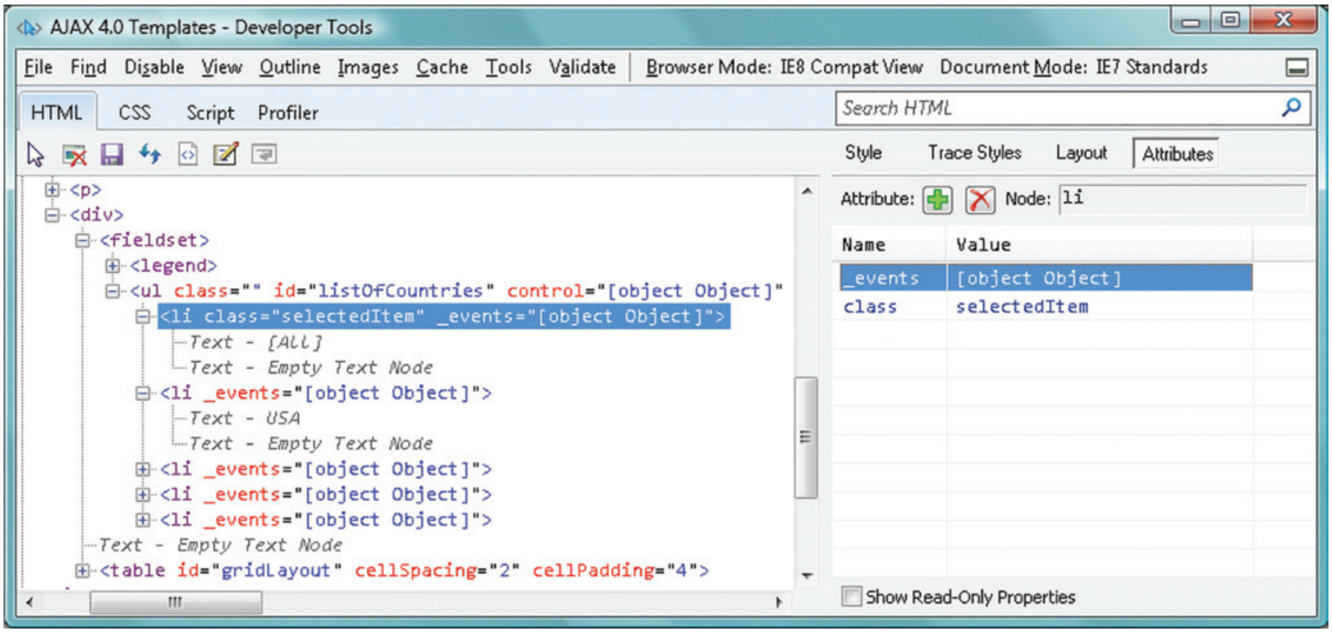
Figure 5 **JavaScript pageLoad Function**

```
<script language="javascript" type="text/javascript">
   function pageLoad()
   {
      $create(Sys.UI.DataView,
           {
             autoFetch: true,
             dataProvider: "/ajax40/mydataservice.asmx",
             fetchOperation: "LookupAllCustomers"
            },
            {},
            {},
            $get("grid")
      );
      $create(Sys.UI.DataView,
            {
                autoFetch: true,
                initialSelectedIndex: 0,
                selectedItemClass:"selectedItem",
                onCommand: refreshTemplate,
                data:theCountries
            },
            {},
            {},
            $get("listOfCountries")
       );
    }
</script>
```

By using the sys:command attribute on a HTML element within a template, you instruct the template builder to dynamically attach a bunch of event handlers to the element, as follows:

```
<ul id="listOfCountries" class="sys-template">
    <li sys:command="select">
        {{ $dataItem }}
    </li>
</ul>
```

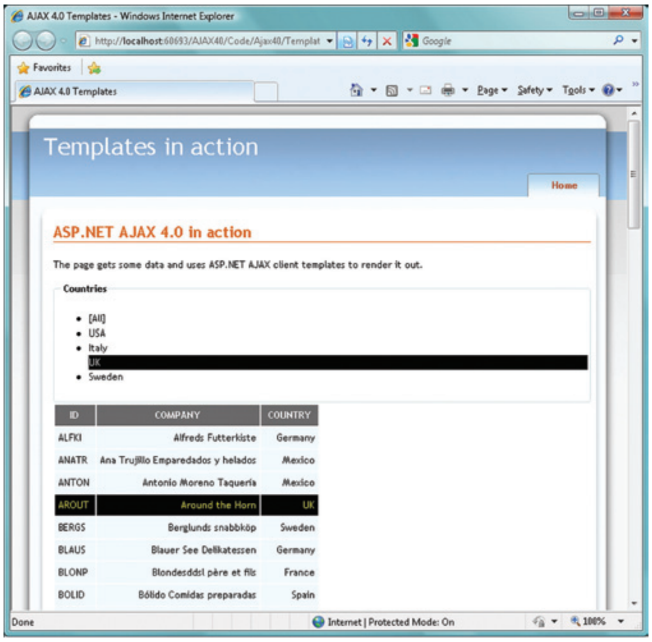
Figure 6 shows how such modified LI elements show up in the Developer Tools window of Internet Explorer 8. The sys:command attribute takes a string value that represent the name of the com-


AJAX 4.0 Templates - Developer Tools
File Find Disable View Outline Images Cache Tools Validate
Browser Mode: IE8 Compat View Document Mode: IE7 Standards
HTML CSS Script Profiler
Search HTML
Style Trace Styles Layout Attributes
Attribute: Node: li
Show Read-Only Properties

```
<p>
<div>
  <fieldset>
    <legend>
    <ul class="" id="listOfCountries" control="[object Object]"
      <li class="selectedItem" _events="[object Object]">
        Text - [All]
        Text - Empty Text Node
      <li _events="[object Object]">
        Text - USA
        Text - Empty Text Node
      <li _events="[object Object]">
      <li _events="[object Object]">
      <li _events="[object Object]">
  Text - Empty Text Node
  <table id="gridLayout" cellSpacing="2" cellPadding="4">
```

Name	Value
_events	[object Object]
class	selectedItem

Figure 6 **Event Handlers Dynamically Added as the Effect of sys:command**


AJAX 4.0 Templates - Windows Internet Explorer
http://localhost:60693/AJAX40/Code/Ajax40/Templat
Google
Favorites
AJAX 4.0 Templates
Page
Safety
Tools
Home
Done
Internet | Protected Mode: On
100%

Templates in action

ASP.NET AJAX 4.0 in action

The page gets some data and uses ASP.NET AJAX client templates to render it out.

Countries

- [All]
- USA
- Italy
- UK
- Sweden

ID	COMPANY	COUNTRY
ALFKI	Alfreds Futterkiste	Germany
ANATR	Ana Trujillo Emparedados y helados	Mexico
ANTON	Antonio Moreno Taquería	Mexico
AROUT	Around the Horn	UK
BERGS	Berglunds snabbköp	Sweden
BLAUS	Blauer See Delikatessen	Germany
BLONP	Blondesddsl père et fils	France
BOLID	Bólido Comidas preparadas	Spain

Figure 7 **Commands are Used to Handle Selection**

mand triggered. The name is actually up to you. Commands are triggered by clicking on the element. Common commands are select, delete, insert and update. When the command is triggered, the DataView fires the onCommand event. Here's the code that handles the onCommand event and refreshes the template:

```
<script type="text/javascript">
    var currentCountry = "";
    function refreshTemplate(args)
    {
      if (args.get_commandName() == "select")
      {
        // Pick up the currently selected item
        currentCountry = args.get_commandSource().innerHTML.trim();

        // Refresh
        var theDataView = $find("grid");
        theDataView.refresh();
      }
    }
</script>
```

Figure 8 **HTML Mapped Attributes**

Attribute	Description
sys:checked	Maps to the "checked" attribute of an INPUT checkbox element.
sys:disabled	Maps to the "disabled" attribute of an INPUT element.
sys:id	Maps to the "id" attribute of any HTML element.
sys:innerHTML	Maps to the "innerHTML" attribute of any HTML element.
sys:innerText	Maps to the "innerText" attribute of any HTML element.
sys:src	Maps to the "src" attribute of an IMG element.

The same approach can be used for a drop-down list as long as you emit it directly in HTML, as below:

```
<select>
  <option sys:command="select"> {{ $dataItem }} </option>
</select>
```

Note, though, that a bug prevents the preceding code to work as expected in beta 1. (**Figure 7** shows the sample page in action.)

HTML Attributes

In ASP.NET AJAX 4, a special bunch of sys: attributes specify ad hoc bindings for HTML attributes. Functionally speaking, these attributes are like HTML attributes and you won't notice any different behavior. **Figure 8** lists the namespaced HTML attributes.

All element attributes in a template can be prefixed with the sys: namespace. So what's the ultimate reason for using mapped attributes? And why are only a few of them are listed in **Figure 8**?

Often you want to bind to HTML attributes but you don't want to set the attribute itself to your {{…}} binding expression. From the DOM perspective, the binding expression is simply a value assigned to an attribute and it is processed as such. This, of course, may have some unpleasant side effects. For example, if you bind to the value attribute of an input element, or to the content of an element, the binding string may appear for a second to the user as the page is being loaded. The same happens if you are using binding expressions outside of a template (i.e., live binding or two-way binding). In addition, there are a bunch of HTML attributes—those listed in **Figure 8**—where the use of binding expressions may originate unwanted effects. For example, consider the following markup:

```
<img src="{{ URL }}" />
```

It triggers a request for the string "URL" rather than the value of the property URL on the data item.

Other issues you may face include XHTML validation issues and in general wrong attribute resolution by the browser. If you prefix such critical attributes with the sys namespace, you solve the issue.

So the best practice is to always prefix with the sys namespace any attributes being assigned a binding expression. The DOM doesn't care about namespaced attributes, so attributes retain their binding expression with no side effects until it is processed by the template builder.

Namespaced attributes are recommended in client-side rendering, even though they are not certainly mandatory except in situations where they can save you the effects of wrong HTML parsing.

Whole New World

Templates and data binding open up a whole new world of possibilities to ASP.NET AJAX developers. Next month, I'll be back to cover various types of binding, including live binding and master/detail views. ■

DINO ESPOSITO *is an architect at IDesign and the co-author of "Microsoft .NET: Architecting Applications for the Enterprise" (Microsoft Press, 2008). Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.*

THANKS to the following technical experts for reviewing this article: *Ed Essey and Stephen Walther.*

Claims-Based Authorization with WIF

Michele Leroux Bustamante

Over the past few years, federated security models and claims-based access control have become increasingly popular. In a federated security model, authentication can be performed by a Security Token Service (STS), and the STS can issue security tokens carrying claims that assert the identity of the authenticated user and the user's access rights. Federation allows users to authenticate in their own domain while being granted access to applications and services that belong to another domain—provided the domains have an established trust relationship. This approach removes the need to provision and manage duplicate accounts for a single user, and enables single sign-on (SSO) scenarios. Claims-based access is central to a federated security model whereby applications and services authorize access to features and functionality based on claims from issuers (the STS) in trusted domains. Claims can contain information about the user, roles or permissions, and this makes for a very flexible authorization model. Together, federated security and claims-based access enable a range of integration scenarios across applications, departments and partners in a wider ecosystem.

This article discusses:

- Federated security
- Claims-based authorization
- Windows Communication Foundation

Technologies discussed:

- WIF
- WCF

Code Download URL:

code.msdn.microsoft.com/mag200911WIF

Platform tools in this area have also come a long way. Windows Identity Foundation (WIF) is a rich identity model framework designed for building claims-based applications and services and for supporting active and passive federated security scenarios. With WIF, you can enable passive federation for any ASP.NET application, and integrate a claims-based authorization model into your ASP.NET applications and WCF services without breaking a sweat. Furthermore, WIF provides the plumbing to build custom STS implementations, and includes features and controls to support authentication scenarios that involve managed information cards and identity selectors such as Windows CardSpace.

WIF significantly reduces the code required to implement rich application scenarios that involve federated and claims-based security. In this two-part article, I'll focus on the framework's core functionality for enabling passive federation in ASP.NET applications and for supporting claims-based security models in both WCF and ASP.NET. I'll focus on WCF in this article and ASP.NET in a later one.

Why Federated and Claims-Based Security?

The benefits of federated and claims-based security can be seen in the context of a few distinct goals:

- Decoupling the authentication mechanism from applications and services.
- Replacing roles with claims as a more flexible, granular artifact for authorization.
- Reducing IT pain related to provisioning and deprovisioning users.
- Granting trusted domains, including possibly external federated partners, access to application features and functionality.

If even one of these goals rings true for your application scenario, adopting a claims-based model that can immediately or eventually involve federated security is incredibly useful.

When you design your applications and services, the authentication and authorization model is part of this design. For example, an intranet application typically expects users to authenticate to a particular domain with their Windows credentials, while an Internet application typically uses a custom credential store such as Microsoft SQL Server. Applications can also require certificates or Smart Card authentication, or support multiple credential types so that different groups of users can use the appropriate type. If your application will only (and always) expect users to authenticate with a single credential type, your job is easy. More often than not, however, the credential types supported by an application can evolve to support alternative modes of authentication or additional modes that accommodate a different set of users.

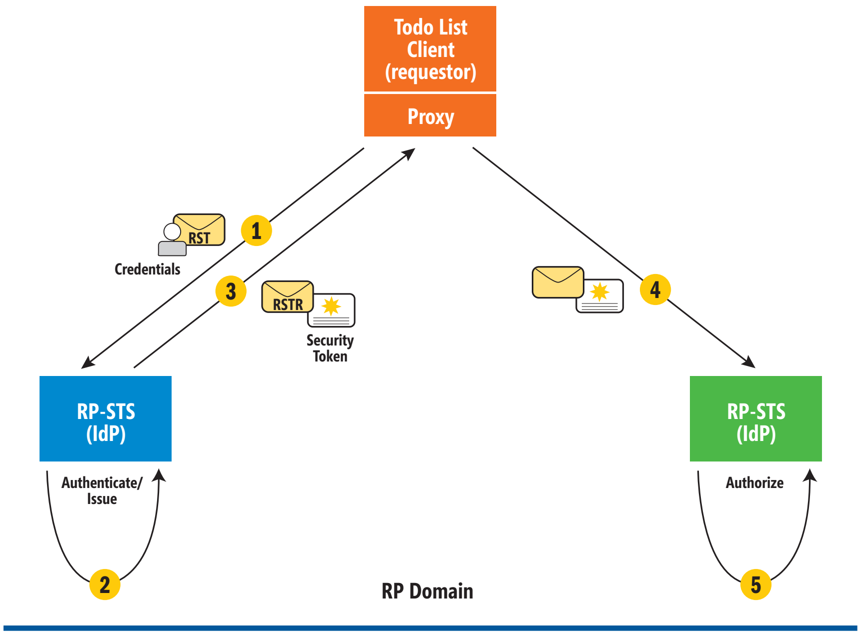


Figure 1 **A Simple Active Federation Scenario**

For example, an application might support internal users behind the firewall within a domain while also supporting external users over the Internet. When the security model for an application is decoupled from the mode of authentication—as it can be with a claims-based model—there is very little, if any, impact to the application when you introduce new modes of authentication.

In a similar vein, applications are more flexible if authorization is not tied to a fixed set of roles. If your application will always rely on a specific set of roles to authorize access, and if those roles will always carry the same meaning in terms of access rights to features and functionality, you're again in good shape. But the meaning of roles often varies across departments that use an application and thus require customization. That might mean evaluating roles differently depending on the user's domain, or allowing custom roles to be created to control access rights. WIF makes adopting a claims-based security model easy so you can decouple roles (if applicable) from the authorization mechanism. This way, logical roles can be mapped to a more granular set of claims, and the application authorizes access based on those claims. If modified or new roles warrant a different set of claims to be issued, the application isn't affected.

Of course, claims can be much more than just roles or permissions. One of the added benefits of working with a claims-based model is that a claim can carry information about an authenticated user, such as e-mail address, full name, birth date and so on. Claims can also be used to verify information, for example, without sharing a user's actual age or birth date (information that many users don't want to be public knowledge). A claim could indicate whether a user is at least the age required to perform an action (a Boolean claim indicating IsOver21 or IsOver13), or verify that a user belongs to a particular department without sharing a list of all departments the user belongs to.

Although decoupling the authentication mechanism and specific roles from applications and services makes accommodating change easier, the claims-based model is also central to federated security scenarios, which make granting access to users belonging to any trusted domain much easier. Federation reduces IT overhead and some of the risks associated with identity management. It removes the need to maintain user credentials across multiple applications or domains, and this helps reduce risks when provisioning and deprovisioning accounts across domains—for example, forgetting to delete an account in multiple places. Password synchronization when multiple copies of an account aren't managed also ceases to be a problem. Federation also facilitates SSO scenarios because users can log on to one application and be granted access to another (possibly across security domains) without having to authenticate again. Finally, adding new trust relationships between domains is also made easy with federated security platforms such as Active Directory Federation Server (ADFS) and WIF. Thus, extending an application to additional domains within a corporate entity, or even to external partner domains, is streamlined.

Active Federation with WIF

Active federation scenarios are based on the WS-Federation Active Requestor Profile (see the WS-Federation TC at oasis-open.org/committees/tc_home.php?wg_abbrev=wsfed) and the WS-Trust specification (see WS-Trust 1.3 at docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html). From a high level, WS-Trust describes a contract with four service operations: Issue, Validate, Renew and Cancel. Respectively, these operations are called by clients to request a security token, to validate a security token, to renew an expired security token and to cancel a security token that should no longer be used. Each operation processes messages in the form of a

Figure 2 **ITodoListService Definition**

```
[ServiceContract(Namespace="urn:TodoListApp/2009/06")]
public interface ITodoListService
{
    [OperationContract]
    List<TodoItem> GetItems();
    [OperationContract]
    string CreateItem(TodoItem item);
    [OperationContract]
    void UpdateItem(TodoItem item);
    [OperationContract]
    void DeleteItem(string id);
}
```

Request for Security Token (RST) and sends responses in the form of an RST Response (RSTR) following the WS-Trust specification. These WS-Trust features are implemented by an STS (or token issuer), an important participant in any federated security scenario.

A simple active federation scenario is illustrated in **Figure 1**. This scenario involves a Windows client application (the requestor), a WCF service (the relying party, or RP), and an STS belonging to the RP domain (RP-STS). As the figure shows, the client uses a WCF proxy to coordinate first authenticating to the RP-STS, then requesting a security token, and then calling the RP, passing the issued security token along with the request.

In this scenario, RP-STS is also the Identity Provider (IdP) for users authenticating to the RP domain. That means that the RP-STS is responsible for authenticating users, asserting the identity of those users, and issuing claims relevant to the RP for authorization. The RP verifies that the security token is issued by RP-STS and authorizes access based on the issued claims.

I've created a Todo List application to facilitate implementation discussions for this scenario. The accompanying code sample includes a WPF client, a WCF service, and an active STS implemented with WIF. To provide further context, the WCF service, TodoListService, implements the ITodoListService contract shown in **Figure 2**. The client calls the service by using a WCF proxy to get all the Todo items, and to add, update or delete items. The TodoListService relies on create, read, update and delete claims to authorize access to its operations.

To implement this active federation scenario, you need to follow these four steps:

1. Expose a federated security WCF endpoint for the TodoListService.
2. Generate a WCF proxy for the client application and initialize the proxy with credentials to authenticate to the RP-STS.
3. Enable WIF for the TodoListService to enable claims-based authorization.
4. Place permission demands (IsInRole) or other authorization checks to control access to service operations or other functionality.

I'll discuss these steps in the sections that follow.

Exposing Federated Endpoints

Claims-based WCF services typically expose federated endpoints that receive issued tokens such as those based on the SAML standard. WCF supplies two bindings to support federated security scenarios with WS-Trust. WSFederationHttpBinding is the original standard binding based on WS-Trust 2005 (an earlier version of the protocol), and WS2007FederationHttpBinding is the latest version of the binding (released with Microsoft .NET Framework 3.5) and supports WS-Trust 1.3, the approved standard. Typically, you should use WS2007FederationHttpBinding unless an interoperability requirement dictates the use of the earlier version. An STS based on ADFS version 2 or WIF can support either version of WS-Trust.

When you expose a federated endpoint for a service, you usually provide information about the expected security token format, the required and optional claim types and the trusted token issuer. **Figure 3** shows the system.serviceModel listing for the TodoListService, which exposes a single federated endpoint over WS2007FederationHttpBinding.

Federated security scenarios typically rely on SAML tokens, although this is not a strict requirement. For this scenario, SAML 1.1 tokens

Figure 3 **The Federated Endpoint Exposed by the TodoListService**

```
<system.serviceModel>
  <services>
    <service name="TodoList.TodoListService"
behaviorConfiguration="serviceBehavior">
      <endpoint address="" binding="ws2007FederationHttpBinding"
bindingConfiguration="wsFed" contract="Contracts.ITodoListService" />
      <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8000/TodoListService"/>
        </baseAddresses>
      </host>
    </service>
  </services>
  <bindings>
    <ws2007FederationHttpBinding>
      <binding name="wsFed">
        <security mode="Message" issuedTokenType=
"http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-.1#SAMLV1.1"
issuedKeyType="SymmetricKey" negotiateServiceCredential="true">
          <message>
            <claimTypeRequirements>
              <add claimType=
"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
isOptional="false"/>
              <add claimType= "urn:TodoListApp/2009/06/claims/permission"
isOptional="false"/>
            </claimTypeRequirements>
            <issuerMetadata address="http://localhost:8010/rpsts/mex" />
          </message>
        </security>
      </binding>
    </ws2007FederationHttpBinding>
  </bindings>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehavior">
        <serviceMetadata/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

are used, as indicated by the issuedTokenType URI (docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1). For an alternate token type, such as SAML 1.0 or SAML 2.0, use the URI for that standard. Of course, the STS indicated in the federated binding configuration must support the token type you are requesting.

Other relevant settings in the message section include issuedKeyType and negotiateServiceCredential. The issuedKeyType setting indicates whether the proof key (see blogs.msdn.com/vbertocci/archive/2008/01/02/on-prooftokens.aspx) is symmetric (the default) or asymmetric (carries more overhead). Once again, this setting must be compatible with the STS. If negotiateServiceCredential is set to true, the client doesn't need access to the RP public key a priori, but negotiation is not an interoperable protocol. If the client is not a WCF client, you should set negotiateServiceCredential to false. But don't worry. If it is set to false, proxy generation with SvcUtil supplies the client with a base64 encoded copy of the RP's public key.

Federated security scenarios typically rely on SAML tokens, although this is not a strict requirement.

The claim types supplied in the claimTypeRequirements section indicate the required and optional claim types that the service relies on for authorization. In this case, the service expects a name claim to identify the user and at least one permission claim—a custom claim type that indicates the user's rights to create, read, update or delete Todo items. (These claim types are listed later in **Figure 4**.) The list of claim types is included in the service metadata so that clients are able to include this information in the RST. Frequently, the STS knows the claims it will issue for a particular RP, which means that the list doesn't need to be exhaustive in the federation binding.

The trusted token issuer for this scenario is RP-STS, which happens to be implemented with WIF. RP-STS exposes a single WS-Trust endpoint at http://localhost:8010/rpsts, and its metadata exchange address is located at http://localhost:8010/rpsts/mex. In **Figure 3**, the issuer's metadata address is supplied in the issuerMetadata section so that when the client generates the proxy, it can discover the available STS endpoints.

Suppose the STS were to expose multiple endpoints—for example, to authenticate intranet users with Windows credentials at http://localhost:8010/rpsts/internal and to authenticate Internet users with a username and password at http://localhost:8010/rpsts/external. The RP service can opt to specify a particular issuer endpoint associated with its federated endpoint configuration so that when clients generate a proxy, the configuration to communicate with the STS matches that endpoint instead of the first compatible endpoint. You accomplish this by supplying an address for both the issuerMetadata and issuer elements as follows:

```
<issuerMetadata address="http://localhost:8010/rpsts/mex" />
<issuer address="http://localhost:8010/rpsts/mex/external" />
```

The advantage of this approach is to simplify proxy generation for clients if there are multiple STS endpoints to choose from and the RP wants to influence which one is used. If the RP doesn't care which endpoint the client authenticates to, it is best to supply only the issuerMetadata setting and let the client application determine the appropriate endpoint for authentication.

Keep in mind that if the service configuration omits the issuerMetadata element and supplies only the issuer address, the address should evaluate to the issuer's logical URI (http://localhost:8010/rpsts/issuer), which may not necessarily map to a physical STS endpoint address. An equivalent configuration at the client will prompt the user to select a managed information card from the same issuer (via Windows CardSpace), and the card must also meet the criteria of the requested token format and claim types. For more information on active federation scenarios with Windows CardSpace, see wpfandcardspace.codeplex.com.

Figure 4 **Claims Issued Per User for the Todo List Application Scenario**

Username	Claim Type	Claim Value
Admin	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	Admin
	urn:TodoListApp/2009/06/claims/permission	/create
	urn:TodoListApp/2009/06/claims/permission	/read
	urn:TodoListApp/2009/06/claims/permission	/update
	urn:TodoListApp/2009/06/claims/permission	/delete
User	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	User
	urn:TodoListApp/2009/06/claims/permission	/create
	urn:TodoListApp/2009/06/claims/permission	/read
Guest	http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	Guest
	urn:TodoListApp/2009/06/claims/permission	/read

Client Proxy Generation

When you generate a proxy for a Windows client using SvcUtil or Add Service Reference, the metadata exchange address for the issuer is used to gather information about the endpoints exposed by the issuer. To reiterate, some possible scenarios are:

- If the federation binding for the RP service endpoint supplies an issuer metadata address without a specific issuer address, the client configuration will include the first protocol-compatible STS endpoint with any other compatible endpoints commented for the client developer to optionally use.

- If the federation binding for the RP service supplies an issuer metadata address and a specific issuer address, the client configuration will include that specific address (assuming it is protocol compatible).
- If the federation binding for the RP service supplies only a metadata address, the client configuration will include only the metadata address as well, without a binding configuration for the issuer. This means an identity selector such as CardSpace will be triggered, as I mentioned earlier.

Assuming that the client generates a proxy for the TodoList-Service whose configuration is shown in **Figure 3** and that the STS exposes a single endpoint, the client-side version of the WS-2007FederationHttpBinding configuration will include the following issuer and issuerMetadata settings:

```
<issuer address="http://localhost:8010/rpsts"
        binding="ws2007HttpBinding"
        bindingConfiguration="http://localhost:8010/rpsts">
  <identity>
    <certificate encodedValue="[base64 encoded RP-STS certificate]" />
  </identity>
</issuer>
<issuerMetadata address="http://localhost:8010/rpsts/mex" />
```

Note that the issuer element specifies the issuer endpoint and the required binding configuration to communicate with that endpoint. In this case, the client authenticates to the STS with a user name and password using message security, as shown in the following WS2007HttpBinding configuration:

```
<ws2007HttpBinding>
    <binding name="http://localhost:8010/rpsts" >
        <security mode="Message">
            <message clientCredentialType="UserName"
                     negotiateServiceCredential="false"
                     algorithmSuite="Default"
                     establishSecurityContext="false" />
        </security>
    </binding>
</ws2007HttpBinding>
```

The client endpoint associates the federation binding configuration with the RP endpoint:

```
<client>
  <endpoint address="http://localhost:8000/TodoListService"
            binding="ws2007FederationHttpBinding"
            bindingConfiguration="wsFed"
            contract="TodoList.ITodoListService" name="default">
    <identity>
      <certificate encodedValue="[base64 encoded RP certificate" />
    </identity>
  </endpoint>
</client>
```

With this configuration, the client proxy need only be initialized with a valid user name and password before calling the service:

```
TodoListServiceProxy _Proxy = new TodoListServiceProxy("default");

if (!ShowLogin()) return;

this._Proxy.ClientCredentials.UserName.UserName = this.Username;
this._Proxy.ClientCredentials.UserName.Password = this.Password;
this._TodoItems = this._Proxy.GetItems();
```

Token Issuance

The proxy first supplies credentials to authenticate to RP-STS, sending an RST that asks for a SAML 1.1 token, indicating that the RP requires at least one name and permission claim. The user is authenticated against the STS credential store, and the appropriate claims are issued for the authenticated user. The proxy then processes the RSTR that carries the issued token and passes that token to the RP to establish a secure session for the authenticated user.

For this example, the STS was built with WIF and authenticates users against a custom credential store, issuing claims for each user according to **Figure 4**.

Note that an STS based on ADFS version 2 authenticates users against the Windows domain and issues claims according to your ADFS configuration. A custom STS based on WIF can authenticate users against a credential store of your choosing, but you must roll your own code to manage the credential store and the relevant claims-mapping process.

Identity Model Configuration

To enable claims-based authorization for your WCF services using WIF, you initialize the ServiceHost instance for federation. You can do this programmatically by calling the ConfigureServiceHost method exposed by the FederatedServiceCredentials type, as follows:

```
ServiceHost host = new ServiceHost(typeof(TodoList.TodoListService));
FederatedServiceCredentials.ConfigureServiceHost(host);
host.Open();
```

You can achieve the same result declaratively by using the behavior extension ConfigurationServiceHostBehaviorExtension:

```
<serviceBehaviors>
  <behavior name="fedBehavior" >
    <federatedServiceHostConfiguration/>
    <serviceMetadata />
  </behavior>
</serviceBehaviors>
```

In either case, the ServiceHost is assigned an instance of the FederatedServiceCredentials type to drive claims-based authorization behavior for the service. This type can be initialized either programmatically or by the microsoft.identityModel configuration section for the service. Identity model settings are specific to WIF and supply settings for claims-based authorization in ASP.NET and WCF applications, most of which are summarized in **Figure 5**.

For WCF services that use WIF, you no longer need to initialize the ServiceHost with typical WCF authentication and authorization behaviors. WIF supersedes this and provides a cleaner way to configure security in general. (WIF is useful beyond claims-based and federated scenarios). **Figure 6** shows the identity model settings used for the TodoListService.

The issuerNameRegistry setting is used to specify any trusted certificate issuers. If you use the ConfigurationBasedIssuerNameRegistry as shown in **Figure 6**, you must provide a list of trusted certificate issuers by specifying their thumbprints. At run time, the ConfigurationBasedIssuerNameRegistry checks X509 security tokens against this list and rejects those with thumbprints not found in the list. You can use the SimpleIssuerNameRegistry to allow any X509 or RSA token, but more likely you will supply a custom IssuerNameRegistry type to validate tokens using your own heuristics if the ConfigurationBasedIssuerNameRegistry doesn't do the trick.

The configuration in **Figure 6** rejects any tokens that are not signed by the RP-STS (using the certificate thumbprint for CN=RPSTS).

Figure 5 **Summary of the Essential microsoft.identityModel Elements**

Section	Description
issuerNameRegistry	Specifies a list of trusted certificate issuers. This list is primarily useful for validating the token signature so that tokens signed by untrusted certificates will be rejected.
audienceUris	Specifies a list of valid audience URIs for incoming SAML tokens. Can be disabled to allow any URI.
securityTokenHandlers	Customizes configuration settings for token handlers or supplies custom token handlers to control how tokens are validated, authenticated and serialized.
maximumClockSkew	Adjusts the allowed time difference between tokens and application servers for token validity. The default skew is 5 minutes.
certificateValidation	Controls how client certificates are validated.
serviceCertificate	Supplies a service certificate for decrypting incoming tokens.
claimsAuthenticationManager	Supplies a custom ClaimsAuthenticationManager type to customize or replace the IClaimsPrincipal type to be attached to the request thread.
claimsAuthorizationManager	Supplies a custom ClaimsAuthorizationManager type to control access to functionality from a central component.
federatedAuthentication	Supplies settings specific to passive federation.

The following configuration instead specifies a custom IssuerNameRegistry type, TrustedIssuerNameRegistry:

```
<issuerNameRegistry type="TodoListHost.TrustedIssuerNameRegistry,
TodoListHost"/>
```

The TrustedIssuerNameRegistry implementation is used to achieve the same result—rejecting tokens not signed by CN=RPSTS by checking the subject name of the incoming token:

```
public class TrustedIssuerNameRegistry : IssuerNameRegistry
{
    public override string GetIssuerName(SecurityToken securityToken)
    {
        X509SecurityToken x509Token = securityToken as
            X509SecurityToken;
        if (x509Token != null)
        {
            if (String.Equals(x509Token.Certificate.SubjectName.Name,
                "CN=RPSTS"))
            {
                return x509Token.Certificate.SubjectName.Name;
            }
        }

        throw new SecurityTokenException("Untrusted issuer.");
    }
}
```

The serviceCertificate setting in **Figure 6** indicates the certificate to be used to decrypt incoming security tokens, assuming they are encrypted for the RP by the issuing STS. For the Todo List application, the RP-STS encrypts tokens using the public key for the RP, CN=RP.

Usually, the SAML token includes an audience URI element that evaluates to the RP, indicating who the token was issued for. You can explicitly refuse tokens that were not intended to be sent to the RP. By default, the audienceUris mode is set to Always, which means that you must supply at least one URI for validation against incoming tokens. In **Figure 6**, the configuration allows only SAML tokens that include an audience URI matching the TodoListService address. Although not generally recommended, you can set the audienceUris mode to Never to suppress evaluation of the audience restriction condition for an incoming SAML token:

```
<audienceUris mode="Never"/>
```

Be aware that when the client sends an RST to the STS, it usually includes an AppliesTo setting that indicates who the token should be issued to—the RP. The STS can use this information to populate the SAML token audience Uri.

The certificateValidation setting controls how incoming X509 tokens—those used for token signatures, for example—are validated. In **Figure 6**, certificateValidationMode is set to PeerTrust, which means that certificates are valid only if the associated certificate is found in the TrustedPeople store. This setting is more appropriate than PeerOrChainTrust (the default) for token issuer validation because it requires you to explicitly install the trusted certificate in the certificate store. PeerOrChainTrust indicates that signatures are also authorized if the root certificate authority (CA) is trusted, which on most machines includes a significant list of trusted CAs.

I'll discuss a few of the other settings from **Figure 5** and **Figure 6** shortly. One other point to make about the subject of WIF initialization is that you can also programmatically initialize an instance of FederatedServiceCredentials and pass it to ConfigureServiceHost rather than initializing from the microsoft.identityModel section. The following code illustrates this:

```
ServiceHost host = new ServiceHost(typeof(TodoList.TodoListService));

ServiceConfiguration fedConfig = new ServiceConfiguration();
fedConfig.IssuerNameRegistry = new TrustedIssuerNameRegistry();
fedConfig.AudienceRestriction.AudienceMode = AudienceUriMode.Always;
fedConfig.AudienceRestriction.AllowedAudienceUris.Add(new
Uri("http://localhost:8000/TodoListService"));
fedConfig.CertificateValidationMode =
X509CertificateValidationMode.PeerTrust;
fedConfig.ServiceCertificate = CertificateUtil.GetCertificate(
StoreName.My, StoreLocation.LocalMachine, "CN=RP");

FederatedServiceCredentials fedCreds =
new FederatedServiceCredentials(fedConfig);

FederatedServiceCredentials.ConfigureServiceHost(host,fedConfig);
host.Open();
```

Programmatic initialization is particularly useful for initializing the ServiceHost from database settings that apply to an entire server farm.

WIF Component Architecture

When you apply WIF behavior to a ServiceHost, several WIF components are initialized to facilitate claims-based authorization—many of them WCF extensions. Ultimately, this leads to a

Figure 6 **Identity Model Settings Frequently Supplied for WCF Services**

```
<microsoft.identityModel>
  <service>
    <issuerNameRegistry type="Microsoft.IdentityModel.Tokens.
      ConfigurationBasedIssuerNameRegistry, Microsoft.IdentityModel,
      Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35">
      <trustedIssuers>
        <add name="http://localhost:8010/rpsts" thumbprint=
"c3 95 cd 4a 74 09 a7 77 d4 e3 de 46 d7 08 49 86 76 1a 99 50"/>
      </trustedIssuers>
    </issuerNameRegistry>
    <serviceCertificate>
      <certificateReference findValue="CN=RP" storeLocation="LocalMachine"
         storeName="My" x509FindType="FindBySubjectDistinguishedName"/>
    </serviceCertificate>
    <audienceUris mode="Always">
      <add value="http://localhost:8000/TodoService"/>
    </audienceUris>
    <certificateValidation certificateValidationMode="PeerTrust" />
    <securityTokenHandlers>
      <remove type="Microsoft.IdentityModel.Tokens.Saml11.
         Saml11SecurityTokenHandler, Microsoft.IdentityModel,
         Version=1.0.0.0, Culture=neutral,
         PublicKeyToken=31bf3856ad364e35"/>
      <add type="Microsoft.IdentityModel.Tokens.Saml11.
         Saml11SecurityTokenHandler, Microsoft.IdentityModel,
         Version=1.0.0.0, Culture=neutral,
         PublicKeyToken=31bf3856ad364e35">
        <samlSecurityTokenRequirement >
          <roleClaimType
            value="urn:TodoListApp/2009/06/claims/permission"/>
        </samlSecurityTokenRequirement>
      </add>
    </securityTokenHandlers>
    <claimsAuthorizationManager
      type="TodoList.CustomClaimsAuthorizationManager, TodoList"/>
  </service>
</microsoft.identityModel>
```

ClaimsPrincipal being attached to the request thread, and this supports claims-based authorization. **Figure 7** captures the relationship between the core WIF components and the ServiceHost.

The FederatedServiceCredentials type replaces the default ServiceCredentials behavior, and the IdentityModelServiceAuthorizationManager (installed during initialization of FederatedServiceCredentials) replaces the default ServiceAuthorizationBehavior. FederatedServiceCredentials also constructs a FederatedSecurityTokenManager instance. Collectively, these types drive authentication and authorization for each request, with the help of the ClaimsAuthenticationManager, the ClaimsAuthorizationManager, and the SecurityTokenHandler that applies to the specific request.

Figure 8 illustrates the flow of communication to these components that leads to constructing a security principal for the request thread—in this case, a ClaimsPrincipal type—and to opportunities to authorize access based on this security principal.

The FederatedSecurityTokenManager returns the appropriate token handler for the request—which in this case would be the Saml11SecurityTokenHandler—providing it with a reference to the ClaimsAuthorizationManager. The token handler constructs a ClaimsIdentity from the incoming token, creates the ClaimsPrincipal (via a wrapper class) and passes it to the ValidateToken method for the ClaimsAuthorizationManager. This yields an opportunity to modify or replace the ClaimsPrincipal that will be attached to the request thread. The default implementation merely returns the same ClaimsPrincipal provided:

```
public virtual IClaimsPrincipal Authenticate(string resourceName,
IClaimsPrincipal incomingPrincipal)
{
    return incomingPrincipal;
}
```

You might consider providing a custom ClaimsAuthenticationManager to transform the incoming claims from the security token into something that the RP can use to authorize access. For this example, however, the SAML token carries the appropriate RP claims issued by RP-STS, and so the ClaimsPrincipal constructed from those claims works for authorization.

Next, the IdentityModelServiceAuthorizationManager, which references the ClaimsAuthorizationManager, calls its CheckAccess method, yielding an opportunity to customize how access is controlled. The default implementation does not restrict access:

```
public virtual bool CheckAccess(AuthorizationCo
ntext context)
{
    return true;
}
```

The AuthorizationContext parameter supplies access to the ClaimsPrincipal and its associated claims, a collection of actions relevant to the request (such as a URI indicating the service operation to be

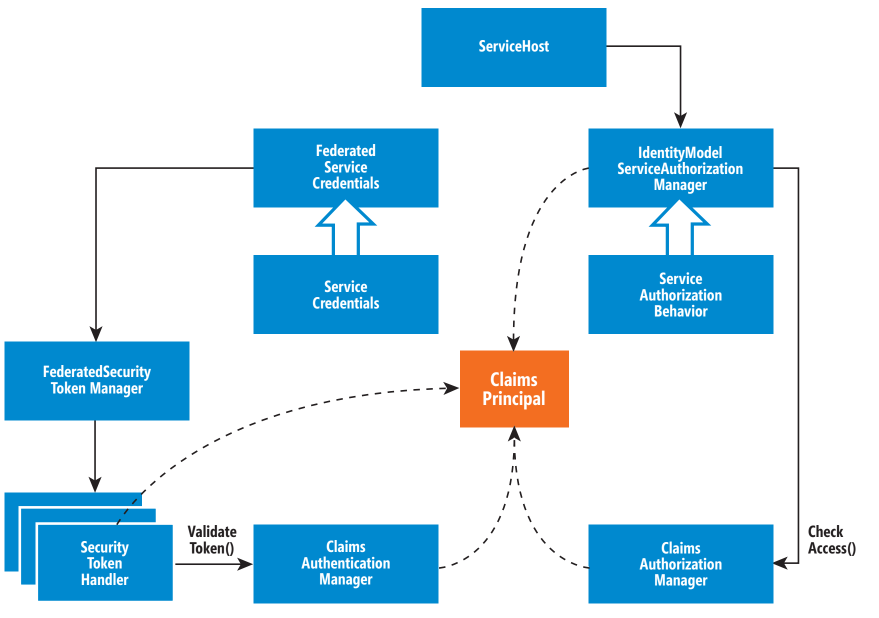
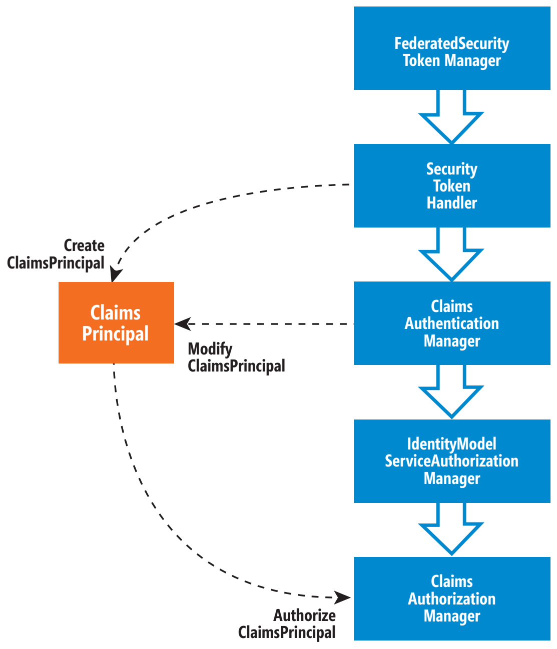

ServiceHost
Federated Service Credentials
IdentityModel ServiceAuthorization Manager
Service Credentials
Service Authorization Behavior
FederatedSecurity Token Manager
Claims Principal
Security Token Handler
Validate Token()
Claims Authentication Manager
Claims Authorization Manager
Check Access()

Figure 7 Core Components Installed with WIF



FederatedSecurity Token Manager
Security Token Handler
Claims Authentication Manager
IdentityModel ServiceAuthorization Manager
Claims Authorization Manager
Create ClaimsPrincipal
Claims Principal
Modify ClaimsPrincipal
Authorize ClaimsPrincipal

Figure 8 **Components That Create and Can Authorize Against the ClaimsPrincipal**

called) and information about the resource associated with the request (for example, the service URI), which can be useful to disambiguate calls to multiple services passing through the same authorization path. To implement centralized authorization, you can supply a custom ClaimsAuthorizationManager. I will describe an example when I discuss techniques for authorization.

Role-based security in the .NET Framework is founded on the premise that a security principal based on IPrincipal is attached to each thread, and this security principal wraps the identity of the authenticated user in an IIdentity implementation. Without WIF, WCF attaches a security principal to each request thread based on the system.serviceModel configuration for authentication and authorization.

An IIdentity type is constructed based on the type of credentials presented for authentication. For example, a Windows credential will evaluate to a WindowsIdentity, an X.509 certificate to an X509Identity, and a UserName token to a GenericIdentity.

The ServiceAuthorizationBehavior controls the type of IPrincipal wrapper for the identity. For example, with Windows authorization, a WindowsPrincipal is constructed; with the ASP.NET membership provider, a RoleProviderPrincipal. Otherwise, a custom authorization policy is used to construct an IPrincipal object of your choosing. The IPrincipal object exposes an IsInRole method that can be called directly or indirectly through permission demands to control access to features and functionality.

WIF extends this model by supplying ClaimsPrincipal and ClaimsIdentity types—based on IClaimsPrincipal and IClaimsIdentity—that ultimately derive from IPrincipal and IIdentity. All tokens are mapped to a ClaimsIdentity with WIF. When each incoming security token is validated, its associated SecurityTokenHandler type constructs a ClaimsIdentity, supplying it the appropriate claims. This ClaimsIdentity is wrapped in a ClaimsIdentityCollection (in the event a token produces multiple ClaimsIdentity instances), and this collection is wrapped in a ClaimsPrincipal and attached to the request thread. It is this ClaimsPrincipal that is the heart of WIF authorization for your WCF services.

Claims-Based Authorization

For WCF services, your approach to authorization will likely involve one of the following techniques:

- Use the ClaimsPrincipal to perform dynamic IsInRole checks.
- Use the PrincipalPermission type to perform dynamic permission demands.
- Use the PrincipalPermissionAttribute to supply declarative permission demands at each operation.
- Provide a custom ClaimsAuthorizationManager to centralize access checks in a single component.

The first three of these options ultimately rely on the IsInRole method exposed by the ClaimsPrincipal type. This doesn't exactly mean you are doing role-based security; it just means that you select a role-claim type so that the correct claims are checked against the requested claims passed to IsInRole. The default role-claim type for WIF is schemas.microsoft.com/ws/2008/06/identity/claims/role. If the STS associated with the federation scenario issues this type of claim, you can optionally control access based on this claim type. For the Todo List application scenario, I mentioned that a custom permission claim type is used for authorization, so the identity model configuration must specify this as the role-claim type to facilitate IsInRole checks.

You provide the role claim type to the SecurityTokenHandler for the expected token type, in this case the Saml11Security-

Figure 9 **Custom ClaimsAuthorizationManager Implementation**

```
class CustomClaimsAuthorizationManager : ClaimsAuthorizationManager
{
    public CustomClaimsAuthorizationManager()
    {
    }

    public override bool CheckAccess(AuthorizationContext context)
    {

        if (context.Resource.Where(x=> x.ClaimType ==
            System.IdentityModel.Claims.ClaimTypes.Name && x.Value ==
            "http://localhost:8000/TodoListService").Count() > 0)
        {
            if (context.Action.Where(x=> x.ClaimType ==
                System.IdentityModel.Claims.ClaimTypes.Name && x.Value ==
                Constants.Actions.GetItems).Count() > 0)
            {
                return
                    context.Principal.IsInRole(
                       Constants.Permissions.Read);
            }

                      // other action checks for TodoListService
        }
        return false;
    }
}
```

TokenHandler. As **Figure 6** illustrates, you can modify the default configuration for a SecurityTokenHandler by removing it and then adding the same one again, specifying the preferred property settings. SAML token handlers have a samlSecurityTokenRequirement section in which you can provide a setting for the name or role-claim type, along with other settings related to certificate validation and Windows tokens. For this scenario, I supplied a custom role claim type:

```
<samlSecurityTokenRequirement >
  <roleClaimType value= "urn:TodoListApp/2009/06/claims/permission"/>
</samlSecurityTokenRequirement>
```

What this means is that any time IsInRole is called for the ClaimsPrincipal, I check for a valid permission claim. One way to accomplish this is to explicitly call IsInRole before a section of code executes that requires a particular claim. You can access the current principal through the Thread.CurrentPrincipal property as follows:

```
if (!Thread.CurrentPrincipal.
IsInRole("urn:TodoListApp/2009/06/claims/permission/delete"))
  throw new SecurityException("Access is denied.");
```

Aside from explicit IsInRole checks at run time, you can also write classic role-based permission demands using the PrincipalPermission type. You initialize the type with the required role claim (the second constructor parameter), and when Demand is called, the IsInRole method of the current principal is called. An exception is thrown if the claim is not found:

```
PrincipalPermission p = new PrincipalPermission("",
"urn:TodoListApp/2009/06/claims/permission/delete");
p.Demand();
```

You can also build a PermissionSet to collect several claims to check for:

```
PermissionSet ps = new PermissionSet(PermissionState.Unrestricted);
ps.AddPermission(new PrincipalPermission("", "urn:TodoListApp/2009/06/
claims/permission/create"));
ps.AddPermission(new PrincipalPermission("", "urn:TodoListApp/2009/06/
claims/permission/read"));
ps.Demand();
```

If access checks apply to the entire service operation, you can apply the PrincipalPermissionAttribute instead, which is a nice way to declaratively associate required claims to the operation being called. These attributes can also be stacked to check for multiple claims:

```
[PrincipalPermission(SecurityAction.Demand, Role = Constants.
Permissions.Create)]
[PrincipalPermission(SecurityAction.Demand, Role = Constants.
Permissions.Read)]
public string CreateItem(TodoItem item)
```

In some cases, you might find it useful to centralize authorization to a single component, which means you would provide a custom ClaimsAuthorizationManager to perform access checks. **Figure 6** illustrates how to configure a custom ClaimsAuthorizationManager, and the implementation of this for the TodoListService is shown in **Figure 9** (partially listed for brevity).

The ClaimsAuthorizationManager provides an override for CheckAccess that receives an AuthorizationContext parameter with reference to the resource (in this case, the service URI); a collection of actions (in this case, a single action indicating the service operation URI); and the ClaimsPrincipal, which is not yet attached to the request thread. You can check the resource if the component is shared across services, as this example does for illustration. Primarily, you will check the action against a list of service operation URIs and perform IsInRole checks according to the requirements of the operation.

Generally, I'm not a big fan of decoupling the authorization check from the protected operation or code block. It is much easier to maintain code that is declared at a location in context with the activity.

In some cases, you might find it useful to centralize authorization to a single component.

To Be Continued

At this point you should have a pretty good idea of how to set up an active federation scenario with WCF and WIF, including understanding federation bindings for WCF and proxy generation semantics; the token issuance process; configuring WIF at the service; and implementing various claims-based authorization techniques. In a follow-up article, I will move on to passive federation with ASP.NET and WIF. ■

Michele Leroux Bustamante *is chief architect at IDesign, Microsoft regional director for San Diego and a Microsoft MVP for Connected Systems. Her latest book is "Learning WCF." Reach her at mlb@idesign.net or visit idesign.net. She also blogs at dasblonde.net.*

Using Active Directory Federation Services 2.0 in Identity Solutions

Zulfiqar Ahmed

As a developer, you probably know something about Windows Identity Foundation (WIF), formerly called "Geneva Framework," which provides a powerful API to claims-enable your applications and to create custom security token services. Perhaps less familiar to you is Active Directory Federation Services version 2.0 (AD FS 2.0), originally code named "Geneva server," which is an enterprise-ready federation and single-sign-on (SSO) solution. AD FS 2.0 is an evolution of AD FS 1.0, and it supports both active (WS-Trust) and passive (WS-Federation and SAML 2.0) scenarios.

In this article, I start with an overview of AD FS 2.0 and then provide insight into how developers can use AD FS 2.0 in their identity solutions. The focus is on the token issuance functionality of AD FS 2.0, based on the Beta 2 release. As you can see from **Figure 1**, this token issuance is only one small piece of AD FS 2.0, but it is one of particular interest to .NET developers moving toward federated identity. Architecturally, AD FS 2.0 is built on top of WIF and Windows Communication Foundation (WCF), so if you're familiar with these technologies, you should feel right at home with AD FS 2.0.

This article discusses:

- Token issuance functionality in AD FS 2.0
- AD FS 2.0 as an identity provider
- AD FS 2.0 STS interactions with WCF
- Federating AD FS 2.0 with another STS

Technologies discussed:

Active Directory Federation Services 2.0, Windows Identity Foundation, Windows Communication Foundation, Visual Studio, ASP.NET

Code Download URL:

code.msdn.microsoft.com/mag200911ADFS2

Overview of AD FS 2.0

At a high level, AD FS 2.0 is a collection of the services shown in **Figure 2**.

At the core of AD FS 2.0 is a security token service (STS) that uses Active Directory as its identity store and Lightweight Directory Access Protocol (LDAP), SQL or a custom store as an attribute store. The STS in AD FS 2.0 can issue security tokens to the caller using various protocols, including WS-Trust, WS-Federation and Security Assertion Markup Language (SAML) 2.0. The AD FS 2.0 STS also supports both SAML 1.1 and SAML 2.0 token formats.

AD FS 2.0 is designed with a clean separation between wire protocols and the internal token issuance mechanism. Different wire protocols are transformed into a standardized object model at the entrance of the system while internally AD FS 2.0 uses the same object model for every protocol. This separation enables AD FS 2.0 to offer a clean extensibility model, independent of the intricacies of different wire protocols. Further details of AD FS 2.0 extensibility will be provided in the AD FS 2.0 SDK prior to RTM.

AD FS 2.0 as an Identity Provider

You can use AD FS 2.0 in several common scenarios. The simplest and most common scenario is to use AD FS 2.0 as an identity pro-

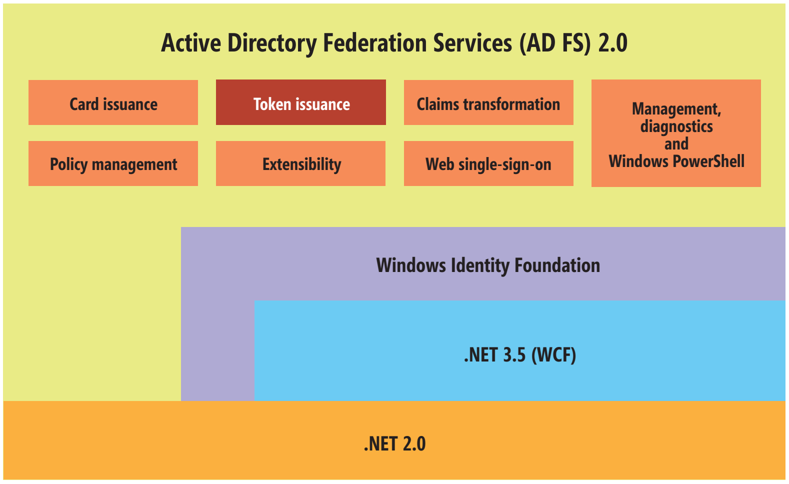

Active Directory Federation Services (AD FS) 2.0
Card issuance
Token issuance
Claims transformation
Management, diagnostics and Windows PowerShell
Policy management
Extensibility
Web single-sign-on
Windows Identity Foundation
.NET 3.5 (WCF)
.NET 2.0

Figure 1 **Architecture of AD FS 2.0**

vider so that it can issue SAML tokens for the identities it manages. For that, a new relying party needs to be created. A relying party in AD FS 2.0 is a representation of an application (a Web site or a Web service) and contains all the security-related information, such as encryption certificate, claims transformation rules and so on.

Setting Up a Relying Party

Setting up a new relying party via AD FS 2.0 is easy. You can access the Add Relying Party Wizard through the Policy node of the AD FS 2.0 Management console. Once there, you or your system administrator needs to specify the appropriate data sources in the Select Data Source page of the wizard, which is shown in **Figure 3**.

The first two options enable you to automatically configure the relying party using federation metadata. If you have access to the relying party's federation metadata on a network or in a local file, select one of these two options because they are less prone to error, they automate the entire process and they auto-update if any details of the relying party change in the future. These options are a major improvement over AD FS 1.0, which doesn't offer such automated processes.

The third option requires you to enter all the configuration details manually. Use this option only when you don't have access to federation metadata or you want to control the details of the relying party configuration.

It's instructive to run through the "Enter relying party configuration manually" option just so you can see all the steps that are required to set up a new relying party. In the next few pages of the wizard, you'll be asked to choose a profile—choose AD FS 2.0 Profile if you want to support both browser-based and WCF-based clients or AD FS 1.x Profile if you only need AD FS 1.x interoperability and don't support active (WCF, CardSpace) clients; configure the certificate that is used to encrypt the token so that only the relying party with the corresponding private key can decrypt and use the issued token; and configure the identifier that will be used to identify this relying party in all token issuance requests.

Once you finish the Add Relying Party Wizard, a Rules Editor tool opens. In this tool, you configure claims issuance and transformation rules. **Figure 4** shows the Rules Editor tool configured to issue a token with a single claim whose value will be retrieved from the main attribute store. Notice that the displayName attribute is mapped to the Given Name claim. AD FS 2.0 introduces a new, textual domain-specific language that enables you to define simple rules for deriving the claims issuance and transformation process. Each rule consists of a condition and an action and ends—as in [c] => a;—with a semicolon. The transformation logic is a series of rules that execute sequentially during the token issuance process. In **Figure 4**, the Simple View tab provides a user interface to define these rules. The Advanced View tab lets you author rules directly using the domain-specific language.

This example has illustrated how easy it is to configure a trusted relying party in AD FS 2.0. At this point, when AD FS 2.0 receives a token issuance request, it extracts an identifier from the wire protocol (for example, the appliesTo element in the case of WS-Trust) and uses it to look up a target relying party. Once a relying party is found, the settings specified in the wizard are used to derive the token issuance logic.

Now let's look at how you can use WCF to request a token for this relying party from AD FS 2.0.

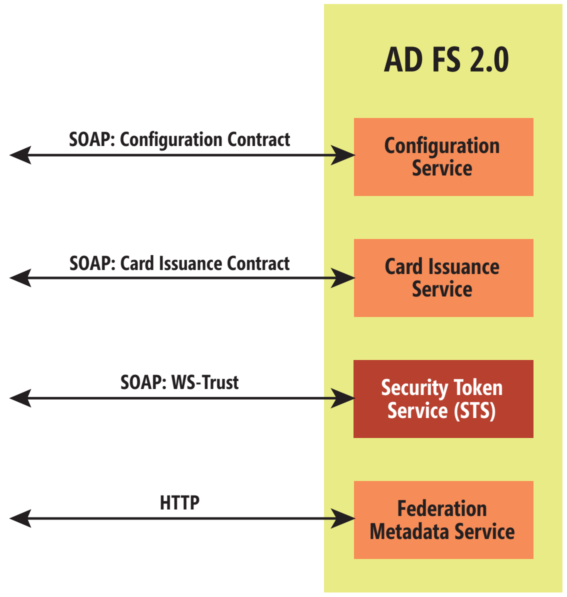

AD FS 2.0
SOAP: Configuration Contract
Configuration Service
SOAP: Card Issuance Contract
Card Issuance Service
SOAP: WS-Trust
Security Token Service (STS)
HTTP
Federation Metadata Service

Figure 2 **Components of AD FS 2.0**

Requesting a Token Using WCF

There are two options for interacting with AD FS 2.0 STS using WCF:

- Explicitly acquire a token acting as a WS-Trust client
- Configure a WCF client so that the infrastructure implicitly acquires a token as part of the call

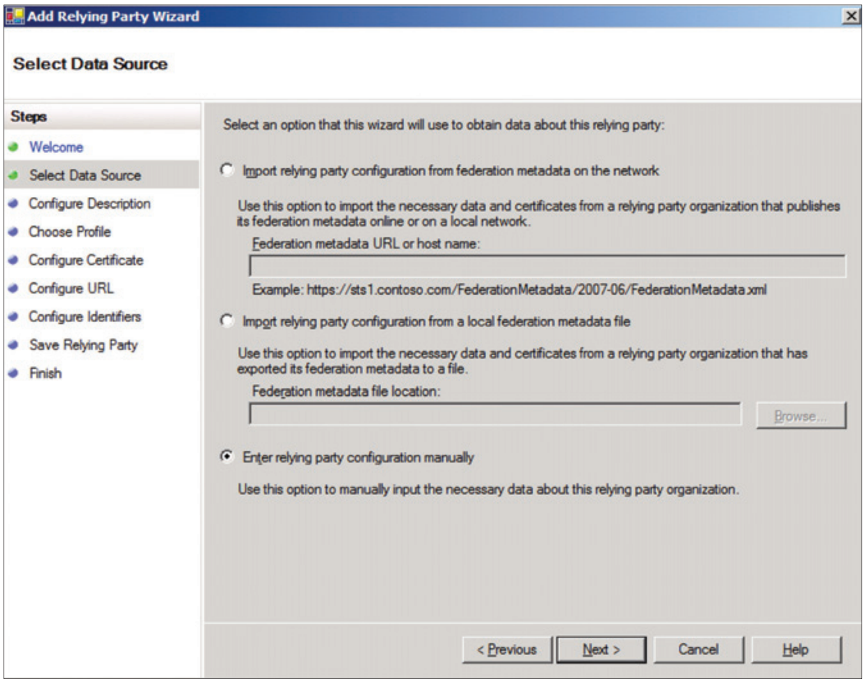
In the explicit option, the WSTrustClient class provides a simple and direct API to request tokens from an STS using the WS-Trust protocol, as shown here.

```
string baseUri = "https://bccoss.com/";

WindowsWSTrustBinding binding = new WindowsWSTru
stBinding(SecurityMode.Transport);

WSTrustClient tokenClient = new
WSTrustClient(binding,
    new EndpointAddress(baseUri + "Trust/2005/
WindowsTransport/"),
    TrustVersion.WSTrustFeb2005,
    new ClientCredentials());

//create a token issuance issuance
RequestSecurityToken rst =
    new RequestSecurityToken(WSTrustFeb2005Const
ants.RequestTypes.Issue);
//Relying Party's identifier
rst.AppliesTo = new EndpointAddress("http://
local.zamd.net/");
//call ADFS STS
SecurityToken token = tokenClient.Issue(rst);
```

This code requests a token using Windows Authentication with transport security. As you can see, in explicit mode, you get access to the raw token, which you can use to call services later on. For example, in a smart client application, you might acquire tokens for different services at application startup or login time, save them in a token cache and then use them throughout the lifetime of the application to call different back-end services. Also, in a scenario where many services live in the same logical security boundary, sharing the same certificate, you can use the explicit mode to acquire a single token and then use it when calling all those services.

In a test environment, in which you usually have access to the relying party's private key, you can use the following code to extract a SAML assertion from the returned token.

```
//Private Key certificate of RP (local.zamd.net)
X509Certificate2 rpCertificate = new
X509Certificate2("zamd.net.pfx", "pwd");
string assertion = Util.
ExtractSAMLAssertion(token, rpCertificate);

<saml:Attribute AttributeName="givenname"
AttributeNamespace="http://schemas.xmlsoap.org/
ws/2005/05/identity/claims">
        <saml:AttributeValue>Zulfiqar Ahmed</saml:AttributeValue>
</saml:Attribute>
```


Add Relying Party Wizard

Select Data Source

Steps
- Welcome
- Select Data Source
- Configure Description
- Choose Profile
- Configure Certificate
- Configure URL
- Configure Identifiers
- Save Relying Party
- Finish

Select an option that this wizard will use to obtain data about this relying party:

○ Import relying party configuration from federation metadata on the network

Use this option to import the necessary data and certificates from a relying party organization that publishes its federation metadata online or on a local network.

Federation metadata URL or host name:

Example: https://sts1.contoso.com/FederationMetadata/2007-06/FederationMetadata.xml

○ Import relying party configuration from a local federation metadata file

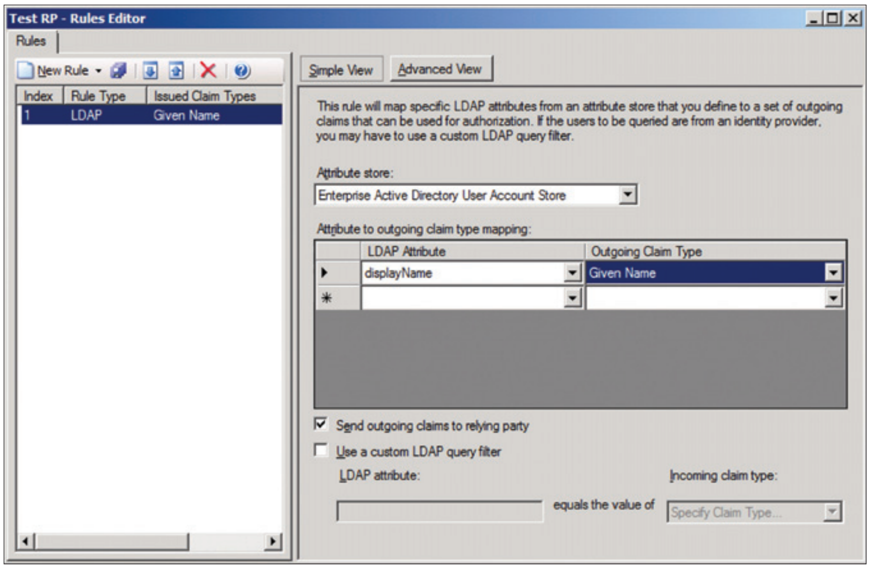
Use this option to import the necessary data and certificates from a relying party organization that has exported its federation metadata to a file.

Federation metadata file location:

● Enter relying party configuration manually

Use this option to manually input the necessary data about this relying party organization.

Figure 3 **Select Data Source Page of Add Relying Part Wizard**


Test RP - Rules Editor

Rules

Index	Rule Type	Issued Claim Types
1	LDAP	Given Name

This rule will map specific LDAP attributes from an attribute store that you define to a set of outgoing claims that can be used for authorization. If the users to be queried are from an identity provider, you may have to use a custom LDAP query filter.

Attribute store: Enterprise Active Directory User Account Store

Attribute to outgoing claim type mapping:

	LDAP Attribute	Outgoing Claim Type
▸	displayName	Given Name
*		

☑ Send outgoing claims to relying party

☐ Use a custom LDAP query filter

LDAP attribute: equals the value of Incoming claim type: Specify Claim Type...

Figure 4 **Rules Editor Tool**

The SAML token contains only the claims configured for this particular relying party. Refer back to **Figure 4**, which shows how this relying party's output token was configured to return a single attribute. You can edit the relying party's configuration to include more claims in the output, and you should see all of them reflected here. You can also use claims policy language directly to define rich transformation and filtering logic.

Both the WSTrustClient API and the new WSTrust bindings are part of WIF, so for the preceding code to work, WIF must be installed on the client. You can also use the WCF API directly to explicitly acquire a token, but the simplicity and ease of use WIF offers can take one task off your to-do list.

In the code in **Figure 5**, IssuedSecurityTokenProvider is the WCF equivalent of WSTrustClient and is normally used by wsFederationBinding when requesting tokens on your behalf. Because it's a public API, you are free to use it in your code should you need access to a raw token. The CustomBinding is equivalent to WindowsWSTrustBinding.

In the implicit option, you can use the standard wsFederationHttpBinding, in which case the WCF infrastructure transparently acquires the token and sends it to the service as part of the call. Whenever you create a new WCF proxy and use it to call a service, the infrastructure fetches a new token for you. Obviously, this would be overkill in some scenarios. The code in **Figure 6** configures a fictional Employee-Service to require tokens from AD FS 2.0.

Mapping AD FS 2.0 Concepts to WCF

The core responsibility of AD FS 2.0 is to issue tokens to authenticated users. Users can be authenticated using different authentication mechanisms (such as Windows Authentication). You can see all the supported authentication mechanisms by selecting the Endpoints node in the management console.

You'll notice two familiar WCF security concepts as column headings within the Endpoints node:

- Authentication Type is the AD FS 2.0 equivalent of the WCF clientCredentialType terminology.
- Security Mode choices are Transport, Message or Mixed. Mixed is the AD FS 2.0 equivalent of WCF's TransportWithMessageCredentials.

Different combinations of these two values are exposed using different endpoints, and you choose a specific endpoint based on your authentication needs. For example, if you need to authenticate using Username/Password, you would choose the Clear Password authentication endpoint.

For AD FS 2.0 STS, mapping these concepts back to Address, Binding and Contract (ABC) in WCF, you get the following equivalents:

- Address = AD FS 2.0 base address + the endpoint's URL Path
- Binding = Endpoint's Security Mode + Authentication Type
- Contract = Standard WS-Trust protocol

Figure 5 **Using IssuedSecurityTokenProvider to Access a Raw Token**

```
sstring baseUri = "https://bccoss.com/";

//Limited edition of WSTrustClient:)
IssuedSecurityTokenProvider provider = new IssuedSecurityTokenProvider();
provider.SecurityTokenSerializer = new WSSecurityTokenSerializer();

//Relying Party's identifier
provider.TargetAddress = new EndpointAddress(new Uri("http://local.zamd.net"));
provider.IssuerAddress = new EndpointAddress(new Uri(baseUri + "Trust/2005/WindowsTransport/"));

provider.SecurityAlgorithmSuite = SecurityAlgorithmSuite.Basic256;
provider.MessageSecurityVersion = MessageSecurityVersion.
WSSecurity10WSTrustFebruary2005WSSecureConversationFebruary2005WSSecurityPolicy11BasicSecurityProfile10;

HttpsTransportBindingElement tbe = new HttpsTransportBindingElement();
tbe.AuthenticationScheme = AuthenticationSchemes.Negotiate;
CustomBinding stsBinding = new CustomBinding(tbe);

provider.IssuerBinding = stsBinding;
provider.Open();
//Request a token from ADFS STS
SecurityToken issuedToken = provider.GetToken(TimeSpan.FromSeconds(30));
```

Figure 6 **Using wsFederationHttpBindingto Acquire a Token Implicitly**

```
<system.serviceModel>
   <services>
      <service name="EmployeeService.EmployeeService">
         <endpoint address="http://localhost:9990/ES"
            binding="ws2007FederationHttpBinding"
            contract="EmployeeServiceContract.IEmployeeService"
            bindingConfiguration="adfsFed"/>
      </service>
   </services>
   <bindings>
      <ws2007FederationHttpBinding>
         <binding name="adfsFed">
            <security mode="Message">
               <message negotiateServiceCredential="false" >
                  <issuer address="https://bccoss.com/Trust13/KerberosMixed"
                          binding="customBinding" bindingConfiguration="MixedKerberos"/>
               </message>
            </security>
         </binding>
      </ws2007FederationHttpBinding>
      <customBinding>
         <binding name="MixedKerberos">
            <security authenticationMode="KerberosOverTransport"/>
            <httpsTransport/>
         </binding>
      </customBinding>
   </bindings>
</system.serviceModel>
```

Federating AD FS 2.0 with Another STS

In addition to creating relying parties, you can establish a trust relationship between AD FS 2.0 and your custom STS or another AD FS 2.0. For example, if you already have an STS that authenticates users and issues tokens, you can simply add it as a trusted identity provider inside AD FS 2.0, which will accept tokens issued from the STS.

Setting Up an Identity Provider

Setting up a new, trusted identity provider in AD FS 2.0 is similar to setting up a new relying party. The Add Identity Provider wizard you use looks and acts a lot like the Add Relying Party Wizard (refer back to **Figure 3**).

To get to the Configure Identifier page, select the manual configuration option again (as you did in **Figure 3**) and select AD FS 2.0 Profile on the Choose Profile page. Leave the default settings on the Configure URL page. You then choose an identifier and a public key certificate for your identity provider and finish the wizard to register the new identity provider.

Requesting a Token Using WCF

Once you register an additional identity provider with AD FS 2.0, the logical architecture looks like the configuration shown in **Figure 7**.

The code in **Figure 8** lets you acquire a token explicitly, giving you the flexibility to cache the token locally and send it to the service as needed.

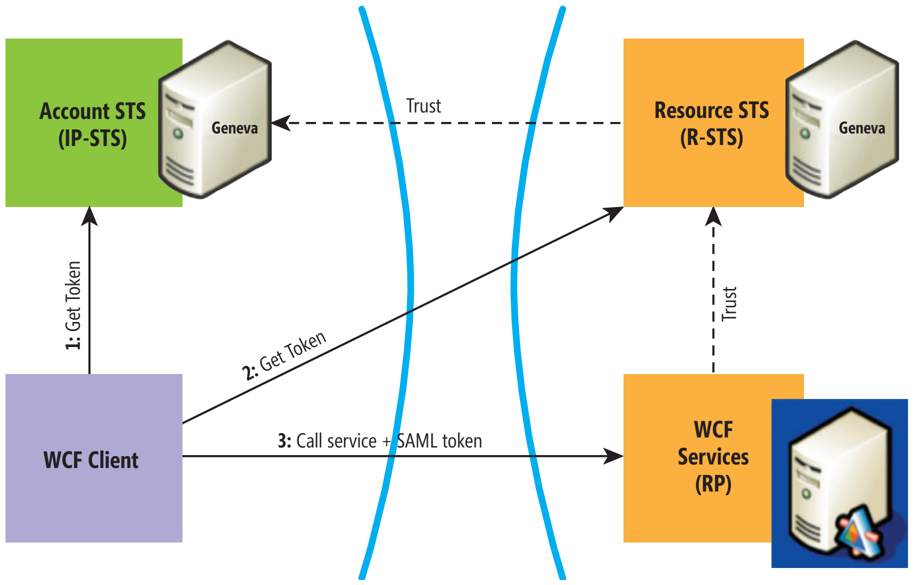


Figure 7 **Architecture of AD FS 2.0 with an Additional Identity Provider**

IssuedTokenWSTrustBinding is very similar to wsFederationHttpBinding in that it hides all the complexity of intermediate tokens by transparently talking to the IP-STS to acquire an intermediate token that is then sent to R-STS as an authentication token.

The code in **Figure 9** uses wsFederationHttpBinding to enable a WCF client to implicitly acquire a token as part of a service call.

Figure 8 **Using IssuedTokenWSTrustBinding to Acquire a Token Explicitly**

```
string adfsStsUri = "http://bccoss.com/Trust/2005/IssuedTokenAsymmetricBasic256";

//binding for local STS(IP-STS)
WSHttpBinding localStsBinding = new WSHttpBinding(SecurityMode.Message);

localStsBinding.Security.Message.ClientCredentialType = MessageCredentialType.None;
localStsBinding.Security.Message.EstablishSecurityContext = false;
localStsBinding.Security.Message.NegotiateServiceCredential = false;

EndpointAddress localStsEpr = new EndpointAddress(
    new Uri("http://localhost:9000/STS/"),
    new X509CertificateEndpointIdentity(new X509Certificate2(@"MyCustomSTSPublicKey.cer")));

//This binding will transparently acquire all the intermediate tokens as part of the call. (R-STS)
IssuedTokenWSTrustBinding fedBinding = new IssuedTokenWSTrustBinding(localStsBinding, localStsEpr);
fedBinding.TrustVersion = TrustVersion.WSTrustFeb2005;


EndpointAddress adfsStsEpr = new EndpointAddress(
    new Uri(adfsStsUri),
    new X509CertificateEndpointIdentity(new X509Certificate2("AdfsStsPubicKeyOnly.cer")));

WSTrustClient trustClient = new WSTrustClient(fedBinding, adfsStsEpr, TrustVersion.WSTrustFeb2005,
  null);

//Create a security token request
RequestSecurityToken rst = new RequestSecurityToken(RequestTypeConstants.Issue);
//Set Relying Party's identifier accordingly
rst.AppliesTo = new EndpointAddress("http://local.zamd.net");

SecurityToken finalToken = trustClient.Issue(rst);
```

Notice that I'm using a customBinding when talking to the /IssuedToken-MixedSymmetricBasic256 endpoint. The standard wsFederationHttpBinding doesn't work here because it tries to establish a secure session, which is incompatible with this AD FS 2.0 endpoint. To federate WCF clients with AD FS 2.0, you have to use either a customBinding or one of the new WS-Trust-based bindings that ships with WIF.

AD FS 2.0 and Browser Clients

AD FS 2.0 has first-class support for Web single sign-on (WebSSO) and federation using both WS-Federation and SAML 2.0 protocols.

Conceptually, WS-Federation and the SAML protocol are similar even though they have different wire representations. The WS-Federation wire format is closely related to WS-Trust protocol, so it is the logical choice when you're serving both active and passive (browser-based) clients. The SAML protocol has better interoperability across different vendors. AD FS 2.0 natively supports both of these protocols. It's a good idea to stick to a single protocol (for example, WS-Federation) inside your security boundary and use AD FS 2.0 as a protocol broker hub for incoming or outgoing SSOs.

Let's consider an example. Say that you have a simple ASP.NET application that provides functionality only to authenticated users. As a stand-alone application, authentication logic is implemented as part of the application, and an interaction with this application would follow the steps shown in **Figure 10**.

Here the usual ASP.NET authentication mechanisms, such as Forms Authentication, are being implemented. Our goal is to extract the authentication functionality from this application and use AD FS 2.0 instead.

In the AD FS 2.0 setup, which is shown in **Figure 11**, the application becomes a trusted relying party inside AD FS 2.0 and therefore trusts tokens issued by AD FS 2.0. The application uses WIF to do all the heavy lifting of token parsing, extracting claims and so on. Identity information is provided to the application using the standard IIdentity/IPrincipal abstractions.

The distributed authentication in AD FS 2.0 is much more flexible than direct authentication, and it provides some major benefits:

- Authentication is externalized from the application, so the authentication mechanism can be changed (for example, from username to Kerberos) without affecting the application.
- The flexibility of a claims-based model can provide all the required information to the application (as part of the token) directly rather than the application itself retrieving that information from different sources.

The Beta 2 release of WIF introduced new project templates that make it easy to externalize an application's authentication logic to an STS. As of this writing, these templates are available only in C#.

Externalizing Authentication Logic

To externalize an application's authentication logic, you use the Microsoft Visual Studio dialog box New Web Site. Select the Claims-aware Web Site template to create a standard ASP.NET Web site that is preconfigured with WIF.

To launch the Federation Utility wizard, shown in **Figure 12**, right-click the Web Site node in Solution Explorer and select Modify STS Reference from the menu.

For this example, let's choose the "Use an existing STS" option and specify AD FS 2.0 as the STS. The wizard requires the URL of the metadata document to automate all the required configurations. The metadata document URL is available as an endpoint inside AD FS 2.0.

Federation metadata contains essential information such as the STS signing certificate, the claims offered and the token issuance URL. Having a standardized format for this information enables

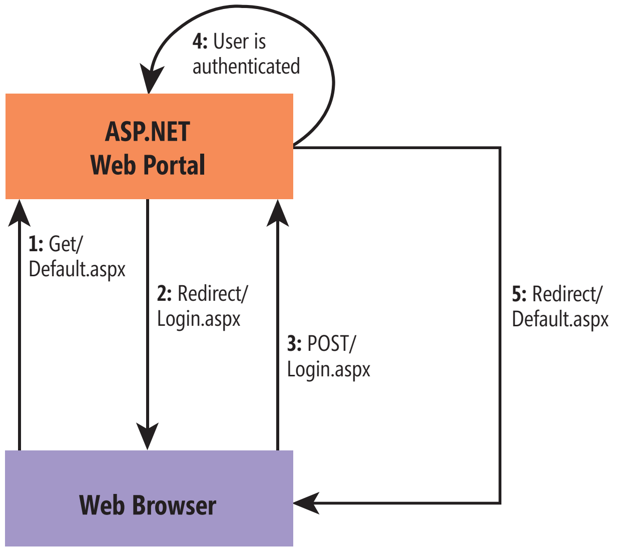


Figure 10 **Direct Authentication in a Simple ASP.NET Application**

tools to automate the establishment of trusts between an STS and relying parties.

The Summary page of the wizard summarizes the changes that are going to be made in the web.config file.

The Federation Utility wizard configures WIF on your Web site to provide the following functionality:

- All unauthenticated requests will be redirected to AD FS 2.0.
- Any request containing a valid token will be processed, and identity information will be presented to the application in the form of ClaimsIdentity/ClaimsPrincipal. The Application

Figure 9 **Using wsFederationHttpBinding to Acquire a Token Implicitly**

```
<system.serivceModel>
   <bindings>
      <wsFederationHttpBinding>
         <binding name="R-STS">
         <security mode="Message">
            <message>
               <issuer address="https://bccoss.com/Trust/2005/IssuedTokenMixedSymmetricBasic256" binding="customBinding" bindingConfiguration="IP-STS"/>
            </message>
            </security>
         </binding>
      </wsFederationHttpBinding>

      <customBinding>
         <binding name="IP-STS">
            <security authenticationMode="IssuedTokenOverTransport">
               <issuedTokenParameters>
                  <issuer address="http://localhost:9000/CustomSTS" binding="wsHttpBinding"/>
               </issuedTokenParameters>
            </security>
            <httpsTransport/>
         </binding>
      </customBinding>
   </bindings>

   <client>
      <endpoint address="http://localhost:9990/ES" binding="wsFederationHttpBinding" bindingConfiguration="R-STS"
                contract="ServiceReference1.IEmployeeService" name="WSFederationHttpBinding_IEmployeeService"/>
   </client>
</system.serviceModel>
```

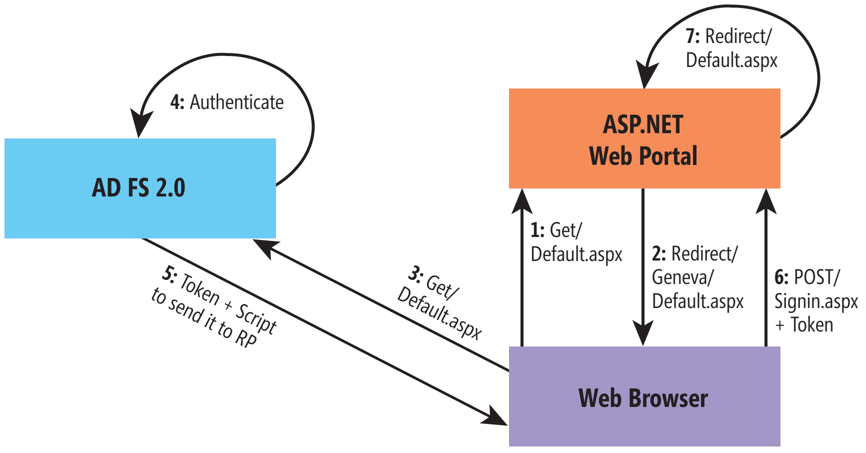

4: Authenticate
AD FS 2.0
5: Token + Script to send it to RP
3: Get/ Default.aspx
7: Redirect/ Default.aspx
ASP.NET Web Portal
1: Get/ Default.aspx
2: Redirect/ Geneva/ Default.aspx
6: POST/ Signin.aspx + Token
Web Browser

Figure 11 Distributed Authentication in AD FS 2.0

will continue to access identity information using the standard IPrincipal/IIdentity abstractions regardless of the source of that information.

Before testing the application, you need to make one last configuration change on AD FS 2.0. You must add an additional endpoint to the relying party for browser clients. This endpoint is required because once AD FS 2.0 has processed a token issuance request, two pieces of information are required before it can send the token back to the browser:

- The address where the token should be sent
- The protocol (SAML or WS-Federation) over which the token should be sent

You can add a passive endpoint to the relying party in the Endpoints tab of the Test RP Properties dialog box. For example, if you select WS-Federation as the Endpoint Type, AD FS 2.0 will send tokens back to the relying party using the WS-Federation protocol. Inside the relying party, the WIF, which natively understands WS-Federation protocol, processes these tokens.

Now when you try to browse to the application, you are automatically redirected to AD FS 2.0 for authentication, where you can choose the authentication method you want to use: Windows Integrated Authentication, Certificate Authentication or Username/Password Form.

Once authentication is successful, you—along with a token issued by AD FS 2.0—are redirected back to the application. WIF processes this token and makes the final identity (in the form of claims) available to the application using the standard ASP.NET mechanisms (for example, Page.User).

Browser-Based Federation

You can extend a basic external authentication scenario into a federation scenario by adding an additional trusted identity provider. The identity provider options are shown during the authentication process.

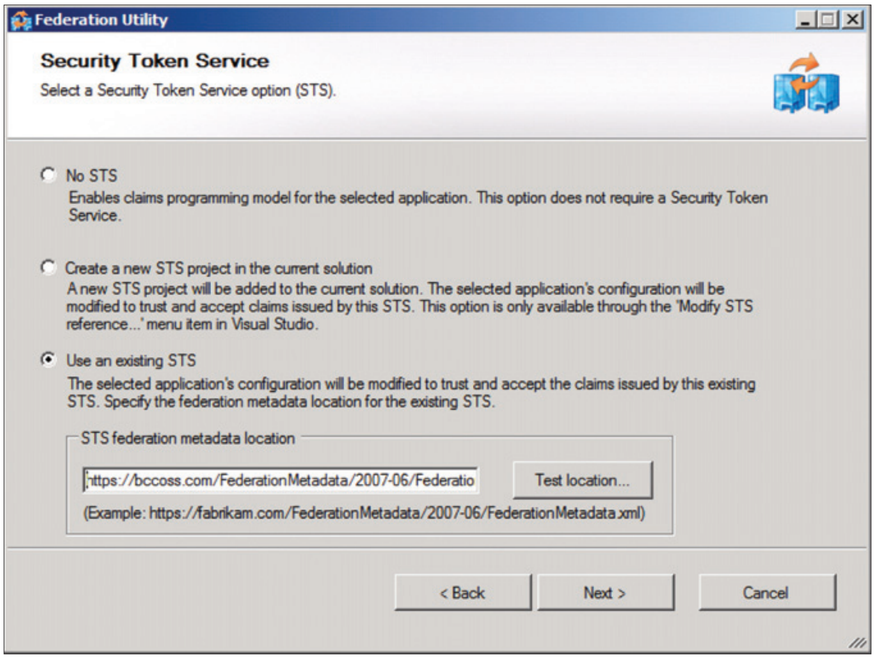
You can authenticate with AD FS 2.0 or another trusted identity provider. If you select a different identity provider, you are redirected to that identity provider and, upon successful authentication, redirected back to AD FS 2.0, which would then authenticate you based on the token issued by the trusted identity provider.

Powerful Combination

As you've seen in this article, AD FS 2.0 STS provides a simply and ready-made solution to claims-enable your WCF services and browser-based applications. STS itself is only one small piece of AD FS 2.0, which also includes a CardSpace provisioning system, a rule-based claims transformation engine, automatic trust management infrastructure, management and configuration services and their respective tools. Along with WIF, AD FS 2.0 creates a powerful combination to program identity solutions on the Windows platform. ■

ZULFIQAR AHMED *is a Senior Consultant on the UK Application Development Consulting (ADC) team and can be reached at http://zamd.net.*

THANKS to the following technical expert for reviewing this article: *Colin Brace*



Security Token Service

Select a Security Token Service option (STS).

○ No STS
Enables claims programming model for the selected application. This option does not require a Security Token Service.

○ Create a new STS project in the current solution
A new STS project will be added to the current solution. The selected application's configuration will be modified to trust and accept claims issued by this STS. This option is only available through the 'Modify STS reference...' menu item in Visual Studio.

● Use an existing STS
The selected application's configuration will be modified to trust and accept the claims issued by this existing STS. Specify the federation metadata location for the existing STS.

STS federation metadata location: https://bccoss.com/FederationMetadata/2007-06/Federatio

(Example: https://fabrikam.com/FederationMetadata/2007-06/FederationMetadata.xml)

Figure 12 **Federation Utility**

Application Guidelines on Digital Signature Practices for Common Criteria Security

Jack Davis

The transition from paper documents with handwritten signatures to electronic files with digital signatures is moving at a rapid pace. To meet user needs and certification requirements, electronic documents with digital signatures need to provide the same functionality and security that manually signed paper documents offer. This article describes how you, as a software developer, can do just that: design applications that have built-in digital signature functionality that meets the requirements for ISO/IEC 15408 Common Criteria security.

In the realm of documents and data files, digital signatures provide two key features:

- Identify the originator (the signer) of the document or file content

This article discusses:

- Best practices for digital signatures
- ISO/IEC Common Criteria security standards
- OOXML and OPC conventions
- Guidelines for signing categories

Technologies discussed:

ISO/IEC, Office Open XML, Microsoft Open XML Format SDK, Microsoft .NET System.IO.Packaging API

- Verify that the signed information hasn't been altered after the signature was applied

Beyond the basic principle of "What You See Is What You Sign" (WYSIWYS), you and your UI designers need to be aware of several additional security and user considerations when building applications that support digital signatures. You also need to understand the security issues that can affect your products in meeting certification requirements. Certifications, such as those for ISO/IEC 15408 Common Criteria security, provide organizations and users a reliable way of identifying products that have been tested to meet usability and security standards. Increasingly, independent security certification is becoming a prerequisite in purchase decisions.

By the end of this article, you'll better understand the range of issues associated with digital signatures and the inherent need for clear, accurate and open disclosure to users.

ISO/IEC 15408 Common Criteria Security Certification

Supported by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), Common Criteria is a suite of methodologies designed to assess and certify the security capabilities of information technology products. Common Criteria security techniques are published in ISO/IEC standards 15408 and 18045.

As of the publication date of this article, the Common Criteria security standard has been approved and adopted by 26 countries. You can find a list of Common Criteria members at commoncriteriaportal.org/members.html.

Office Open XML and Open Packaging Conventions

In 2008, ISO/IEC approved and adopted Office Open XML (OOXML) as an open international standard. The Open Packaging Conventions (OPC) component of OOXML provides an industry standard container-file technology. OPC combines the use of ZIP and XML to enable applications to define file formats that are open and easily accessible. OPC also defines functionality to support digital signatures, metadata properties and content relationships. Access to both application data and OPC elements is provided through a common set of APIs.

Microsoft Word (.docx), Excel (.xlsx) and PowerPoint (.pptx) and the XML Paper Specification (.xps) are examples of OPC-based file formats, and each employs its own unique content organization (schema) that is associated with a specific application. Each OPC file format can also define its own signing policy for digital signatures. A file format signing policy specifies what content items must be signed, what items can optionally be signed and what items shouldn't be signed as part of the digital signature signing or validation process.

In this article, you'll learn more about how to validate and display signature information stored with digitally signed electronic documents. These design considerations are based on the digital signature functionality provided by the OPC component of the ISO/IEC 29500 and ECMA 376 Office Open XML standards. Keep in mind that the digital signature signing policies of specific OPC-based file formats might employ only a subset of the signing options described in this article.

Digital Signatures

OPC file formats, such as those described in OOXML, use X.509 certificates and XML Digital Signature (W3C, 2008) technologies to provide two elements of security:

- Identity repudiation securely identifies the individual or group originating a signature.
- Content validation securely ensures that all signed content is present and hasn't been modified in any way after it was signed.

An alteration to any part of a digital signature or to any content signed by the signature will be detected during signature validation, and the user will be notified through an error message that the signature is broken.

Some of the figures in this article are sample notification or status messages for the user. Whether they confirm success or failure of a signature validation or highlight special features that the user should be aware of, they represent the kind of clear and specific wording you should use. Your application defines when and where the user sees these digital signature status messages. Digital signature validation is sometimes scheduled to perform automatically whenever a signed document is opened. In this case, the message would be displayed when a user opens the document. If the user is required to perform the digital signature validation manually through a menu selection, you might want to present the signature status in a pop-up dialog box. You and your UI designer have some leeway in the timing and placement of these user messages—just remember to make them as intuitive and easy to understand as possible.

Getting your application to report success or failure when verifying the integrity of the signature and the signed content is only the first step in writing a digitally secure program.

Signing Categories

Getting your application to report success or failure when verifying the integrity of the signature and the signed content is only the first step in writing a digitally secure program. You also need to consider the various types of content your application will encounter.

In the simplest form of signing, called comprehensive signing, a user applies his or her signature to all the content in a single document. The digital signature and signed content are later verified and then reported as either valid or invalid. **Figure 1** is an example of the kind of digital signature status message a user would see.

Document elements in four additional categories also need to be considered to avoid confusion or mistaken trust assumptions:

- Unsigned content
- Signed content groups
- Externally referenced content
- Dynamic content

Comprehensive Signing

Most people are very familiar with signing paper documents and contracts. Individuals initial each page or sign at the end of a document to identify themselves and acknowledge their review, approval and acceptance.

Figure 1 **Sample User Message for Comprehensive Signing**

This document and all items contained in the document file are signed.
All signatures are valid. Click here to view the signer's identity.

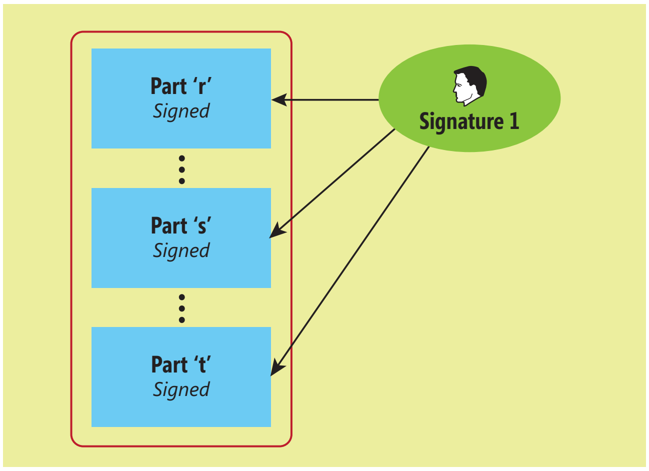


Figure 2 **Single-Signature Signed Content**

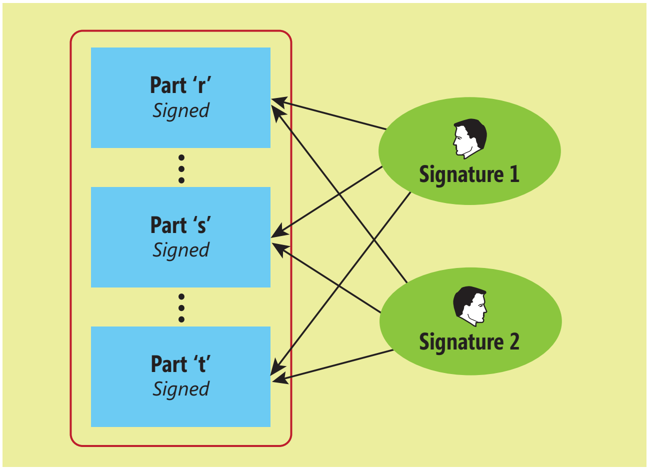


Figure 3 **Multiple-Signature Signed Content**

As mentioned earlier, the simplest and most straightforward signing case in a digital document is comprehensive signing. To be classified as this type of signature, all content associated with the document file must meet the following four criteria:

- All content items are located in a single document package.
- All content items are static.
- There are no links or references to external content.
- All content items in the document package are signed.

For a document to be fully static, content elements can't contain any dynamically alterable items such as text created through macros or input from external sources. When all content elements in a static document file are signed, both the digital signature and the signed content can be verified with a summary result reporting the signature and associated signed content as either valid or invalid.

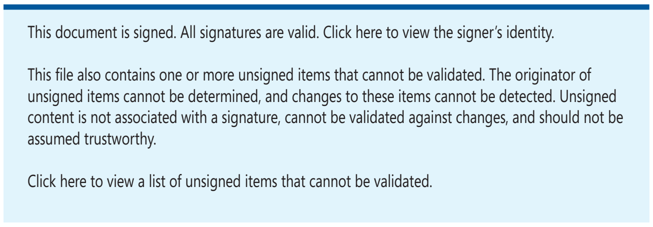
In **Figure 2**, Signature 1 signs document parts *r*, *s* and *t*. A later validation performed on the signature will verify that Signature 1 is still intact and valid, and that each of its associated signed parts (*r*, *s* and *t*) hasn't been altered after being signed.

Figure 3 illustrates a variation in signing, in which two (or more) signatures are applied to a document. Multiple signatures commonly occur in a review and approval cycle that requires individuals to indicate their approval and acceptance of a document after reviewing it. In this form, each signature can be verified independently along with the associated signed document parts.

You can use the following method and property to verify signatures and identify each signer in a signature and signed content validation scenario.

- The PackageDigitalSignatureManager.VerifySignatures method verifies that the signatures are valid and the signed parts within the package haven't been modified.
- The PackageDigitalSignature.Signer property returns a copy of the X.509 certificate that identifies the signer.

Figure 4 **Sample User Message for Unsigned Content**


This document is signed. All signatures are valid. Click here to view the signer's identity.

This file also contains one or more unsigned items that cannot be validated. The originator of unsigned items cannot be determined, and changes to these items cannot be detected. Unsigned content is not associated with a signature, cannot be validated against changes, and should not be assumed trustworthy.

Click here to view a list of unsigned items that cannot be validated.

Unsigned Content

Unsigned content is a potential risk to the implied trust association. With paper documents, people commonly sign or initial pages to identify themselves and acknowledge their review and approval. Any pages left unsigned (or added later as unsigned pages) have no formal association or legal weight with the signer.

In digital documents, the validation of a digital signature and its related content verifies that the user can trust both the signature and the associated signed content. Unfortunately, many nontechnical users don't have the expertise to differentiate between digital signatures, signed content, unsigned content and the container in which the document components are packaged. If only a positive validation of a digital signature and its signed content is displayed, users might incorrectly assume that the validation extends to the entire package, which could include unsigned content that shouldn't be associated with a trusted signature. Unsigned content has no identifiable source, and it can be added after other components are signed or modified within the document package.

Signature trust associations need to be clear to all users. Users should be made aware of unsigned content included within any package and provided with a means to clearly distinguish it from trusted signed content. **Figure 4** shows the kind of message users should receive to alert them to unsigned content within a valid digitally signed document.

In **Figure 5**, document parts *m* and *n* are signed and can be validated with Signature 1. The same document package also contains unsigned parts *x* and *y* that can't be validated. Parts *x* and *y* are independent and not associated, even implicitly, with Signature 1.

You can use the following properties, method and pseudocode to identify packages that contain unsigned parts:

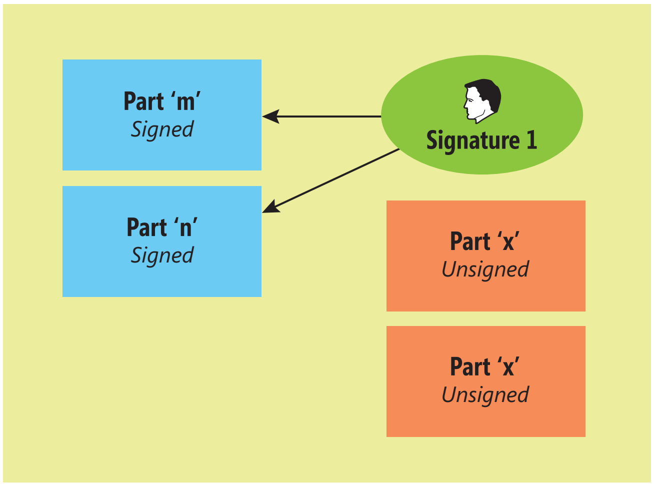


Figure 5 **Signed Document That Includes Unsigned Parts**

- The PackageDigitalSignature.Signer property returns a copy of the X.509 certificate that identifies the signer.
- The Package.GetParts method returns a collection of all the parts contained in the package.
- The PackageDigitalSignature.SignedParts property returns a collection of all the parts signed with a given signature.

The following pseudocode determines whether any parts in a document package aren't signed:

```
PackageDigitalSignatureManager dsm = new PackageDigitalSignatureManager
(package);
PackagePartCollection partCollection = Package.GetParts();
foreach (PackagePart part in partCollection)
    foreach (PackageDigitalSignature signature in dsm.Signatures)
    {
        // Search if "part" is included in signature.SignedParts.
        // If "part" is included in signature.SignedParts
        //     then it is a signed part
        //     else it is an unsigned part
    }
```

Signed Content Groups

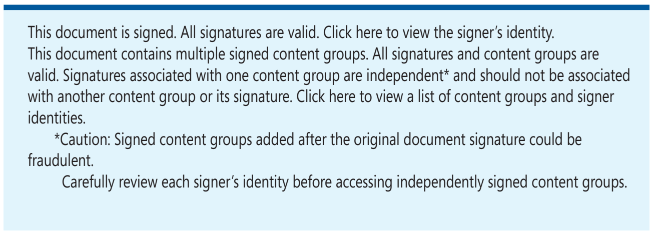
A document with multiple signed content groups is another risk for implied trust by association. With paper documents, one individual can sign pages in one section and a second person can sign pages in a separate section. The signature and pages the first individual signs have no formal association with the pages the second individual signs. Likewise, the signature and pages the second individual signs have no formal association with the pages the first individual signs. Only pages cosigned by both individuals represent a joint association. An example of signed content groups is a document in which the body is written and signed by one individual and appendix sections added for reference are written and signed by other individuals.

In some file formats, different content elements could be signed by different individuals. Users need to know that when signatures relate to different content groups they are independent of one another and that each signature and content group must be validated separately. **Figure 6** shows a user message that could accompany a document with signed content groups.

In **Figure 7**, document parts *r*, *s* and *t* are signed and validated with Signature 1, and document parts *x*, *y* and *z* are signed and validated with Signature 2. The two signatures and their related content groups are independent and not associated with each other even though they are contained in the same document.

In a variation of **Figure 7**, **Figure 8** illustrates a situation in which one or more document parts are cosigned with two or more signatures. While Signature 1 and Signature 2 jointly sign one or more common items (Group C), the signature relationship for Signature 1 with Group A and Signature 2 with Group B remains independent and unassociated with the other. In describing signature information to users, you must explicitly and clearly explain the signature associations of independent and jointly signed content.

Figure 6 **Sample User Message for Signed Content Groups**


This document is signed. All signatures are valid. Click here to view the signer's identity.
This document contains multiple signed content groups. All signatures and content groups are valid. Signatures associated with one content group are independent* and should not be associated with another content group or its signature. Click here to view a list of content groups and signer identities.
*Caution: Signed content groups added after the original document signature could be fraudulent.
Carefully review each signer's identity before accessing independently signed content groups.

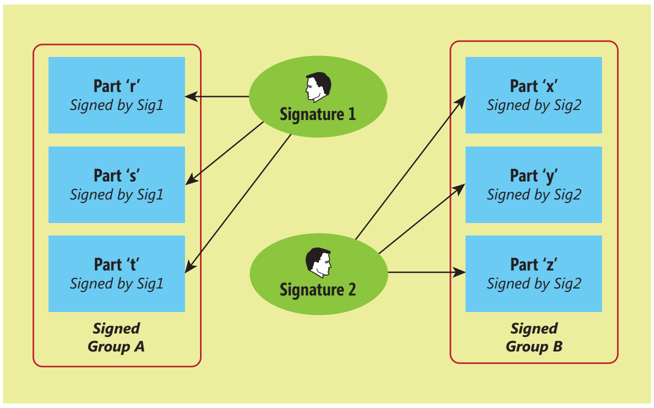


Figure 7 **Two Signed Groups**

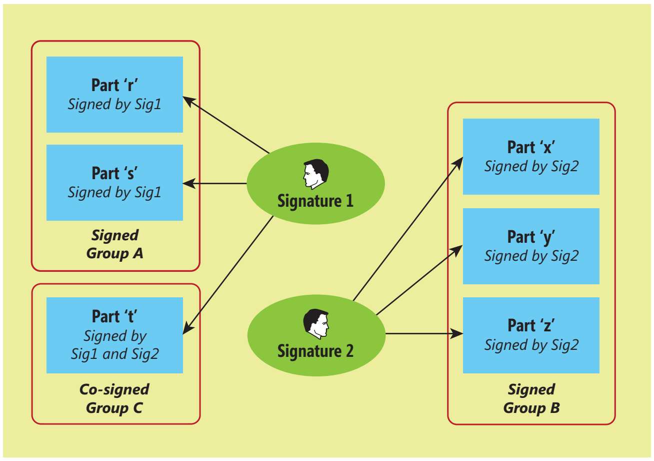


Figure 8 **Signed and Cosigned Groups**

Figure 9 **Sample User Message for Externally Referenced Content**

This document is signed. All signatures are valid. Click here to view the signer's identity.
This document contains references or links to external materials that are unsigned, that are not associated with a signature, and that cannot be validated for changes or modifications. Click here to view a list of unsigned externally referenced content.

The following properties and pseudocode identify packages that contain multiple signed content groups:

- The PackageDigitalSignature.Signer property returns a copy of the X.509 certificate that identifies the signer.
- The PackageDigitalSignature.SignaturePart property returns the name of the part that contains the signature.

The following pseudocode creates a list of the signed content groups contained in a package:

```
PackageDigitalSignatureManager dsm =
   new PackageDigitalSignatureManager(package);
foreach (PackageDigitalSignature signature in dsm.Signatures)
{
    // Add to list(signature.SignaturePart, signature.Signer);
}
// Upon completion the list will contain one entry for each signature
// along with the X.509 certificate that identifies the signer.
// If the list contains more than one entry and different signers,
// the package contains multiple signed content groups.
```

SECURING AGAINST MULTIPLE SIGNED CONTENT GROUPS

Multiple signed content groups add complexity to a file format signing policy given that they increase the possibility that someone could add fraudulently signed content to a document with the intent to create a misleading association to an existing signature and content. To secure your application against this threat, you can require a user's signature to sign the package's signature-origin-relationships part. (The signature-origin-relationships part is a special file in the package with the name \package\services\digital-signatures_rels\origin.psdor.rels.) When both the content and the signature-origin-relationships part are signed, any signatures added later will modify the signature-origin-relationships part, causing the original signature to fail and report the package as invalid.

Externally Referenced Content

Externally referenced content is another potentially improper trust by association scenario. Items within a signed document can contain references to separate, external documents. Unless these other documents are also signed, external references are considered informational and have no formal association with the signed document.

Content accessed through external links and references outside of the document package can change at any time. Ideally, the content being signed shouldn't contain any links or references to materials outside the document package. Users must be able to clearly identify and understand situations that could involve signing and validating content that includes links and references to external materials. **Figure 9** is a sample message indicating externally referenced content within a document.

When validating signed content that includes links to external materials, use the following guidelines:

- Alert users to the presence of references and links to external materials. The notification should clarify that external materials are unsigned and not associated with any signature or signature trust.
- Provide users with the means to clearly distinguish between signed static content and unsigned external material. For example, when users are validating and viewing signed content, only internal links to other signed content should be shown—external links and references to external content should be hidden.

In **Figure 10**, document parts *r*, *s* and *t* are signed and can be validated with Signature 1. Document part *s*, however, contains a link to an internal signed reference for Part *t* and links to two external references for unsigned Content *x* and Content *y*. When validating and describing signature information to users, your application should explicitly and clearly notify the user that links *x* and *y* refer to unsigned content that isn't associated with Signature 1 and is unrelated to other validated content elements (parts *r*, *s* and *t*).

Identifying items that refer to external content is dependent on the schema of the specific OPC file format. In the online material that accompanies this article, which is at msdn.microsoft.com/magazine/dd890995, you can find methods, properties and pseudocode that you can use to identify externally referenced content in Microsoft Office file formats for Word (.docx), Excel (.xlsx) and PowerPoint (.pptx).

Dynamic Content

Dynamic unsigned content is also a potential improper trust by association situation. Dynamic alterable content has no analogy in paper documents. Changes or alterations made to a paper document after it is signed legally invalidate the signature. To bear

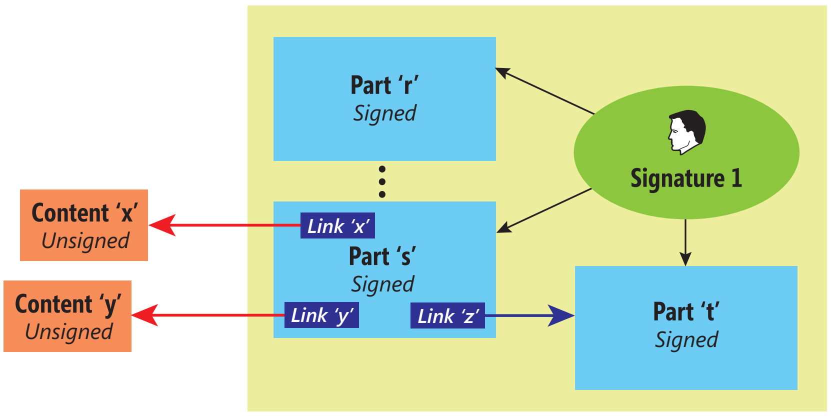


Figure 10 **Externally Referenced Content**

Figure 11 Sample User Message for Dynamic Content

This document is signed. All signatures are valid. Click here to view the signer's identity.
This document contains content items that are dynamically computed. Dynamic content can change independently and is not related to any signatures or other signed content contained in this document. Click here to view a list of dynamic content elements.

legal weight, modifications to a signed paper document must be reviewed, acknowledged and re-signed by the original signators.

From a signature-trust standpoint, content that can be dynamically added, removed or altered is by its very nature unsignable. Dynamic content, such as text created through the execution of functions and macros, can violate the fundamental principle of WYSIWYS. Dynamic content is commonly produced from functions that insert text for variables such as "today's date" and "last saved date," formulas such as Sum, Reference fields or other content created when custom macros are executed.

Dynamic content created as a result of running custom macros can be used maliciously, and it presents a security vulnerability. Executable operations add complexity that makes dynamic content difficult for users to clearly understand. To enforce the WYSIWYS principle, documents that need to be signed shouldn't include dynamic content and should instead use static content only.

Situations that involve signing and validating documents that contain dynamic content must be openly, clearly and accurately represented to users. When signing a document that contains dynamic content is unavoidable, adhere to the following guidelines:

- Alert users to the presence of dynamic content. The notification should clarify that the dynamic content is unsigned and unrelated to any signature, and can't be associated with any level of signature trust.
- Provide users with the means to clearly identify and distinguish between signed static content and unsigned dynamic content. For example, when validating and viewing signed content, dynamic content should be hidden; the signer should see only the static content originally displayed.

Figure 11 shows a message that warns the user of dynamically computed content.

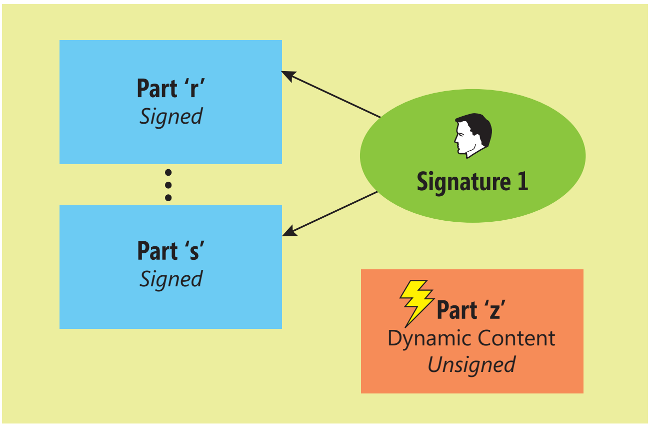


Figure 12 **Dynamic Content**

In **Figure 12**, document parts *r* and *s* are signed and can be validated with Signature 1. However, the document package also contains a piece of dynamic content, Part *z*, that can't be signed or validated. Part *z* is independent and not associated, even implicitly, with Signature 1.

Identifying items that contain dynamic content is dependent on the schema of the specific OPC file format. In the online material that accompanies this article, which is at msdn.microsoft.com/magazine/dd890995, you can find methods, properties and pseudocode that you can use to identify dynamic content in Microsoft Office file formats for Word (.docx), Excel (.xlsx) and PowerPoint (.pptx).

Maintaining Transparency

I'll say it again: In building applications that support digital signatures, you need to ensure that users are given the same functionality and transparency they would have with a handwritten signature. In addition to being aware of the basic operations of validating signature certificates (X.509) and signed content (cryptographic hashes), you and your UI designers need to be watchful for any situations that might not be intuitively clear to the user.

To summarize, potential issues could arise from the following culprits:

- Presence and identification of unsigned content
- Associations between signers of multiple content groups
- Presence and identification of unsigned externally referenced material.

Presence and identification of unsigned dynamic content.

If you don't exclude such types of content from your application by design, you must openly and accurately present them to the user. Again, the clear and accurate disclosure of digital signature information is a fundamental user need and a requirement for ISO/IEC 15408 Common Criteria security certification.

In the shift to electronic documents, security-conscious groups and individuals will increasingly rely on independent certifications, such as ISO/IEC 15408 Common Criteria. To ensure that your product is positioned to succeed in today's market, take the time to write a software application that meets the user and security certification standards.

For more information on and links to various standards, go to the online material that accompanies this article, at msdn.microsoft.com/magazine/dd890995. You'll also find an overview of how to identify dynamic and externally referenced content in Microsoft Office documents. ■

JACK DAVIS *is program manager for the Windows OPC "Packaging" team. Davis is a prior contributor to* MSDN Magazine *("OPC: A New Standard for Packaging Your Data," August 2007) and blogs on the Microsoft Packaging team's Web site at blogs.msdn.com/opc. He can be reached at jack.davis@microsoft.com.*

THANKS to the following technical experts for reviewing this article: *Miguel Bañón, Andree Gagnon (LCA), Zeyad Rajabi, Jose Emilio Rico, Sarjana Sheth, Miriam Wallace and Bob Watson.*

Building *N*-Tier Apps with EF4

Daniel Simmons

This article is the third in a series about *n*-tier programming with the Entity Framework (see msdn.microsoft.com/magazine/dd882522.aspx and msdn.microsoft.com/magazine/ee321569.aspx), specifically about building custom Web services with the Entity Framework (EF) and Windows Communication Foundation (WCF). (In some situations, a REST-based service or some other approach is appropriate, but in these articles, I've focused on custom Web services.) The first article described a number of important design considerations and antipatterns. In the second article, I wrote about four patterns that can be used successfully in an *n*-tier application. That article also included code samples that illustrate how the first release of the Entity Framework (EF 3.5 SP1) can be used to implement what I call the Simple Entities pattern. In this article, I'll look at some features coming in the second release of the Entity Framework (EF4) and how you use them to implement the Self-Tracking Entities and Data Transfer Objects (DTOs) *n*-tier patterns.

While Simple Entities is usually not the preferred pattern for *n*-tier applications, it is the most viable option in the first release of the EF. EF4, however, significantly changes the options for *n*-tier programming with the framework. Some of the key new features include the following:

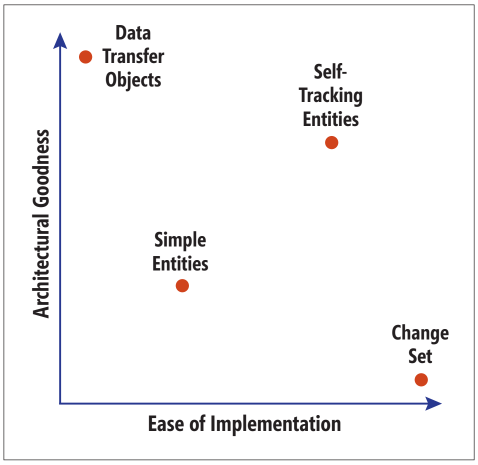


Figure 1 **Comparing *N*-Tier Patterns with EF4**

1. New framework methods that support disconnected operations, such as ChangeObjectState and ChangeRelationshipState, which change an entity or relationship to a new state (added or modified, for example); ApplyOriginalValues, which lets you set the original values for an entity; and the new ObjectMaterialized event, which fires whenever an entity is created by the framework.
2. Support for Plain Old CLR Objects (POCO) and foreign key values on entities. These features let you create entity classes that can be shared between the mid-tier service implementation and other tiers, which may not have the same version of the Entity Framework (.NET 2.0 or Silverlight, for example). POCO objects with foreign keys also have a straightforward serialization format that simplifies interoperability with platforms like Java. The use of foreign keys also enables a much simpler concurrency model for relationships.
3. T4 templates to customize code generation. These templates provide a way to generate classes implementing the Self-Tracking Entities or DTOs patterns.

The Entity Framework team has used these features to implement the Self-Tracking Entities pattern in a template, making that pattern a lot more accessible, and while DTOs still require the most work during initial implementation, this process is also easier with EF4. (The Self-Tracking Entities template and a few other EF features are available as part of a Web download feature community technology preview (CTP) rather than in the Visual Studio 2010/.NET 4 box. The samples in this article assume that Visual Studio 2010/.NET 4 and the feature CTP are installed.) With

Disclaimer: Portions of this article are based on a pre-release version of the Entity Framework. All information is subject to change.

Technologies discussed:

Entity Framework, Windows Communication Foundation

This article discusses:

Entity Framework 4, Self-Tracking Entities pattern, Data Transfer Objects pattern

Code Download URL:

code.msdn.microsoft.com/mag200911EF4

these new capabilities, one way to evaluate the four patterns I've described (Simple Entities, Change Set, Self-Tracking Entities and DTOs) is in terms of a trade-off between architectural goodness (separation of concerns/loose coupling, strength of contract, efficient wire format and interoperability) and ease of implementation and time to market. If you plot the four patterns on a graph that represents this trade-off, the result might look something like **Figure 1**.

The right pattern for a particular situation depends on a lot of factors. In general, DTOs provide many architectural advantages at a high initial implementation cost. Change Set exhibits few good architectural characteristics but is easy to implement (when it's available for a particular technology—for example, the DataSet in traditional ADO.NET).

I recommend a pragmatic/agile balance between these concerns by starting with Self-Tracking Entities and moving to DTOs if the situation warrants it. Often, you can get up and running quickly with Self-Tracking Entities and still achieve many important architectural goals. This approach represents a much better trade-off than Change Set or Simple Entities, either of which I would recommend only if you have no other viable options. DTOs, on the other hand, are definitely the best choice as your application becomes larger and more complex or if you have requirements that can't be met by Self-Tracking Entities, like different rates of change between the client and the server. These two patterns are the most important tools to have in your toolbox, so let's take a look at each of them.

Self-Tracking Entities

To use this pattern with the Entity Framework, start by creating an Entity Data Model that represents your conceptual entities and map it to a database. You can reverse engineer a model from a database you have and customize it, or you can create a model from scratch and then generate a database to match (another new feature in EF4). Once this model and mapping are in place, replace the default code generation template with the Self-Tracking Entities template by right-clicking the entity designer surface and choosing Add Code Generation Item.

Next, choose the Self-Tracking Entities template from the list of installed templates. This step turns off default code generation and adds two templates to your project: one template generates the ObjectContext, and the other template generates entity classes. Separating code generation into two templates makes it possible to split the code into separate assemblies, one for your entity classes and one for your context.

The main advantage of this approach is that you can have your entity classes in an assembly that has no dependencies on the Entity Framework. This way, the entity assembly (or at least the code that it generates) and any business logic you have implemented there can be shared by the mid-tier and the client if you want. The context is kept in an assembly that has dependencies on both the entities and the EF. If the client of your service is running .NET 4, you can just reference the entity assembly from the client project. If your client is running an earlier version of .NET or is running Silverlight, you probably want to add links from the client project to the generated files and recompile the entity source in that project (targeting the appropriate CLR).

Regardless of how you structure your project, the two templates work together to implement the Self-Tracking Entities pattern. The generated entity classes are simple POCO classes whose only feature beyond basic storage of entity properties is to keep track of changes to the entities—the overall state of an entity, changes to critical properties such as concurrency tokens, and changes in relationships between entities. This extra tracking information is part of the DataContract definition for the entities (so when you send an entity to or from a WCF service, the tracking information is carried along).

On the client of the service, changes to the entities are tracked automatically even though the entities are not attached to any context. Each generated entity has code like the following for each property. If you change a property value on an entity with the Unchanged state, for instance, the state is changed to Modified:

```
[DataMember]
public string ContactName
{
    get { return _contactName; }
    set
    {
        if (!Equals(_contactName, value))
        {
            _contactName = value;
            OnPropertyChanged("ContactName");
        }
    }
}
private string _contactName;
```

Similarly, if new entities are added to a graph or entities are deleted from a graph, that information is tracked. Since the state of each entity is tracked on the entity itself, the tracking mechanism behaves as you would expect even when you relate entities retrieved from more than one service call. If you establish a new relationship, just that change is tracked—the entities involved stay in the same state, as though they had all been retrieved from a single service call.

The context template adds a new method, ApplyChanges, to the generated context. ApplyChanges attaches a graph of entities to the context and sets the information in the ObjectStateManager to match the information tracked on the entities. With the information that the entities track about themselves and ApplyChanges, the generated code handles both change tracking and concurrency concerns, two of the most difficult parts of correctly implementing an *n*-tier solution.

As a concrete example, **Figure 2** shows a simple ServiceContract that you could use with Self-Tracking Entities to create an *n*-tier order submission system based on the Northwind sample database.

The GetProducts service method is used to retrieve reference data on the client about the product catalog. This information is usually cached locally and isn't often updated on the client. GetCustomer retrieves a customer and a list of that customer's orders. The implementation of that method is quite simple, as shown here:

```
public Customer GetCustomer(string id)
{
    using (var ctx = new NorthwindEntities())
    {
        return ctx.Customers.Include("Orders")
            .Where(c => c.CustomerID == id)
            .SingleOrDefault();
    }
}
```

This is essentially the same code that you would write for an implementation of this kind of method with the Simple Entities pattern. The difference is that the entities being returned are self-tracking, which means that the client code for using these methods is also quite simple, but it can accomplish much more.

To illustrate, let's assume that in the order submission process you want not only to create an order with appropriate order detail lines but also to update parts of the customer entity with the latest contact information. Further, you want to delete any orders that have a null OrderDate (maybe the system marks rejected orders that way). With the Simple Entities pattern, the combination of adding, modifying and deleting entities in a single graph would require multiple service calls for each type of operation or a very complicated custom contract and service implementation if you tried to implement something like Self-Tracking Entities in the first release of the EF. With EF4, the client code might look like **Figure 3.**

This code creates the service, calls the first two methods on it to get the product list and a customer entity, and then makes changes to the customer entity graph using the same sort of code you would write if you were building a two-tier Entity Framework application that talks directly to the database or were implementing a service on the mid-tier. (If you aren't familiar with this style of creating a WCF service client, it automatically creates a client proxy for you without creating proxies for the entities, since we are reusing the entity classes from the Self-Tracking entities template. You could also use the client generated by the Add Service Reference command in Visual Studio if you want.) But here, there is no ObjectContext involved. You are just manipulating the entities themselves. Finally, the client calls the SubmitOrder service method to push the changes up to the mid-tier.

Of course, in a real application the client's changes to the graph would probably have come from a UI of some sort, and you would add exception handling around the service calls (especially important when you have to communicate over the network), but the code in **Figure 3** illustrates the principles. Another important item to notice is that when you create the order detail entity for the new order, you set just the ProductID property rather than the Product entity itself. This is the new foreign key relationship feature in action. It reduces the amount of information that travels over the wire because you serialize only the ProductID back to the mid-tier, not a copy of the product entity.

Figure 2 **A Simple Service Contract for the Self-Tracking Entities Pattern**

```
[ServiceContract]
public interface INorthwindSTEService
{
    [OperationContract]
    IEnumerable<Product> GetProducts();

    [OperationContract]
    Customer GetCustomer(string id);

    [OperationContract]
    bool SubmitOrder(Order order);

    [OperationContract]
    bool UpdateProduct(Product product);
}
```

Figure 3 **Client Code for the Self-Tracking Entities Pattern**

```
var svc = new ChannelFactory<INorthwindSTEService>(
    "INorthwindSTEService")
    .CreateChannel();

var products = new List<Product>(svc.GetProducts());
var customer = svc.GetCustomer("ALFKI");

customer.ContactName = "Bill Gates";

foreach (var order in customer.Orders
    .Where(o => o.OrderDate == null).ToList())
{
    customer.Orders.Remove(order);
}

var newOrder = new Order();
newOrder.Order_Details.Add(new Order_Detail()
    {
        ProductID = products.Where(p => p.ProductName == "Chai")
                        .Single().ProductID,
        Quantity = 1
    });
customer.Orders.Add(newOrder);

var success = svc.SubmitOrder(newOrder);
```

It's in the implementation of the SubmitOrder service method that Self-Tracking Entities really shines:

```
public bool SubmitOrder(Order newOrder)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Orders.ApplyChanges(newOrder);
        ValidateNewOrderSubmission(ctx, newOrder);
        return ctx.SaveChanges() > 0;
    }
}
```

The call to ApplyChanges performs all the magic. It reads the change information from the entities and applies it to the context in a way that makes the result the same as if those changes had been performed on entities attached to the context the whole time.

Validating Changes

Something else you should notice in the SubmitOrder implementation is the call to ValidateNewOrderSubmission. This method, which I added to the service implementation, examines the ObjectStateManager to make sure that only the kinds of changes we expect in a call to SubmitOrder are present.

This step is really important because by itself, ApplyChanges pushes whatever changes it finds in an entire graph of related objects into the context. Our expectation that the client will only add new orders, update the customer and so on doesn't mean that a buggy (or even malicious) client would not do something else. What if it changed the price on a product to make an order cheaper or more expensive than it should be? The details of how the validation is performed are less important than the critical rule that you should always validate changes before saving them to the database. This rule applies regardless of the *n*-tier pattern you use.

A second critical design principle is that you should develop separate, specific service methods for each operation. Without these separate operations, you do not have a strong contract representing what is and isn't allowed between your two tiers, and properly validating your changes can become impossible. If you had a single SaveEntities service method instead of a SubmitOrder and a separate UpdateProduct method (only accessible by users authorized to modify the product catalog), you could easily implement the apply and save part of that method, but you would be unable to validate properly because you would have no way to know when product updates are allowed and when they are not.

Data Transfer Objects

The Self-Tracking Entities pattern makes the *n*-tier process easy, and if you create specific service methods and validate each one, it can be fairly sound architecturally. Even so, there are limits to what you can do with the pattern. When you run into those limits, DTOs are the way out of the dilemma.

In DTOs, instead of sharing a single entity implementation between the mid-tier and the client, you create a custom object that's used only for transferring data over the service and develop separate entity implementations for the mid-tier and the client. This change provides two main benefits: it isolates your service contract from implementation issues on the mid-tier and the client, allowing that contract to remain stable even if the implementation on the tiers changes, and it allows you to control what data flows over the wire. Therefore, you can avoid sending unnecessary data (or data the client is not allowed to access) or reshape the data to make it more convenient for the service. Generally, the service contract is designed with the client scenarios in mind so that the data can be reshaped between the mid-tier entities and the DTOs (maybe by combining multiple entities into one DTO and skipping properties not needed on the client), while the DTOs can be used directly on the client.

These benefits, however, come at the price of having to create and maintain one or two more layers of objects and mapping. To extend the order submission example, you could create a class just for the purpose of submitting new orders. This class would combine properties of the customer entity with properties from the order that are set in the new order scenario, but the class would leave out properties from both entities that are computed on the mid-tier or set at some other stage in the process. This makes the DTO as small and efficient as possible. The implementation might look like this:

```
public class NewOrderDTO
{
    public string CustomerID { get; set; }
    public string ContactName { get; set; }
    public byte[] CustomerVersion { get; set; }
    public List<NewOrderLine> Lines { get; set; }
}

public class NewOrderLine
{
    public int ProductID { get; set; }
    public short Quantity { get; set; }
}
```

Okay, this is really two classes—one for the order and one for the order detail lines—but the data size is kept as small as possible. The only seemingly extraneous information in the code is the CustomerVersion field, which contains the row version information used for concurrency checks on the customer entity. You need this information for the customer entity because the entity already exists in the database. For the order and detail lines, those are new entities being submitted to the database, so their version information and the OrderID aren't needed—they are generated by the database when the changes are persisted.

The service method that accepts this DTO uses the same lower-level Entity Framework APIs that the Self-Tracking Entities template uses to accomplish its tasks, but now you need to call those APIs directly rather than let the generated code call them for you. The implementation comes in two parts. First, you create a graph of customer, order and order detail entities based on the information in the DTO (see **Figure 4**).

Then you attach the graph to the context and set the appropriate state information:

```
ctx.Customers.Attach(customer);
var customerEntry = ctx.ObjectStateManager.GetObjectStateEntry(customer);
customerEntry.SetModified();
customerEntry.SetModifiedProperty("ContactName");

ctx.ObjectStateManager.ChangeObjectState(order, EntityState.Added);
foreach (var order_detail in order.Order_Details)
{
    ctx.ObjectStateManager.ChangeObjectState(order_detail,
       EntityState.Added);
}

return ctx.SaveChanges() > 0;
```

The first line attaches the entire graph to the context, but when this occurs, each entity is in the Unchanged state, so first you tell the ObjectStateManager to put the customer entity in the Modified state, but with only one property, ContactName, marked as modified. This is important because you don't actually have all the customer information—just the information that was in the DTO. If you marked all properties as modified, the Entity Framework would try to persist a bunch of nulls and zeroes to other fields in the customer entity.

Next you change the state of the order and each of its order details to Added, and then you call SaveChanges.

Hey, where's the validation code? In this case, because you have a very specific DTO for your scenario, and you are interpreting that object as you map the information from it into your entities, you perform the validation as you go along. There's no way this code could inadvertently change the price of a product because you never touch the product entity. This is another benefit of the DTO pattern, but only in a roundabout way. You still have to do the validation work; the pattern just forces one level of validation. In many cases, your code needs to include additional validation of the values or other business rules.

One other consideration is properly handling concurrency exceptions. As I mentioned earlier, the version information for the customer entity is included in the DTO, so you are set up to properly detect concurrency issues if someone else modifies the same customer. A more complete sample would either map this exception to a WCF fault so that the client could resolve the conflict, or it would catch the exception and apply some sort of automatic policy for handling the conflict.

If you wanted to extend the sample further by adding another operation, like the ability to modify an order, you would create another

DTO specifically for that scenario, with just the right information for it. This object would look something like our NewOrderDTO, but it would have the OrderID and Version properties for the order and order details entities as well as each property you want to allow the service call to update. The service method implementation would also be similar to the SubmitOrderDTO method shown earlier—walking through the DTO data, creating corresponding entity objects and then setting their state in the state manager before saving the changes to the database.

If you were to implement the order update method both with Self-Tracking Entities and Data Transfer Objects, you would find that the Self-Tracking Entities implementation reuses the entities and shares almost all the same service implementation code between it and the new order submission method—the only difference would be the validation code, and even some of that might be shared. The DTO implementation, however, requires a separate Data Transfer Object class for each of the two service methods, and the method implementations follow similar patterns but have very little if any code that can be shared.

Figure 4 **Creating a Graph of Entities**

```
var customer = new Customer
    {
        CustomerID = newOrderDTO.CustomerID,
        ContactName = newOrderDTO.ContactName,
        Version = newOrderDTO.CustomerVersion,
    };

var order = new Order
    {
        Customer = customer,
    };

foreach (var line in newOrderDTO.Lines)
{
    order.Order_Details.Add(new Order_Detail
        {
            ProductID = line.ProductID,
            Quantity = line.Quantity,
        });
}
```

Tips from the Trenches

Here are some tips for what to watch out for and understand.

- Make certain to reuse the Self-Tracking Entity template's generated entity code on your client. If you use proxy code generated by Add Service Reference in Visual Studio or some other tool, things look right for the most part, but you will discover that the entities don't actually keep track of their changes on the client.
- Create a new ObjectContext instance in a Using statement for each service method so that it is disposed of before the method returns. This step is critical for scalability of your service. It makes sure that database connections are not kept open across service calls and that temporary state used by a particular operation is garbage collected when that operation is over. The Entity Framework automatically caches metadata and other information it needs in the app domain, and ADO.NET pools database connections, so re-creating the context each time is a quick operation.
- Use the new foreign key relationships feature whenever possible. It makes changing relationships between entities much easier. With independent associations—the only type of relationship available in the first release of the Entity Framework—concurrency checks are performed on relationships independently of the concurrency checks performed on entities, and there is no way to opt out of these relationship concurrency checks. The result is that your services must carry the original values of relationships and set them on the context before changing relationships. With foreign key relationships, though, the relationship is simply a property of the entity, and if the entity passes its concurrency check, no further check is needed. You can change a relationship just by changing the foreign key value.
- Be careful of EntityKey collisions when attaching a graph to an ObjectContext. If, for instance, you are using DTOs and parts of your graph represent newly added entities for which the entity key values have not been set because they will be generated in the database, you should call the AddObject method to add the whole graph of entities first and then change entities not in the Added state to their intended state rather than calling the Attach method and then changing Added entities to that state. Otherwise, when you first call Attach, the Entity Framework thinks that every entity should be put into the Unchanged state, which assumes that the entity key values are final. If more than one entity of a particular type has the same key value (0, for example), the Entity Framework will throw an exception. By starting with an entity in the Added state, you avoid this problem because the framework does not expect Added entities to have unique key values.
- Turn off automatic lazy loading (another new EF4 feature) when returning entities from service methods. If you don't, the serializer will trigger lazy loading and try to retrieve additional entities from the database, which will cause more data than you intended to be returned (if your entities are thoroughly connected, you might serialize the whole database), or, more likely, you will receive an error because the context will be disposed of before the serializer tries to retrieve the data. Self-Tracking Entities does not have lazy loading turned on by default, but if you are creating a DTOs solution, this is something to watch out for.

And in the End

The .NET 4 release of the Entity Framework makes the creation of architecturally sound *n*-tier applications much easier. For most applications, I recommend starting with the Self-Tracking Entities template, which simplifies the process and enables the most reuse. If you have different rates of change between service and client, or if you need absolute control over your wire format, you should move up to a Data Transfer Objects implementation. Regardless of which pattern you choose, always keep in mind the key principles that the antipatterns and patterns represent—and never forget to validate your data before saving. ■

DANIEL SIMMONS *is an architect on the Entity Framework team at Microsoft.*

THANKS to the following technical experts for reviewing this article: *Jeff Derstadt, Diego Vega and Alex James.*

BRYAN SULLIVAN

XML Denial of Service Attacks and Defenses

Denial of service (DoS) attacks are among the oldest types of attacks against Web sites. Documented DoS attacks exist at least as far back as 1992, which predates SQL injection (discovered in 1998), cross-site scripting (JavaScript wasn't invented until 1995), and cross-site request forgery (CSRF attacks generally require session cookies, and cookies weren't introduced until 1994).

From the beginning, DoS attacks were highly popular with the hacker community, and it's easy to understand why. A single "script kiddie" attacker with a minimal amount of skill and resources could generate a flood of TCP SYN (for synchronize) requests sufficient to knock a site out of service. For the fledgling e-commerce world, this was devastating: if users couldn't get to a site, they couldn't very well spend money there either. DoS attacks were the virtual equivalent of erecting a razor-wire fence around a brick-and-mortar store, except that any store could be attacked at any time, day or night.

Over the years, SYN flood attacks have been largely mitigated by improvements in Web server software and network hardware. However, lately there has been a resurgence of interest in DoS attacks within the security community—not for "old school" network-level DoS, but instead for application-level DoS and particularly for XML parser DoS.

XML DoS attacks are extremely asymmetric: to deliver the attack payload, an attacker needs to spend only a fraction of the processing power or bandwidth that the victim needs to spend to handle the payload. Worse still, DoS vulnerabilities in code that processes XML are also extremely widespread. Even if you're using thoroughly tested parsers like those found in the Microsoft .NET Framework System.Xml classes, your code can still be vulnerable unless you take explicit steps to protect it.

This article describes some of the new XML DoS attacks. It also shows ways for you to detect potential DoS vulnerabilities and how to mitigate them in your code.

XML Bombs

One type of especially nasty XML DoS attack is the XML bomb—a block of XML that is both well-formed and valid according to the rules of an XML schema but which crashes or hangs a program when that program attempts to parse it. The best-known example of an XML bomb is probably the Exponential Entity Expansion attack.

Inside an XML document type definition (DTD), you can define your own entities, which essentially act as string substitution macros. For example, you could add this line to your DTD to replace all occurrences of the string &companyname; with "Contoso Inc.":

```
<!ENTITY companyname "Contoso Inc.">
```

You can also nest entities, like this:

```
<!ENTITY companyname "Contoso Inc.">
<!ENTITY divisionname "&companyname; Web Products Division">
```

While most developers are familiar with using external DTD files, it's also possible to include inline DTDs along with the XML data itself. You simply define the DTD directly in the <!DOCTYPE > declaration instead of using <!DOCTYPE> to refer to an external DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE employees [
  <!ELEMENT employees (employee)*>
  <!ELEMENT employee (#PCDATA)>
  <!ENTITY companyname "Contoso Inc.">
  <!ENTITY divisionname "&companyname; Web Products Division">
]>
<employees>
  <employee>Glenn P, &divisionname;</employee>
  <employee>Dave L, &divisionname;</employee>
</employees>
```

An attacker can now take advantage of these three properties of XML (substitution entities, nested entities, and inline DTDs) to craft a malicious XML bomb. The attacker writes an XML document with nested entities just like the previous example, but instead of nesting just one level deep, he nests his entities many levels deep, as shown here:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;
&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;
&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;
&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;
&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;
&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;
&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;
&lol8;">
]>
<lolz>&lol9;</lolz>
```

Some information in this article is based on a prerelease version of Microsoft .NET Framework 4. All information is subject to change.

Send your questions and comments to briefs@microsoft.com.

It should be noted that this XML is both well-formed and valid according to the rules of the DTD. When an XML parser loads this document, it sees that it includes one root element, "lolz", that contains the text "&lol9;". However, "&lol9;" is a defined entity that expands to a string containing ten "&lol8;" strings. Each "&lol8;" string is a defined entity that expands to ten "&lol7;" strings, and so forth. After all the entity expansions have been processed, this small (< 1 KB) block of XML will actually contain a billion "lol"s, taking up almost 3GB of memory! You can try this attack (sometimes called the Billion Laughs attack) for yourself using this very simple block of code—just be prepared to kill your test app process from Task Manager:

```
void processXml(string xml)
{
    System.Xml.XmlDocument document = new XmlDocument();
    document.LoadXml(xml);
}
```

Some of the more devious readers may be wondering at this point whether it's possible to create an infinitely recursing entity expansion consisting of two entities that refer to each other:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol1 "&lol2;">
  <!ENTITY lol2 "&lol1;">
]>
<lolz>&lol1;</lolz>
```

This would be a very effective attack, but fortunately it isn't legal XML and will not parse. However, another variation of the Exponential Entity Expansion XML bomb that does work is the Quadratic Blowup attack, discovered by Amit Klein of Trusteer. Instead of defining multiple small, deeply nested entities, the attacker defines one very large entity and refers to it many times:

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
  <!ENTITY a "aaaaaaaaaaaaaaaaaa...">
]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

If an attacker defines the entity "&a;" as 50,000 characters long, and refers to that entity 50,000 times inside the root "kaboom" element, he ends up with an XML bomb attack payload slightly over 200 KB in size that expands to 2.5 GB when parsed. This expansion ratio is not quite as impressive as with the Exponential Entity Expansion attack, but it is still enough to take down the parsing process.

Another of Klein's XML bomb discoveries is the Attribute Blowup attack. Many older XML parsers, including those in the .NET Framework versions 1.0 and 1.1, parse XML attributes in an extremely inefficient quadratic O(n^2) runtime. By creating an XML document with a large number of attributes (say 100,000 or more) for a single element, the XML parser will monopolize the processor for a long period of time and therefore cause a denial of service condition. However, this vulnerability has been fixed in .NET Framework versions 2.0 and later.

External Entity Attacks

Instead of defining entity replacement strings as constants, it is also possible to define them so that their values are pulled from external URIs:

```
<!ENTITY stockprice SYSTEM
   "http://www.contoso.com/currentstockprice.ashx">
```

While the exact behavior depends on the particular XML parser implementation, the intent here is that every time the XML parser encounters the entity "&stockprice;" the parser will make a request to www.contoso.com/currentstockprice.ashx and then substitute the response received from that request for the stockprice entity. This is undoubtedly a cool and useful feature of XML, but it also enables some devious DoS attacks.

The simplest way to abuse the external entity functionality is to send the XML parser to a resource that will never return; that is, to send it into an infinite wait loop. For example, if an attacker had control of the server adatum.com, he could set up a generic HTTP handler file at http://adatum.com/dos.ashx as follows:

```
using System;
using System.Web;
using System.Threading;

public class DoS : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        Thread.Sleep(Timeout.Infinite);
    }

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

He could then craft a malicious entity that pointed to http://adatum.com/dos.ashx, and when the XML parser reads the XML file, the parser would hang. However, this is not an especially effective attack. The point of a DoS attack is to consume resources so that they are unavailable to legitimate users of the application. Our earlier examples of Exponential Entity Expansion and Quadratic Blowup XML bombs caused the server to use large amounts of memory and CPU time, but this example does not. All this attack really consumes is a single thread of execution. Let's improve this attack (from the attacker's perspective) by forcing the server to consume some resources:

```
public void ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/plain";
    byte[] data = new byte[1000000];
    for (int i = 0; i < data.Length; i++) { data[i] = (byte)'A'; }
    while (true)
    {
        context.Response.OutputStream.Write(data, 0, data.Length);
        context.Response.Flush();
    }
}
```

This code will write an infinite number of 'A' characters (one million at a time) to the response stream and chew up a huge amount of memory in a very short amount of time. If the attacker is unable or unwilling to set up a page of his own for this purpose—perhaps he doesn't want to leave a trail of evidence that points back to him—he can instead point the external entity to a very large resource on a third-party Web site. Movie or file downloads can be especially effective for this purpose; for example, the Visual Studio 2010 Professional beta download is more than 2GB.

Yet another clever variation of this attack is to point an external entity at a target server's own intranet resources. Discovery of this attack technique is credited to Steve Orrin of Intel. This technique does require the attacker to have internal knowledge of intranet sites accessible by the server, but if an intranet resource attack can be executed, it can be especially effective because the server is spend-

Figure 1 **Configuring Timeout Values**

```
private const int TIMEOUT = 10000;  // 10 seconds

public override object GetEntity(Uri absoluteUri, string role,
   Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");

    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;
    return responseStream;
}
```

ing its own resources (processor time, bandwidth, and memory) to attack itself or its sibling servers on the same network.

Defending Against XML Bombs

The easiest way to defend against all types of XML entity attacks is to simply disable altogether the use of inline DTD schemas in your XML parsing objects. This is a straightforward application of the principle of attack surface reduction: if you're not using a feature, turn it off so that attackers won't be able to abuse it.

In .NET Framework versions 3.5 and earlier, DTD parsing behavior is controlled by the Boolean ProhibitDtd property found in the System.Xml.XmlTextReader and System.Xml.XmlReaderSettings classes. Set this value to true to disable inline DTDs completely:

```
XmlTextReader reader = new XmlTextReader(stream);
reader.ProhibitDtd = true;
```

or

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = true;
XmlReader reader = XmlReader.Create(stream, settings);
```

The default value of ProhibitDtd in XmlReaderSettings is true, but the default value of ProhibitDtd in XmlTextReader is false, which means that you have to explicitly set it to true to disable inline DTDs.

In .NET Framework version 4.0 (in beta at the time of this writing), DTD parsing behavior has been changed. The ProhibitDtd property has been deprecated in favor of the new DtdProcessing property. You can set this property to Prohibit (the default value) to cause the runtime to throw an exception if a <!DOCTYPE> element is present in the XML:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;
XmlReader reader = XmlReader.Create(stream, settings);
```

Alternatively, you can set the DtdProcessing property to Ignore, which will not throw an exception on encountering a <!DOCTYPE> element but will simply skip over it and not process it. Finally, you can set DtdProcessing to Parse if you do want to allow and process inline DTDs.

If you really do want to parse DTDs, you should take some additional steps to protect your code. The first step is to limit the size of expanded entities. Remember that the attacks I've discussed work by creating entities that expand to huge strings and force the parser to consume large amounts of memory. By setting the MaxCharactersFromEntities property of the XmlReaderSettings object,

Figure 2 **Capping the Maximum Amount of Data Retrieved**

```
private const int TIMEOUT = 10000;                    // 10 seconds
private const int BUFFER_SIZE = 1024;                 // 1 KB
private const int MAX_RESPONSE_SIZE = 1024 * 1024;    // 1 MB

public override object GetEntity(Uri absoluteUri, string role,
   Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");

    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;

    MemoryStream copyStream = new MemoryStream();
    byte[] buffer = new byte[BUFFER_SIZE];
    int bytesRead = 0;
    int totalBytesRead = 0;
    do
    {
        bytesRead = responseStream.Read(buffer, 0, buffer.Length);
        totalBytesRead += bytesRead;
        if (totalBytesRead > MAX_RESPONSE_SIZE)
            throw new XmlException("Could not resolve external entity");
        copyStream.Write(buffer, 0, bytesRead);
    } while (bytesRead > 0);

    copyStream.Seek(0, SeekOrigin.Begin);
    return copyStream;
}
```

you can cap the number of characters that can be created through entity expansions. Determine a reasonable maximum and set the property accordingly. Here's an example:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

Defending Against External Entity Attacks

At this point, we have hardened this code so that it is much less vulnerable to XML bombs, but we haven't yet addressed the dangers posed by malicious external entities. You can improve your resilience against these attacks if you customize the behavior of XmlReader by changing its XmlResolver. XmlResolver objects are used to resolve external references, including external entities. XmlTextReader instances, as well as XmlReader instances returned from calls to XmlReader.Create, are prepopulated with default XmlResolvers (actually XmlUrlResolvers). You can prevent XmlReader from resolving external entities while still allowing it to resolve inline entities by setting the XmlResolver property of XmlReaderSettings to null. This is attack surface reduction at work again; if you don't need the capability, turn it off:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
settings.XmlResolver = null;
XmlReader reader = XmlReader.Create(stream, settings);
```

If this situation doesn't apply to you—if you really, truly need to resolve external entities—all hope is not lost, but you do have a little more work to do. To make XmlResolver more resilient to denial of service attacks, you need to change its behavior in three ways. First, you need to set a request timeout to prevent infinite delay attacks. Second, you need to limit the amount of data that it will retrieve. Finally, as a defense-in-depth measure, you need to restrict the XmlResolver from retrieving resources on the local host. You can do all of this by creating a custom XmlResolver class.

The behavior that you want to modify is governed by the XmlResolver method GetEntity. Create a new class XmlSafeResolver derived from XmlUrlResolver and override the GetEntity method as follows:

```
class XmlSafeResolver : XmlUrlResolver
{
    public override object GetEntity(Uri absoluteUri, string role,
        Type ofObjectToReturn)
    {

    }
}
```

The default behavior of the XmlUrlResolver.GetEntity method looks something like the following code, which you can use as a starting point for your implementation:

```
public override object GetEntity(Uri absoluteUri, string role,

    Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    System.Net.WebResponse response = request.GetResponse();
    return response.GetResponseStream();
}
```

The first change is to apply timeout values when making the request and when reading the response. Both the System.Net.WebRequest and the System.IO.Stream classes provide inherent support for timeouts. In the sample code shown in **Figure 1**, I simply hardcode the timeout value, but you could easily expose a public Timeout property on the XmlSafeResolver class if you want greater configurability.

The next step is to cap the maximum amount of data that is retrieved in the response. There's no "MaxSize" property or the equivalent for the Stream class, so you have to implement this functionality yourself. To do this, you can read data from the response stream one chunk at a time and copy it into a local stream cache. If the total number of bytes read from the response stream exceeds a predefined limit (again hardcoded for simplicity only), you stop reading from the stream and throw an exception (see **Figure 2**).

As an alternative, you can wrap the Stream class and implement the limit checking directly in the overridden Read method (see **Figure 3**). This is a more efficient implementation since you save the extra memory allocated for the cached MemoryStream in the earlier example.

Now, simply wrap the stream returned from WebResponse.GetResponseStream in a LimitedStream and return the LimitedStream from the GetEntity method (see **Figure 4**).

Finally, add one more defense-in-depth measure by blocking entity resolution of URIs that resolve to the local host (see **Figure 5**).This includes URIs starting with http://localhost, http://127.0.0.1, and file:// URIs. Note that this also prevents a very nasty information disclosure vulnerability in which attackers can craft entities pointing to file://resources, the contents of which are then duly retrieved and written into the XML document by the parser.

Figure 3 **Defining a Size-Limited Stream Wrapper Class**

```
class LimitedStream : Stream
{
    private Stream stream = null;
    private int limit = 0;
    private int totalBytesRead = 0;

    public LimitedStream(Stream stream, int limit)
    {
        this.stream = stream;
        this.limit = limit;
    }

    public override int Read(byte[] buffer, int offset, int count)
    {
        int bytesRead = this.stream.Read(buffer, offset, count);
        checked { this.totalBytesRead += bytesRead; }
        if (this.totalBytesRead > this.limit)
            throw new IOException("Limit exceeded");

        return bytesRead;
    }

    ...
}
```

Now that you've defined a more secure XmlResolver, you need to apply it to XmlReader. Explicitly instantiate an XmlReaderSettings object, set the XmlResolver property to an instance of XmlSafeResolver, and then use the XmlReaderSettings when creating XmlReader, as shown here:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = new XmlSafeResolver();
settings.ProhibitDtd = false;   // comment out if .NET 4.0 or later
settings.DtdProcessing = DtdProcessing.Parse;  // comment out if
                                               // .NET 3.5 or earlier
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

Additional Considerations

It's important to note that in many of the System.Xml classes, if an XmlReader is not explicitly provided to an object or a method, then one is implicitly created for it in the framework code. This implicitly created XmlReader will not have any of the additional defenses specified in this article, and it will be vulnerable to attack. The very first code snippet in this article is a great example of this behavior:

```
void processXml(string xml)
{
    System.Xml.XmlDocument document = new XmlDocument();
    document.LoadXml(xml);
}
```

This code is completely vulnerable to all the attacks described in this article. To improve this code, explicitly create an XmlReader with appropriate settings (either disable inline DTD parsing or specify a safer resolver class) and use the XmlDocument.Load(XmlReader) overload instead of XmlDocument.LoadXml or any of the other XmlDocument.Load overloads, as shown in **Figure 6**.

XLinq is somewhat safer in its default settings; the XmlReader created by default for System.Xml.Linq.XDocument does allow DTD parsing, but it automatically sets MaxCharactersFromEntities to 10,000,000 and prohibits external entity resolution. If you are explicitly providing an XmlReader to XDocument, be sure to apply the defensive settings described earlier.

Figure 4 **Using LimitedStream in GetEntity**

```
private const int TIMEOUT = 10000;                   // 10 seconds
private const int MAX_RESPONSE_SIZE = 1024 * 1024;   // 1 MB

public override object GetEntity(Uri absoluteUri, string role, Type
ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");


    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;

    return new LimitedStream(responseStream, MAX_RESPONSE_SIZE);
}
```

Figure 5 **Blocking Local Host Entity Resolution**

```
public override object GetEntity(Uri absoluteUri, string role,
    Type ofObjectToReturn)
{
    if (absoluteUri.IsLoopback)
        return null;

    ...


}
```

Figure 6 **Applying Safer Entity Parsing Settings to XmlDocument**

```
void processXml(string xml)
{
    MemoryStream stream =
        new MemoryStream(Encoding.Default.GetBytes(xml));
    XmlReaderSettings settings = new XmlReaderSettings();

    // allow entity parsing but do so more safely
    settings.ProhibitDtd = false;
    settings.MaxCharactersFromEntities = 1024;
    settings.XmlResolver = new XmlSafeResolver();

    XmlReader reader = XmlReader.Create(stream, settings);
    XmlDocument doc = new XmlDocument();
    doc.Load(reader);
}
```

Wrapping Up

XML entity expansion is a powerful feature, but it can easily be abused by an attacker to deny service to your application. Be sure to follow the principle of attack surface reduction and disable entity expansion if you don't require its use. Otherwise, apply appropriate defenses to limit the maximum amount of time and memory your application can spend on it. ■

BRYAN SULLIVAN *is a security program manager for the Microsoft Security Development Lifecycle team, specializing in Web application and .NET security issues. He is the author of "AJAX Security" (Addison-Wesley, 2007).*

THANKS to the following technical expert for reviewing this article: *Sergey Dubinets*

Visualizing Spatial Data

In SQL Server 2008, Microsoft introduced spatial data support with two new built-in data types, geometry and geography. Although you could "see" the data in spatial columns in three formats—Well-Known Text, Well-Known Binary, and Geographic Markup Language (GML)—the only built-in way to visualize your data on a map was via the Spatial Results tab that was added to SQL Server Management Studio. This was a boon to developers who were visualizing map polygons or even geometric data unrelated to a map (the geometric layout of a warehouse, perhaps), but if you had a collection of points with locations of cities, you could see only the points. To add a "base map" (for example, a map of the world to layer your city locations on), you could use the UNION ALL syntax with a SELECT statement and visualize it with the Spatial Results tab:

```
SELECT city_name, geog FROM cities
UNION ALL
SELECT NULL, geog FROM map_of_the_world_table;
```

But SQL Server Management Studio is an administrator and programmer tool. You'd like to be able to visualize your data inside the reports you're producing for management. It's easier to see trends when graphics are involved, and as Waldo Tobler's First Law of Geography states, "Everything is related to everything else, but closer things are more closely related." In this article, I want to show you three new arrivals on the SQL Server spatial visualization scene: the map control in SQL Server 2008 R2 Reporting Services (SSRS), the ESRI MapIt product, and the MapPoint Add-In for SQL Server 2008.

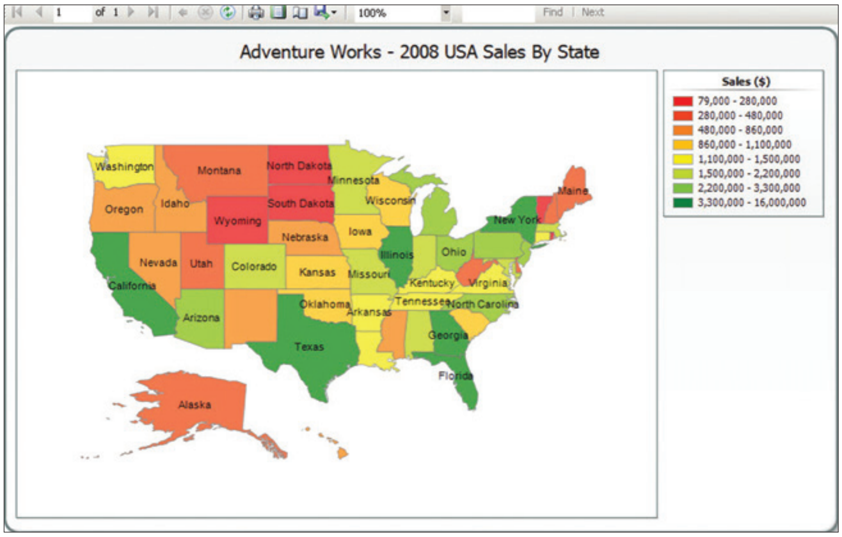


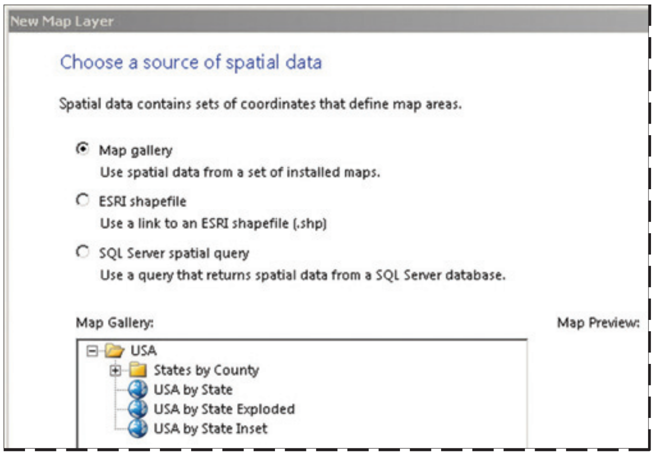
Figure 1 **Choropleth Map of Sales in the U.S. by State**

Reporting Services Map Control

Map visualizations use a layer concept. The background or base map (for example, the map of the world) is overlaid with one or more layers of spatial information and database information. Spatial information might consist of sets of polygons like your sales territories, linestrings like roads or rivers, or points like store locations. One of the most common types of map for analyzing business data is a choropleth, a thematic map in which areas are shaded in proportion to the statistical variable you'd like to analyze. For example, a choropleth map showing sales by state in the United States might look like the screenshot in **Figure 1**. Add a second layer that shows the location of your stores, and you have another layer of analysis available.

As a report programmer, you can access SQL Server 2008 R2 Map Control through either Report Builder 3.0 or a Reporting Services project in Business Intelligence Development Studio (BIDS). The Map appears as one of the choices on the main Report Builder template. In a BIDS project, you add a Map Control to a report by dragging it from the toolbox. Either way, the map wizard walks you through the basics.

Send your questions and comments for Bob to mmdbdev@microsoft.com.


New Map Layer

Choose a source of spatial data

Spatial data contains sets of coordinates that define map areas.

[✓ Map gallery]
Use spatial data from a set of installed maps.

ESRI shapefile
Use a link to an ESRI shapefile (.shp)

SQL Server spatial query
Use a query that returns spatial data from a SQL Server database.

Map Gallery:
- USA
  - States by County
  - USA by State
  - USA by State Exploded
  - USA by State Inset

Figure 2 **New Map Layer Dialog in SSRS 2008 R2**

First, you choose your base map using the New Map Layer dialog. The base map can come from the built-in Map Gallery, an ESRI shapefile (a standard data format for representing geospatial data), or a SQL Server spatial query. This is shown in **Figure 2**.

If you choose a SQL Server spatial query, you'll obtain map data from a spatial column in a SQL Server table. The wizard walks you through creating a Data Connection when you are creating a query with the Query Designer. Your query must contain at least one column of data type geometry or geography or you'll get an error. The next dialog, Choose Spatial Data and Map View options, allows you to choose a spatial field (in case your query includes multiple spatial fields) and a Layer Type as shown in **Figure 3**. Ordinarily, a spatial column in a table will contain all points, all linestrings or all polygons, and the Map wizard looks at the data and returns the kind of layer type that corresponds. You can change the layer type; however, if you choose a field that contains all polygons (or multipolygons) but select Layer Type: Point, no map data will appear in the preview pane. One nice feature is the option to include a Bing Maps background, so if you have point data, you can use Bing Maps tiles as the base map background for your map control. With Bing Maps tiles as a background, you can choose Street Map, Aerial, or Hybrid view. If you choose Bing Maps, your spatial data will be layered on top of the Bing Map base layer.

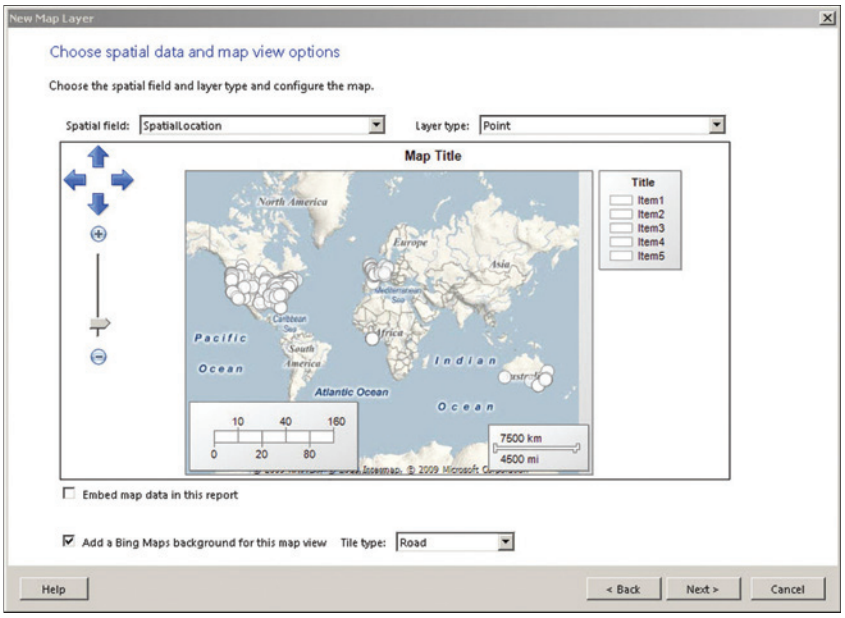
Once you have set your base layer or layers, you're presented with a set of map type choices, which vary depending whether your spatial data consists of points, linestrings or polygons. For example, if your layer contains polygons, you can choose between Basic Map (just the spatial data you've selected), Color Analytical Map (the choropleth maps mentioned earlier where the color of each polygon is based on an analytic variable), or Bubble Map (where a symbol in the center of each area is sized proportionally by an analytic variable). Linestring data gives you a choice of Basic Line Map or Analytical Line Map. Point data gives you a choice of Basic Marker Map, Analytical Marker Map or Bubble Map. The choropleth map in **Figure 1** is a Color Analytical Map produced by the wizard.

Choosing a color analytical or bubble map leads to a panel that lets you select the data column to analyze. This data may be in the same dataset as your spatial data or in a different dataset with a related field. For example, you might have a table that contains state information and another table that contains SalesTotals for each state. Although the map shape data can only come from shapefiles, map gallery or SQL Server spatial tables, the analytic data can come from any data source, including SQL Server Analysis Services.

After you've chosen your analytic data source, if needed, the final panel allows you to choose common visualization aspects, such as the size of bubbles in a bubble map or polygon colors, to visualize data. You can also choose whether your layers' labels will appear. For example, if your map consists of polygons that represent shapes of states, the label might be the name of the state. Bear in mind that labels only appear in polygons where the polygon is large enough to hold the text.

Of course, as with wizard-based development in general, the wizard only scratches the surface of what you can do. The Map portion of the control is contained within a Viewport, which is a separate control that enforces the boundaries of the map on the report page. The map control properties are divided roughly into Viewport Properties, Map Properties and Layer Properties. You can change the properties using either the context menus or a more detailed view provided in the Properties Windows. The set of properties that you see in the corresponding Properties Window depend on which part of the Map Control has the focus.

The Map Projection is specified in the Viewport's properties. Your choices depend on the spatial column type you're using in the


New Map Layer

Choose spatial data and map view options

Choose the spatial field and layer type and configure the map.

Spatial field: SpatialLocation

Layer type: Point

Embed map data in this report

[✓ Add a Bing Maps background for this map view]

Tile type: Road

Figure 3 **Choose Spatial Data and Map Options**

layer, and it's important to realize how your data and SRID (spatial reference ID) affects your choice. To start with the simplest case, suppose you'd like a map of the layout of your company's warehouse (a physical warehouse where goods are stored, not a data warehouse). You'd likely measure the warehouse, draw a floor plan, and map the placement of the goods based on their location in the floor plan. In this case, the distance being mapped is so small that the fact that the Earth is round rather than flat does not matter. You're measuring in terms of a geometric coordinate system (X and Y coordinates) and SQL Server's geometry data type. In this case, you set the Viewport's Coordinate System property to Planar.

The more common case is that you're plotting positions on the Earth in terms of latitude and longitude. Your map data contains latitude and longitude, so you're probably using the geography data type. To produce a flat map from the earth (which is technically an oblate spheroid), the map control provides a set of map projections. To use the map projections appropriately, your geography data type column should be using SRID 4326, 4269 or one of the other common Earth-related SRIDs that the map control is expecting. In this case, you set the Viewport's Coordinate System property to Geographic and choose a map projection as the Viewport's Projection property list.

The last case is when your data is using a projected coordinate system. In this scenario, you're using SQL Server's geometry data type, not geography. The map control will do nothing to project the data, because the projection information is already part of the data type itself. Common SRIDs for the projected coordinate system include the State Plane data projection or the British National Grid. In this case, you set the Viewport's Coordinate System property to Planar.

The reason I mention all this is that, except for projecting geography coordinates (which are assumed to be 4326 - WGS84 coordinates), the map control will not automatically reproject between different coordinate systems. For example, you can't mix a Bing Maps Tile Layer (geographic) with a British National Grid Layer (planar) on the same map. The result wouldn't be pretty.

The big win with the Reporting Services 2008 R2 Map Control is that the SQL Server spatial data is automatically visualizable in a variety of map formats. Correlations between the business data and spatial data are easy to accomplish with the wizard, and all the additional power you need is available with a host of customizations through properties. One of my favorite (nonspatial) features of the map control is the ability to enable drilldown through the normal "Action" mechanism in SQL Server Reporting Services. Simply select the Action tab in the Map Properties dialog and you can link together reports that contain maps that show Sales by Country on a World map, then drill down to State or Region, then to City. At the City level, you could add a point layer with the location of your stores. For additional information about preparing and using spatial data with the SQL Server 2008 R2 Reporting Services map control, see Ed Katibah's excellent blog series starting at blogs.msdn.com/edkatibah/archive/2009/05/09/cartographic-adjustment-of-spatial-data-for-sql-server-reporting-services-part-1.aspx.

Figure 4 **Using the Spatial Data Service with a View**

```
<!—UserControl element and namespace declarations elided for clarity -->
<Grid x:Name="LayoutRoot">
  <esri:Map x:Name="MyMap"  >
    <esri:Map.Layers>
      <esri:ArcGISTiledMapServiceLayer ID="StreetMapLayer"
        Url="http://server.arcgisonline.com/ArcGIS/rest/services/ESRI_
StreetMap_World_2D/MapServer"/>
        <esri:FeatureLayer ID="MyFeatureLayer"
          Url="http://zmv10/SDS/databases/AdventureWorks2008/Sales.
vIndividualCustomerSpatial"
          Where="CountryRegionName = 'United States'">
         <esri:FeatureLayer.Clusterer>
           <esri:FlareClusterer FlareBackground="#99FF0000"
             FlareForeground="White" MaximumFlareCount="9" />
          </esri:FeatureLayer.Clusterer>
        </esri:FeatureLayer>
      </esri:Map.Layers>
    </esri:Map>
</Grid>
```

The ESRI MapIt Product

ESRI, the world's leading GIS company, released a product named MapIt at its 2009 Users Conference. MapIt is actually a set of components that make it easier to work with existing business data that involves location. This product directly produces and consumes SQL Server 2008 spatial data types, so no additional ESRI software is required to use it.

The most common example of location-based business data is address data, and so MapIt includes a program, the Spatial Data Assistant, that performs geocoding on address data in different formats. The Spatial Data Assistant will add a new Geometry column to any SQL Server table with addresses and populate it by calling Web-based services. You have a choice of the ESRI Map Service or the Bing Maps Geocoding service. Once the existing data is updated, you can use geocoding in a trigger to keep the location in sync. An example of such a trigger is available on the MapIt support site.

To display your business data along with other map layers (for example, addresses of students along with a layer that consists of school district boundaries), you'll need to import the additional location information into SQL Server 2008. The Spatial Data Assistant can import existing GIS data in ESRI Shapefile format or sets of free ESRI map data available online. On import, you're allowed to change the data projection to correspond to the projection used by Bing Maps or the ARCGIS Server. You can even specify the SRID of your choice from a list of supported SRIDs. The import function produces a SQL Server geometry column in the imported table.

Once you have your business data and other map layer data in SQL Server, you can visualize it in your own programs. ESRI provides a free Silverlight and Windows Presentation Foundation (WPF) API that can be used to create rich Web and Windows-based applications. To enable the use of your SQL Server spatial data with these APIs, MapIt includes a REST-based Spatial Data Service, which allows you to expose one or more Web endpoints that produce JSON or HTML output in the format that the Silverlight APIs consume. It includes a Spatial Data Services directory that allows you to browse your tables and views, and query the ones that include geometry and geography columns. Once you've decided on the data you'd like to use in your map, simply copy that table or view's URL from the Spatial Data Services directory location and paste it into an application that uses the API. Here's an example of exposing the Customer addresses in the AdventureWorks2008 database using the Spatial Data Service with a view, Sales.vIndividualCustomerSpatial, that I created by adding the SpatialLocation column to an existing AdventureWorks2008 view (**Figure 4**). This map includes one of my favorite features—adding a clustering component to cluster points at lower map resolutions. As you drill down into higher map resolutions, the point clusters are visible as individual points.

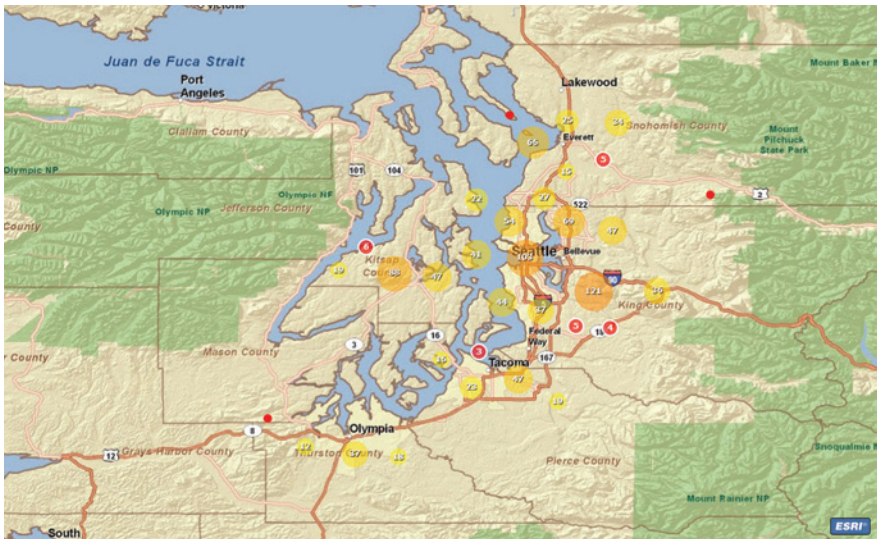


Figure 5 **MapIt-Generated Map with Customer Locations Showing Point Clustering**

This code produces the results shown in **Figure 5**. Note that you can mix and match SQL Server 2008 spatial data layers with ARCGIS or Bing Maps base map layers, as well as other layers you've imported with the Spatial Data Assistant. For example, you can add a layer containing SalesPerson addresses to visualize the location of your salespeople vis-à-vis your customers. A Layer type is also available for Graphics and a Drawing surface is supported in the API for further map interactivity. Finally, there are specialized layers available at the ESRI support Web site for GEORSS and KML. I've barely scratched the surface of the API's functionality. For more information about the Silverlight and WPF API, see resources.esri.com/arcgisserver/apis/silverlight/index.cfm.

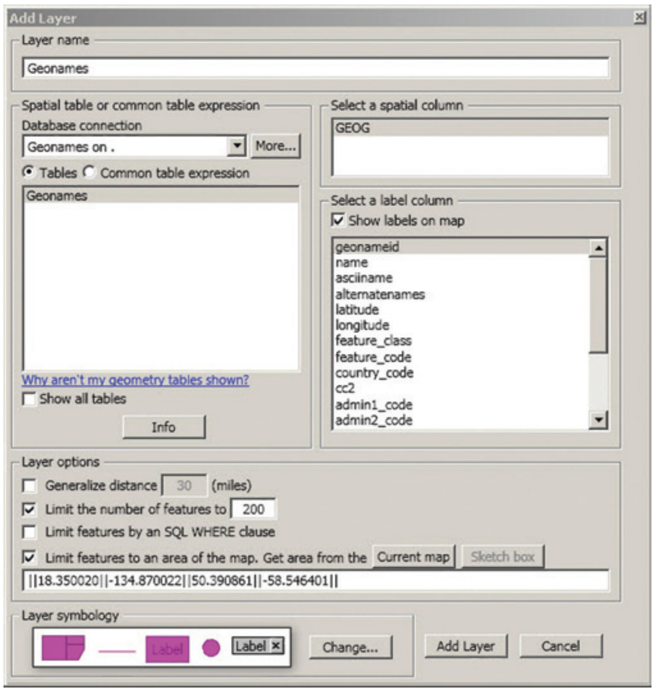
You may want to provide professional-looking maps based on your SQL Server data without incurring the cost of programming and maintaining a Silverlight or WPF application. MapIt includes a Web Part that allows you to add maps to SharePoint sites with no programming. You can include spatial data from:

- SharePoint lists that include addresses or latitude/longitude fields
- SQL Server spatial data exposed with the Spatial Data Assistant
- ARCGIS Server data

Editing an instance of the ESRI SharePoint Web Part permits you to import layers of spatial data by providing a URL or SharePoint list name. You can use either Bing Maps or ARCGIS Server as a base map. Additional dialogs enable you to specify filter expressions, define pop-ups that display additional information as you hover over individual points (these are known as MapTips in the Silverlight API), choose symbology and add point clustering by clicking a checkbox. The maps will refresh to display the most current information if you have dynamically changing data. You can configure data refresh and caching at an individual Web Part level, or globally for the Spatial Data Service.

MapPoint 2010 Add-In for SQL Server 2008

MapPoint is Microsoft's original offering in the realm of business spatial data visualization and location-based queries. One of its strengths is that MapPoint comes with a rich collection of multi-level map information that works in an offline mode, as well as letting you add your own spatial layers. The latest versions of MapPoint even come with GPS integration. In August 2009, Microsoft released a free MapPoint Add-In for SQL Server 2008 spatial data, which makes it easy to add layers of SQL Server-based information, perform spatial queries against the data and the layers, and save the end product to disk for further refinement. The saved version of the map can be distributed without requiring access to SQL Server to use.


Add Layer

Layer name
Geonames

Spatial table or common table expression
Database connection
Geonames on . More...
(•) Tables ( ) Common table expression
Geonames
Why aren't my geometry tables shown?
☐ Show all tables
Info

Select a spatial column
GEOG

Select a label column
☑ Show labels on map
geonameid
name
asciiname
alternatenames
latitude
longitude
feature_class
feature_code
country_code
cc2
admin1_code
admin2_code

Layer options
☐ Generalize distance 30 (miles)
☑ Limit the number of features to 200
☐ Limit features by an SQL WHERE clause
☑ Limit features to an area of the map. Get area from the Current map Sketch box
||18.350020||-134.870022||50.390861||-58.546401||

Layer symbology
Label Label ✕ Change...
Add Layer Cancel

Figure 6 **MapPoint Add-In Add Layer Dialog**

The WPF-based add-in uses a direct ADO.NET connection to talk to SQL Server. Once you've connected to the SQL Server instance and the database that contains your spatial data in the Add Layer tab (shown in **Figure 6**), you choose a table that contains a spatial column to use as your layer. You can reduce the amount of data that is returned (and therefore make the layer populate faster) by limiting features to the current map extent, as well as selecting a subset of columns and generalizing the data that's returned. You can also specify a SQL WHERE clause on your query. The query interface was designed for maximum speed; to ensure the spatial index is used, you can choose to use a common table expression for more complex queries. Of course, a dialog is provided that gives you complete control of the symbology.

The Map Tips that appear when you select spatial features like points include all of the data in the table you've specified. This data is "live," that is, you can edit the data corresponding to individual spatial feature tables and the updated data will be saved to the database when you hit the Save button. You can even add new features (rows containing spatial data) or delete features directly from the add-in.

Finally, you can refine queries based on an existing spatial data layer. For example, once I've retrieved a set of school locations based on a map extent that roughly corresponds to my city limits, I can query that set to find the schools within a mile of my house. Or I can query on any of the other fields I've retrieved. Each additional query will bring back an additional layer, and you can hide, show and refine your layers before saving the map. When you save the map, it not only saves the graphic layers but also the SQL queries that produced them, so you can update your layers if the data changes. For sub-layer queries, the generated SQL is visible in the window before you execute the query. There's a user-settable timeout value, and you can cancel executing layer queries in progress. The fact that you're connecting directly to SQL Server makes the experience much more interactive. There's even a utility that allows you to import data into SQL Server from shape files or MapInfo .MIF files.

The products and features I've mentioned here are just the latest ones that expand the ways to use SQL Server spatial data. In addition, Safe Software's FME for SQL Server 2008 product is integrated with SQL Server Integration Services (SSIS) to allow spatial ETL in SSIS workflows. Perhaps the next release of SQL Server will see more integration with SQL Server Analysis Services and even data mining features. The built-in and third-party visualization support expands the usefulness of location data beyond analyzing addresses, to make ordinary business data come alive. ■

BOB BEAUCHEMIN *is a database-centric application practitioner and architect, course author and instructor, writer and Developer Skills partner at SQLskills. He's written books and articles on SQL Server, data access and integration technologies, and database security. You can reach him at bobb@sqlskills.com.*

THANKS to the following technical expert for reviewing this article: *Ed Katibah*

Workflow Services for Local Communication

In a previous column (see "Workflow Communications" in the September 2007 issue of *MSDN Magazine* at msdn.microsoft.com/magazine/cc163365.aspx), I wrote about the core communication architecture in Windows Workflow Foundation 3 (WF3). One topic I did not cover is the local communications activities that are one abstraction on top of this communication architecture. If you look at .NET Framework 4 Beta 1, you will notice no HandleExternalEvent activity. In fact, with WF4, the communications activities included are built on Windows Communication Foundation (WCF). This month, I'll show you how to use WCF for communication between a workflow and a host application in Windows Workflow Foundation 3. Gaining this knowledge should help with your development efforts using WF3 and prepare you for WF4, where WCF is the only abstraction over queues (referred to as "bookmarks" in WF4) that ships with the framework. (For basic information on Workflow Services in WF3, see my Foundations column in the Visual Studio 2008 Launch issue of *MSDN Magazine*, at msdn.microsoft.com/magazine/cc164251.aspx.)

Overview

Communication between host applications and workflows proves challenging for some developers because they can easily overlook the fact that the workflow and host often execute on different threads. The design of the communication architecture is intended to shield developers from having to worry about managing thread context, marshaling data and other low-level details. One abstraction over the queuing architecture in WF is the WCF messaging integration that was introduced in Version 3.5 of the .NET Framework. Most examples and labs show how the activities and extensions to WCF can be used to expose a workflow to clients that are external to the hosting process, but this same communication framework can be used to communicate within the same process.

Implementing the communication involves several steps, but the work does not amount to much more than what you would have to do with the local communication activities.

Before you can do anything else, you need to define (or minimally begin to define in an iterative approach) the contracts for communication using WCF service contracts. Next, you need to use those contracts in your workflows to model the communication points in the logic. Finally, to hook it all together, the workflow and other services need to be hosted as WCF services with endpoints configured.

Figure 1 **Contracts for Communication**

```
[ServiceContract(
  Namespace = "urn:MSDN/Foundations/LocalCommunications/WCF")]
public interface IHostInterface
{
[OperationContract]
void OrderStatusChange(Order order, string newStatus, string oldStatus);
}

[ServiceContract(
  Namespace="urn:MSDN/Foundations/LocalCommunications/WCF")]
public interface IWorkflowInterface
{
  [OperationContract]
  void SubmitOrder(Order newOrder);

  [OperationContract]
  bool UpdateOrder(Order updatedOrder);
}

[DataContract]
public class Order
{
  [DataMember]
  public int OrderID { get; set; }
  [DataMember]
  public string CustomerName { get; set; }
  [DataMember]
  public double OrderTotal { get; set; }
  [DataMember]
  public string OrderStatus { get; set; }
  }
```

Modeling Communication

The first step in modeling communication is to define the contracts between your host application and the workflow. WCF services use contracts to define the collection of operations that make up the service and the messages that are sent and received. In this case, because you are communicating from the host to the workflow and from the workflow to the host, you need to define two service contracts and related data contracts, as shown in **Figure 1.**

With the contracts in place, modeling the workflow using the Send and Receive activities works as it does for remote communication. That is one of the beautiful things about WCF: remote or local, the programming model is the same. As a simple example, **Figure 2** shows a workflow with two Receive activities

Send your questions and comments for Matt to mmnet30@microsoft.com.

Code download available at code.msdn.microsoft.com/mag200911Foundations.

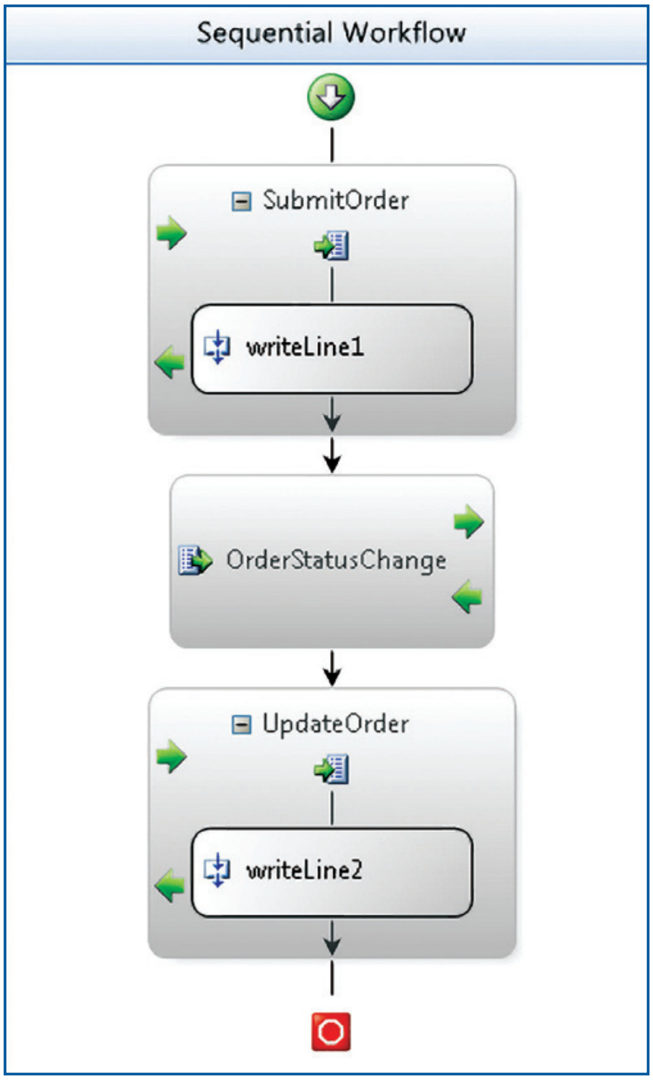


Figure 2 **Workflow Modeled Against Contracts**

and one Send activity modeling the communication between the workflow and the host. The receive activities are configured with the IWorkflowInterface service contract, and the Send activity uses the IHostInterface contract.

So far, using WCF for local communications is not much different from using WCF for remote communications and is very similar to using the local communications activities and services. The main difference comes in how the host code is written to start the workflow and handle communication coming from the workflow.

Hosting the Services

Because we want communication to flow both ways using WCF, we need to host two services—the workflow service to run the workflow and a service in the host application to receive messages from the workflow. In my example, I built a simple Windows Presentation Foundation (WPF) application to act as the host and used the App class's OnStartup and OnExit methods to manage the hosts. Your first inclination might be to create the WorkflowServiceHost class

Figure 3 **Hosting the Workflow Service**

```
ThreadPool.QueueUserWorkItem((o) =>
{

//host the workflow
workflowHost = new WorkflowServiceHost(typeof(
  WorkflowsAndActivities.OrderWorkflow));
workflowHost.AddServiceEndpoint(
  "Contracts.IWorkflowInterface", LocalBinding, WFAddress);

try
{
  workflowHost.Open();
}
catch (Exception ex)
{
  workflowHost.Abort();
  MessageBox.Show(String.Format(
    "There was an error hosting the workflow as a service: {0}",
    ex.Message));
}
});
```

and open it right in the OnStartup method. Since the Open method does not block after the host is open, you can continue processing, load the user interface and begin interacting with the workflow. Because WPF (and other client technologies) uses a single thread for processing, this soon leads to problems because both the service and the client call cannot use the same thread, so the client times out. To avoid this, the WorkflowServiceHost is created on another thread using the ThreadPool, as shown in **Figure 3.**

The next challenge you encounter is choosing the appropriate binding for local communication. Currently, there is no in-memory or in-process binding that is extremely lightweight for these kinds of scenarios. The best option for a lightweight channel is to use the NetNamedPipeBinding with security turned off. Unfortunately, if you try to use this binding and host the workflow as a service, you get an error informing you that the host requires a binding with the Context channel present because your service contract may require a session. Further, there is no NetNamedPipeContextBinding included with the .NET Framework, which ships with only three context bindings: BasicHttpContextBinding, NetTcpContextBinding and WSHttpContextBinding. Fortunately, you can create your own custom bindings to include the context channel. **Figure 4** shows a custom binding that derives from the NetNamedPipeBinding class and injects the ContextBindingElement into the binding. Communication in both directions can now use this binding in the endpoint registration by using different addresses.

With this new binding, you can create an endpoint on the WorkflowServiceHost and open the host with no more errors. The workflow is ready to receive data from the host using the service contract. To send that data, you need to create a proxy and invoke the operation, as shown in **Figure 5**.

Because you're sharing the contracts, there is no proxy class, so you have to use the ChannelFactory<TChannel> to create the client proxy.

While the workflow is hosted and ready to receive messages, it still needs to be configured to send messages to the host. Most important, the workflow needs to be able to get a client endpoint when using the Send activity. The Send activity allows you to specify the

Figure 4 **NetNamedPipeContextBinding**

```
public class NetNamedPipeContextBinding : NetNamedPipeBinding
{
 public NetNamedPipeContextBinding() : base(){}

 public NetNamedPipeContextBinding(
   NetNamedPipeSecurityMode securityMode):
   base(securityMode) {}

 public NetNamedPipeContextBinding(string configurationName) :
  base(configurationName) {}

 public override BindingElementCollection CreateBindingElements()
 {
  BindingElementCollection baseElements = base.CreateBindingElements();
  baseElements.Insert(0, new ContextBindingElement(
    ProtectionLevel.EncryptAndSign,
    ContextExchangeMechanism.ContextSoapHeader));

  return baseElements;
 }
}
```

endpoint name, which is typically a mapping to a named endpoint in the configuration file. Although putting the endpoint information in a configuration file works, you can also use the ChannelManager-Service (as discussed in my August 2008 column at msdn.microsoft.com/magazine/cc721606.aspx) to hold the client endpoints used by your Send activities in the workflow. **Figure 6** shows the hosting code to create the service, provide it with a named endpoint, and add it to the WorkflowRuntime hosted in the WorkflowServiceHost.

One way to handle the context problem is to use the same client proxy for every invocation of the service.

Having the workflow service hosted provides the ability to send messages from the host to the workflow, but to get messages back to the host, you need a WCF service that can receive messages from the workflow. This service is a standard WCF service self-hosted in the application. Because the service is not a workflow service, you can use the standard NetNamedPipeBinding or reuse the NetNamedPipeContextBinding shown previously. Finally, because this service is invoked from the workflow, it can be hosted on the UI thread, making interaction with UI elements simpler. **Figure 7** shows the hosting code for the service.

With both services hosted, you can now run the workflow, send a message and receive a message back. However, if you try to send a second message using this code to the second receive activity in the workflow, you will receive an error about the context.

Handling Instance Correlation

One way to handle the context problem is to use the same client proxy for every invocation of the service. This enables the client

Figure 5 **Host Code to Start a Workflow**

```
App a = (App)Application.Current;
  IWorkflowInterface proxy = new ChannelFactory<IWorkflowInterface>(
    a.LocalBinding, a.WFAddress).CreateChannel();

  proxy.SubmitOrder(
    new Order
    {
      CustomerName = "Matt",
      OrderID = 0,
      OrderTotal = 250.00
    });
```

Figure 6 **Adding the ChannelManagerService to the Runtime**

```
ServiceEndpoint endpoint = new ServiceEndpoint
(
 ContractDescription.GetContract(typeof(Contracts.IHostInterface)),
  LocalBinding, new EndpointAddress(HostAddress)
);
endpoint.Name = “HostEndpoint”;

WorkflowRuntime runtime =
 workflowHost.Description.Behaviors.Find<WorkflowRuntimeBehavior>().
WorkflowRuntime;

ChannelManagerService chanMan =
 new ChannelManagerService(
  new List<ServiceEndpoint>
  {
   endpoint
  });

runtime.AddService(chanMan);
```

proxy to manage the context identifiers (using the NetNamedPipeContextBinding) and send them back to the service with subsequent requests.

In some scenarios, it's not possible to keep the same proxy around for all requests. Consider the case where you start a workflow, persist it to a database and close the client application. When the client application starts up again, you need a way to resume the workflow by sending another message to that specific instance. The other common use case is when you do want to use a single client proxy, but you need to interact with several workflow instances, each with a unique identifier. For example, the user interface provides a list of orders, each with a corresponding workflow, and when the user invokes an action on a selected order, you need to send a message to the workflow instance. Letting the binding manage the context identifier will not work in this scenario because it will always be using the identifier of the last workflow with which you interacted.

For the first scenario—using a new proxy for each call—you need to manually set the workflow identifier into the context by using the IContextManager interface. IContextManager is accessed through the GetProperty<TProperty> method on the IClientChannel interface. Once you have the IContextManager, you can use it to get or set the context.

The context itself is a dictionary of name-value pairs, the most important of which is the instanceId value. The following code shows how you retrieve the ID from the context so it can be stored by your client application for later, when you need to interact with the same workflow instance. In this example, the ID is being displayed in the client user interface rather than being stored in a database:

```
IContextManager mgr = ((IClientChannel)proxy).
GetProperty<IContextManager>();

string wfID = mgr.GetContext()["instanceId"];
wfIdText.Text = wfID;
```

Once you make the first call to the workflow service, the context is automatically populated with the instance ID of the workflow by the context binding on the service endpoint.

When using a newly created proxy to communicate with a workflow instance that was previously created, you can use a similar method to set the identifier in the context to ensure your message is routed to the correct workflow instance, as shown here:

```
IContextManager mgr = ((IClientChannel)proxy).
GetProperty<IContextManager>();
  mgr.SetContext(new Dictionary<string, string>{
    {"instanceId", wfIdText.Text}
  });
```

When you have a newly created proxy, this code works fine the first time but not if you try to set the context a second time for invoking another workflow instance. The error you get tells you that you cannot change the context when automatic context management is enabled. Essentially, you are told that you can't have your cake and it eat too. If you want the context to be managed automatically, you can't manipulate it manually. Unfortunately, if you want to manage the context manually, you fail to get automatic management, which means you cannot retrieve the workflow instance ID from the context as I showed previously.

To deal with this mismatch, you handle each case separately. For the initial call to a workflow, you use a new proxy, but for all subse-

Figure 7 **Hosting the Host Service**

```
ServiceHost appHost = new ServiceHost(new HostService());
appHost.AddServiceEndpoint(“Contracts.IHostInterface”,
LocalBinding, HostAddress);

try
{
 appHost.Open();
}
catch (Exception ex)
{
 appHost.Abort();
 MessageBox.Show(String.Format(
  “There was an error hosting the local service: {0}”,
  ex.Message));
}
```

Figure 8 **Disabling Automatic Context Management**

```
App a = (App)Application.Current;

if (updateProxy == null)
{
 if (factory == null)
  factory = new ChannelFactory<IWorkflowInterface>(
   a.LocalBinding, a.WFAddress);

 updateProxy = factory.CreateChannel();
 IContextManager mgr =
  ((IClientChannel)updateProxy).GetProperty<IContextManager>();
 mgr.Enabled = false;
 ((IClientChannel)updateProxy).Open();
}
```

Figure 9 **Using OperationContextScope**

```
using (OperationContextScope scope =
 new OperationContextScope((IContextChannel)proxy))
{
 ContextMessageProperty property = new ContextMessageProperty(
  new Dictionary<string, string>
  {
   {"instanceId", wfIdText.Text}
  });

 OperationContext.Current.OutgoingMessageProperties.Add(
  "ContextMessageProperty", property);

 proxy.UpdateOrder(
  new Order
    {
     CustomerName = "Matt",
     OrderID = 2,
     OrderTotal = 250.00,
     OrderStatus = "Updated"
    });
}
```

quent calls to an existing workflow instance, you use a single client proxy and manage the context manually.

For the initial call, you should use a single ChannelFactory<TChannel> to create all the proxies. This results in better performance because the creation of the ChannelFactory has some overhead you do not want to duplicate for every first call. Using code like that shown earlier in **Figure 5**, you can use a single ChannelFactory<TChannel> to create the initial proxy. In your calling code, after using the proxy, you should follow the best practice of calling the Close method to release the proxy.

For making subsequent calls, you need to manage the context yourself, and this entails using WCF client code that is not as frequently used by developers.

This is standard WCF code for creating your proxy using the channel factory method. Because the binding is a context binding, you get automatic context management by default, which means you can extract the workflow instance identifier from the context after making the first call to the workflow.

For making subsequent calls, you need to manage the context yourself, and this entails using WCF client code that is not as frequently used by developers. To set the context manually, you need to use an OperationContextScope and create the MessageContextProperty yourself. The MessageContextProperty is set on

Figure 10 Service Implementation with INotifyPropertyChanged

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
internal class HostService : IHostInterface, INotifyPropertyChanged
{
 public void OrderStatusChange(Order order, string newStatus,
   string oldStatus)
 {
  CurrentMessage = String.Format("Order status changed to {0}",
    newStatus);
 }

 private string msg;

 public string CurrentMessage {
 get { return msg; }
 set
  {
   msg = value;
   if (PropertyChanged != null)
    PropertyChanged(this, new PropertyChangedEventArgs(
      "CurrentMessage"));
  }
 }

 public event PropertyChangedEventHandler PropertyChanged;
}
```

the message as it is being sent, which is equivalent to using the IContextManager to set the context, with the exception that using the property directly works even when the context management is disabled. **Figure 8** shows the code to create the proxy using the same ChannelFactory<TChannel> that was used for the initial proxy. The difference is that in this case, the IContextManager is used to disable the automatic context management feature and a cached proxy is used rather than creating a new one on each request.

Once the proxy is created, you need to create an Operation-ContextScope and add the MessageContextProperty to the outgoing message properties on the scope. This enables the property to be included on the outgoing messages during the duration of the scope. **Figure 9** shows the code to create and set the message property using the OperationContextScope.

This might seem like quite a bit of work just to talk between the host and the workflow. The good news is that much of this logic and the management of identifiers can be encapsulated in a few classes. However, it does involve coding your client in a particular way to ensure that the context is managed correctly for those cases in which you need to send more than one message to the workflow instance. In the code download for this article, I have included a sample host for a workflow using local communications that attempts to encapsulate much of the complexity, and the sample application shows how to use the host.

A Word About User Interface Interaction

One of the main reasons you send data from the workflow to the host is that you want to present it to a user in the application interface. Fortunately, with this model you have some options to take advantage of user interface features, including data binding in WPF. As a simple example, if you want your user interface to use data binding and update the user interface when data is received from the workflow, you can bind your user interface directly to the host's service instance.

The key to using the service instance as the data context for your window is that the instance needs to be hosted as a singleton. When you host the service as a singleton, you have access to the instance and can use it in your UI. The simple host service shown in **Figure 10** updates a property when it receives information from the workflow and uses the INotifyPropertyChangedInterface to help the data binding infrastructure pick up the changes immediately. Notice the ServiceBehavior attribute indicating that this class should be hosted as a singleton. If you look back to **Figure 7**, you can see the ServiceHost instantiated not with a type but with an instance of the class.

To databind to this value, the DataContext of the window, or a particular control in the window, can be set with the instance. The instance can be retrieved by using the SingletonInstance property on the ServiceHost class, as shown here:

```
HostService host = ((App)Application.Current).appHost.SingletonInstance
as HostService;
 if (host != null)
  this.DataContext = host;
```

Now you can simply bind elements in your window to properties on the object, as shown with this TextBlock:

```
<TextBlock Text="{Binding CurrentMessage}" Grid.Row="3" />
```

As I said, this is a simple example of what you can do. In a real application, you likely would not bind directly to the service instance but instead bind to some objects to which both your window and the service implementation had access.

Looking Ahead to WF4

WF4 introduces several features that will make local communications over WCF even easier. The primary feature is message correlation that does not rely on the protocol. That is, the use of a workflow instance identifier will still be an option, but a new option will enable messages to be correlated based on the content of the message. So, if each of your messages contain an order ID, a customer ID or some other piece of data, you can define correlations between those messages and not have to use a binding that supports context management.

Additionally, the fact that both WPF and WF build on the same core XAML APIs in .NET Framework Version 4 might open up some interesting possibilities for integrating the technologies in new ways. As we get closer to the release of .NET Framework 4, I will provide more details on integrating WF with WCF and WPF, along with other content on the inner workings of WF4. ■

MATT MILNER *is a member of the technical staff at Pluralsight, where he focuses on connected systems technologies (WCF, Windows Workflow Foundation, BizTalk, "Dublin," and the Azure Services Platform). Matt is also an independent consultant specializing in Microsoft .NET application design and development. He regularly shares his love of technology by speaking at local, regional and international conferences such as Tech·Ed. Microsoft has recognized Milner as an MVP for his community contributions around connected systems technology. You can contact him via his blog at pluralsight.com/community/blogs/matt.*

Windows Web Services

One of the main reasons many developers flocked to the Microsoft .NET Framework, and Java to a lesser degree, was the fact that it made it much easier to write software for the Internet. Whether you were writing an HTTP client or server application, the .NET Framework had you covered with classes for making HTTP requests and processing XML easily. You could even generate SOAP clients from WSDL documents and implement SOAP servers with ASP.NET. As the standards around Web services matured, Microsoft developed the Windows Communications Foundation (WCF), also built on the .NET Framework, to make it easier to use the increasingly complex Web standards for handling different transports, such as TCP and UDP, and provide more versatile security options.

Solutions that help C++ developers are a key part of Microsoft's strategy for the Windows platform.

C++ developers, however, were left wondering whether it was even practical to use C++ to write Web applications. Microsoft had provided a couple of temporary solutions in the form of ATL classes and a COM-based toolkit, but in the end these couldn't keep up with the progress that the managed SOAP stacks had made and thus were largely abandoned.

Despite the seemingly single-minded focus on the .NET Framework, developers at Microsoft haven't forgotten about the C++ developer. In fact, many of them are still and will continue to be passionate C++ developers. As a result, solutions that help C++ developers are a key part of Microsoft's strategy for the Windows platform. Of course, many of the APIs targeted at C++ developers also end up underpinning many of the managed frameworks.

Send your questions and comments to mmwincpp@microsoft.com.

Code download available at msdn.microsoft.com/msdnmag/kerr1009.aspx.

Although there are too many to enumerate here, a few are worth mentioning. The Windows HTTP Services (WinHTTP) API provides a powerful and flexible solution for writing HTTP clients. You can read more about WinHTTP in my August 2008 column. The HTTP Server (HTTP.sys) API provides the foundations for building high-performance HTTP servers without relying on a full-fledged Web server like Internet Information Services (IIS). In fact, IIS itself is based on this same API. And of course the XmlLite API provides a small and fast XML parser for native code. You can read more about XmlLite in my April 2007 feature article.

Given all of this, it is still quite a daunting task to write a SOAP client or server. Although SOAP started off "simple" (that's what the "S" stands for), it didn't stay that way for long. WinHTTP, HTTP.sys and XmlLite may get you a long way by handling the HTTP transport and parsing the XML, but there is still a ton of code to write to handle the communications layer: things like formatting and interpreting SOAP headers, not to mention supporting other transports like TCP or UDP. Even if you could somehow manage all of this, you're still left with parsing SOAP envelopes instead of being able to treat logical SOAP operations as function calls.

Well, these headaches are a thing of the past. With the introduction of the Windows Web Services (WWS) API, C++ developers no longer have to think of themselves as second-class citizens in the world of Web Services. WWS is designed from the ground up to be a completely native-code implementation of SOAP, including support for many of the WS-* protocols. WWS is, strictly speaking, exposed

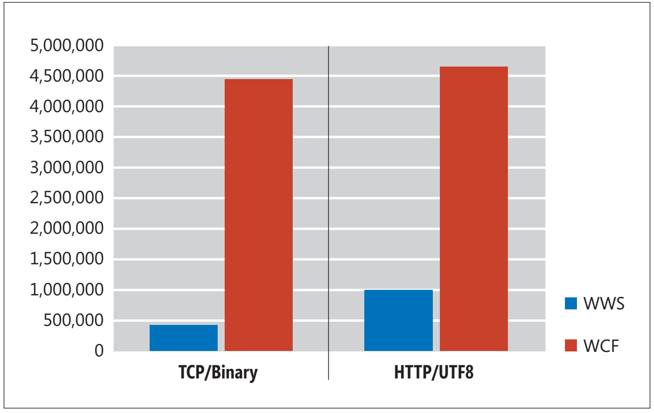


Figure 1 **Comparing Client Working Set (Lower Is Better)**

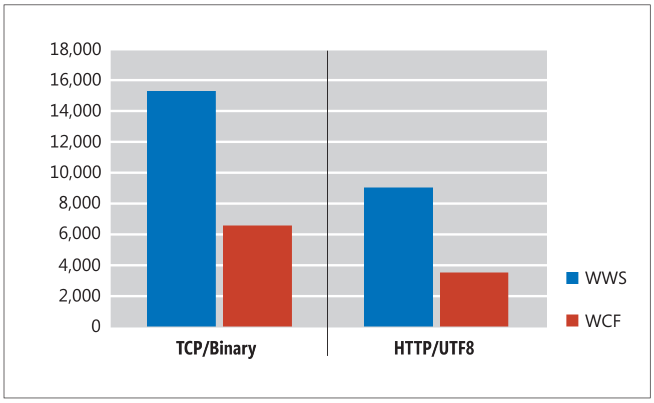


Figure 2 **Comparing Server Throughput (Higher Is Better)**

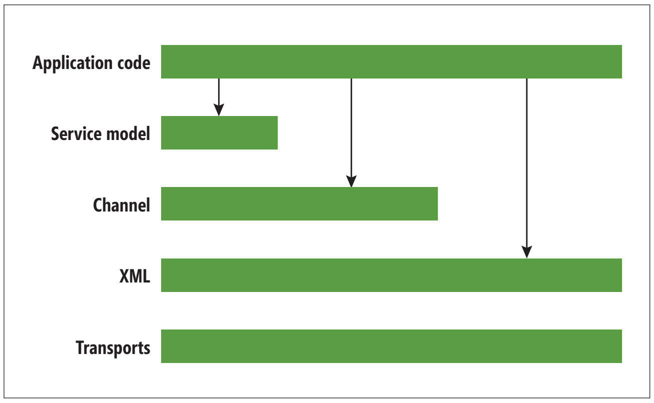


Figure 3 **Layered API**

through a C API, making interoperability with other languages and runtimes very straightforward, but it is the C++ developer who will likely benefit the most. In fact, with a little help from C++, it can be a real joy to use—as you shall see in this article.

Architecture and Principles

WWS embodies everything that .NET Framework-based libraries are not. It is designed for native code. It is designed to introduce a minimal number of dependencies. It is designed to use as little memory as possible. And it is designed to be fast. Really fast. The team responsible for developing WWS runs performance tests on each new build, comparing it with WCF and RPC. RPC is used as a sort of baseline since nothing could be faster, but it does provide a reliable way of tracking speed regressions. It is, however, illuminating when you compare WCF and WWS. **Figure 1** shows a comparison of the working set for a client using WCF and WWS respectively. That's a pretty dramatic margin, but perhaps not that surprising when you think that the .NET Framework is involved. **Figure 2**, however, should be surprising if you, like many others, consider WCF to be state of the art. It shows the throughput in operations per second for a server using WCF and WWS respectively. WWS is more than twice as fast! Don't get me wrong: There is nothing wrong with WCF or the .NET Framework, but when you need something small and fast, it's hard to beat C++ and native code. But you know that already!

The WWS runtime is packaged in WebServices.dll, which is included with Windows 7 and Windows Server 2008 R2. It is also available as a system update for Windows XP and later. Functions exported from WebServices.dll represent the WWS API and you can gain access to them by linking to WebServices.lib and including the WebServices.h header file from the Windows SDK. So far,

Figure 4 **Web Service Definition**

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="ttp://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://calculator.example.org/"
 targetNamespace="http://calculator.example.org/">

 <wsdl:types>
   <xsd:schema elementFormDefault="qualified"
    targetNamespace="http://calculator.example.org/">
    <xsd:element name="AddRequest">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="first" type="xsd:double" />
       <xsd:element name="second" type="xsd:double" />
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
    <xsd:element name="AddResponse">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="result" type="xsd:double" />
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:schema>
  </wsdl:types>

  <wsdl:message name="AddRequestMessage">
    <wsdl:part name="parameters" element="tns:AddRequest" />
  </wsdl:message>
  <wsdl:message name="AddResponseMessage">
    <wsdl:part name="parameters" element="tns:AddResponse" />
  </wsdl:message>

  <wsdl:portType name="CalculatorPort">
    <wsdl:operation name="Add">
      <wsdl:input message="tns:AddRequestMessage" />
      <wsdl:output message="tns:AddResponseMessage" />
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="CalculatorBinding" type="tns:CalculatorPort">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Add">
      <soap:operation soapAction=
       "http://calculator.example.org/Add" style="document"/>
      <wsdl:input>
       <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
       <soap:body use="literal"/>
      </wsdl:output>
     </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="CalculatorService">
    <wsdl:port name="CalculatorPort" binding="tns:CalculatorBinding">
     <soap:address location="http://localhost:81/calculator"/>
    </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

Figure 5 **Error Object Wrapper**

```
class WsError
{
    WS_ERROR* m_h;
public:
    WsError* m_h(0)
    {}
    ~WsError()
    {
       if (0 != m_h)
    }
    HRESULT Create(const WS_ERROR_PROPERTY* properties,
            ULONG propertyCount)
    {
       return WsCreateError(properties, propertyCount, &m_h);
    }
    HRESULT GetProperty(WS_ERROR_PROPERTY_ID id, void* buffer,
               ULONG bufferSize)
    {
        return WsGetErrorProperty(m_h, id, buffer, bufferSize);
    }
    template <typename T>
    HRESULT GetProperty(WS_ERROR_PROPERTY_ID id, out T* buffer)
    {
        return GetProperty(id, buffer, sizeof(T));
    }
    HRESULT GetString(ULONG index, WS_STRING* string)
    {
        return WsGetErrorString(m_h, index, string);
    }
    operator WS_ERROR*() const
    {
      return m_h;
    }
};
```

Figure 6 **Enumerate the Strings in an Error Object.**

```
WsError error;
HR(error.Create(0, 0));
// Something goes wrong . . .
ULONG stringCount = 0;
HR(error.GetProperty(WS_ERROR_PROPERTY_STRING_COUNT,&stringCount));

for (ULONG i = 0; i < stringCount; ++i)
{
    WS_STRING string;
    HR(error.GetString(i, &string));
    wprintf(L"Error %d: %.*s\n", i, string.length, string.chars);
}
```

so good. But what does the API look like? Well, unlike COM-style APIs like that of XmlLite or Direct2D, this C API requires you to imagine a set of logical layers and objects that live behind the scenes and are just waiting to break out. Let's first take a look at it in terms of layers. **Figure 3** illustrates the layers of functionality exposed by the WWS API, with each layer building upon the one beneath it. Each layer is represented by a number of functions and structures and provides a set of abstractions. As you can guess, the application can make use of any of the layers, but for the most part you will want to stick with the service model that provides the simplest programming model and hides many of the details for you. The transports layer is just a reminder that underneath it all there will be some network protocol. WWS will use WinHTTP, HTTP.sys, or Winsock, depending on the selected transport and whether it is used to implement a client or server.

As its name suggests, the service model layer entirely abstracts the SOAP message exchanges and models the logical Web service

Figure 7 **Heap Object Wrapper**

```
class WsHeap
{
    WS_HEAP* m_h;
public:
    WsHeap() : m_h(0)
    {}
    ~WsHeap()
    {
        if (0 != m_h) WsFreeHeap(m_h);
    }
    HRESULT Create(SIZE_T maxSize, SIZE_T trimSize,
            const WS_HEAP_PROPERTY* properties,
            ULONG propertyCount,
            in_opt WS_ERROR* error)
    {
        return WsCreateHeap(maxSize, trimSize, properties, propertyCount,
                     &m_h, error);
    }
    operator WS_HEAP*() const
    {
        return m_h;
    }
};
```

operations as function calls. It does not, however, stand alone but relies on the use of a command-line tool called Wsutil.exe from the Windows SDK. Given a WSDL file, this tool will generate a header file as well as a C source file with most of the code necessary to both connect to a Web service of the given description and to implement such a Web service, taking care to ensure the channel binding is properly configured and messages are properly formatted. This is by far the simplest approach and provides a programming model that is very much like what you would expect from traditional RPC.

The error object can store a number of strings with different levels of information about an error.

The channel layer, on the other hand, exposes the messages sent and received on a particular transport, but still optionally shields you from having to actually format the messages yourself. The benefit here is that you are shielded from the particular transport and encoding that is used. The channel layer is where you control binding information and where you can secure your communication whether for authentication or privacy.

The XML layer exposes you to the formatting of messages and serialization of data. You have full access to the content of messages but are shielded from the particular encoding whether you're communicating with text, binary or MTOM. You might be surprised to hear that WWS has its own XML parser. Why doesn't it just use XmlLite? Although XmlLite is certainly lightweight and very fast, it isn't quite a perfect match for a number

Figure 8 **Service Proxy Wrapper**

```
class WsServiceProxy
{
    WS_SERVICE_PROXY* m_h;
public:
    WsServiceProxy() : m_h(0)
    {}
    ~WsServiceProxy()
    {
        if (0 != m_h)
        {
            Close(0, 0); // error
            WsFreeServiceProxy(m_h);
        }
    }
    HRESULT Open(const WS_ENDPOINT_ADDRESS* address,
            const WS_ASYNC_CONTEXT* asyncContext,
            WS_ERROR* error)
    {
        return WsOpenServiceProxy(m_h, address, asyncContext, error);
    }
    HRESULT Close(const WS_ASYNC_CONTEXT* asyncContext,
            WS_ERROR* error)
    {
        return WsCloseServiceProxy(m_h, asyncContext, error);
    }
    WS_SERVICE_PROXY** operator&()
    {
        return &m_h;
    }
    operator WS_SERVICE_PROXY*() const
    {
        return m_h;
    }
};
```

of reasons. The most obvious reason is that WWS needs to support different encodings while XmlLite supports only text. SOAP messages are also typically encoded using UTF-8, while XmlLite exposes all properties with Unicode strings and this introduces unnecessary cost when copying values. WWS also has very strict memory consumption goals (there is actually an API for this, as we'll see later on) that cannot be met with XmlLite. In the end, the WWS team was able to implement a custom parser specifically for SOAP that is considerably faster than XmlLite. Keep in mind that the WWS parser is not meant to replace XmlLite. As a general purpose XML parser, it is hard to beat, but the WWS XML layer provides developers with very specific features aimed at efficiently serializing data into and out of SOAP message using the subset of XML required by SOAP.

Apart from the functions and data structures that logically belong to these three layers, the WWS API provides a number of facilities that are common to all layers, including error handling, asynchronous completion, cancellation, memory management and more. Because I have limited space and want to help you get started quickly, I'm going to limit the rest of this article to the use of the service model for building a Web service client and server. In a future article, I'll dig more deeply into the other parts of WWS.

Getting Started

To get started, I'll use the minimal Web service definition from **Figure 4**. This WSDL document defines the types, messages, operations, endpoints and channel bindings of the service. The first thing to do is run it through Wsutil.exe as follows:

```
Wsutil.exe Calculator.wsdl
```

This will produce a header file called Calculator.wsdl.h and a C source file called Calculator.wsdl.c.

Before we take a look at what was generated, we need to get some groundwork done, regardless of whether you're implementing the client or the server. The first thing you'll need is a way to express error information. The WWS API exposes rich error information both for its own functions as well as SOAP faults via an error object. Being a C API, this object is represented by an opaque handle and a set of functions. In this case, WS_ERROR* represents a handle to an error object and WsCreateError is the function that creates it. The object is freed by calling the WsFreeError function. The error object can store a number of strings with different levels of information about an error. To retrieve these, you need to first determine how many strings are present. This is done by calling the WsGetErrorProperty function, giving it the handle to the error object and specifying the WS_ERROR_PROPERTY_STRING_COUNT constant. Armed with this information, you call the WsGetErrorString function, giving it the handle to the error object as well as the zero-based index of the string to retrieve. You can also use the API functions to populate your own error objects. Naturally, a little C++ will go a long way to making this simpler and more reliable. **Figure 5** provides a simple error object wrapper. You can easily enumerate the strings in an error object, as shown in **Figure 6**.

Figure 9 **Create and Open Service Proxy**

```
WsServiceProxy serviceProxy;

HR(CalculatorBinding_CreateServiceProxy(0, // template value
                                         0, // properties
                                         0, // property count
                                         &serviceProxy,
                                         error));

WS_ENDPOINT_ADDRESS address =
{
    {
        static_cast<ULONG>(wcslen(url)),
        const_cast<PWSTR>(url)
    }
};

HR(serviceProxy.Open(&address,
                     0, // async context
                     error));
```

The next thing we'll need is a heap object. The heap object provides precise control over memory allocation when producing or consuming messages and when needing to allocate various other API structures. It also simplifies the programming model since there is no need for you to know precisely how much memory is required for any particular function to succeed. Many older functions in the Windows SDK will, for example, either require you to guess how much storage is required or to first call a function in

Figure 10 **CalculatorBinding_CreateServiceEndpoint**

```
WsServiceProxy serviceProxy;
const WS_STRING address =
{
    static_cast<ULONG>(wcslen(url)),
    const_cast<PWSTR>(url)
};

CalculatorBindingFunctionTable functions =
{
    AddCallback
};

WS_SERVICE_ENDPOINT* endpoint = 0;

HR(CalculatorBinding_CreateServiceEndpoint(0, // template value
                                           &address,
                                           &functions,
                                           0, // authz callback
                                           0, // properties
                                           0, // property count
                                           heap,
                                           &endpoint,
                                           error));
```

a particular way to determine how much memory you should allocate for the function to succeed. The use of the WWS heap object removes all this extra coding and provides a nice way to control the memory usage of the API. This also comes in handy from a security perspective where you may want to specify expected limits on how much the API may allocate. WS_HEAP* represents a handle to a heap object and WsCreateHeap is the function that creates it. The object is freed by calling the WsFreeHeap function. Once it's created, you can allocate memory from the heap using the WsAlloc function, but for this article we'll just pass the handle to the heap object to other API functions for them to use as necessary.

Figure 7 provides a simple heap object wrapper. Given this, you can create a heap object limited to 250 bytes as follows:

```
WsHeap heap;

HR(heap.Create(250, // max size
               0, // trim size
               0, // properties
               0, // property count
               error));
```

Notice how I'm passing the error object to the heap object's Create method. Should anything go wrong while creating the heap object, I'll be able to interrogate the error object to find out the cause.

The Client

The client-side of the service model centers on the service proxy object. The generated source code includes a function called CalculatorBinding_CreateServiceProxy. The name is derived from that of the endpoint or binding name defined in the WSDL document. This function creates a service proxy object and returns a WS_SERVICE_PROXY* representing an opaque handle to that object. The object is freed by calling the WsFreeServiceProxy function. Once created, your application can open the service endpoint using the WsOpenServiceProxy function and then make calls via the service proxy to the Web service. What exactly WsOpenServiceProxy does is dependent on the transport being used. You must also take care to close the service proxy prior to freeing the object using the WsCloseServiceProxy function. Naturally, all of this housekeeping can be nicely wrapped up in a simple class provided in **Figure 8**. Given this, you can create and open the service proxy, as shown in **Figure 9**.

Figure 11 **Service Host Wrapper**

```
class WsServiceHost
{
    WS_SERVICE_HOST* m_h;
public:
  WsServiceHost() : m_h(0)
  {}
  ~WsServiceHost()
  {
    if (0 != m_h)
    {
        Close(0, 0);
        WsFreeServiceHost(m_h);
    }
  }
  HRESULT Create(const WS_SERVICE_ENDPOINT** endpoints,
          const USHORT endpointCount,
          const WS_SERVICE_PROPERTY* properties,
          ULONG propertyCount, WS_ERROR* error)
  {
     return WsCreateServiceHost(endpoints, endpointCount, properties,
                   propertyCount, &m_h, error);
  }
  HRESULT Open(const WS_ASYNC_CONTEXT* asyncContext, WS_ERROR* error)
  {
    return WsOpenServiceHost(m_h, asyncContext, error);
  }
  HRESULT Close(const WS_ASYNC_CONTEXT* asyncContext, WS_ERROR* error)
  {
    return WsCloseServiceHost(m_h, asyncContext, error);
  }
  operator WS_SERVICE_HOST*() const
  {
    return m_h;
  }
};
```

The WS_ENDPOINT_ADDRESS structure is used to address the messages being sent. You will typically use this structure when programming at the channel layer. In this case, we populate only the URL portion and the service proxy takes care of the rest.

At this point, we can use another generated function namely CalculatorBinding_Add, which unsurprisingly represents the Web service's Add operation. It really doesn't get much easier than this:

```
const double first = 1.23;
const double second = 2.34;
double result = 0.0;

HR(CalculatorBinding_Add(serviceProxy,
                         first,
                         second,
                         &result,
                         heap,
                         0, // properties
                         0, // property count
                         0, // async context
                         error));

ASSERT(3.57 == result);
```

Once you are done interacting with the Web service you just need to close the service proxy's communication channel:

```
HR(serviceProxy.Close(0, // async context
                      error));
```

The Server

While the client-side programming model centers on the service proxy, the server instead creates and manages a service host, which provides the necessary runtime to listen on the various endpoints based on the provided channel information. Again, because we're using the service model, most of the details are abstracted away, and we're left only to create the service endpoint and host. WWS will do the rest.

The first step is to create the service endpoint. The service model takes care of this in the form of another generated function, namely CalculatorBinding_CreateServiceEndpoint, as **Figure 10** shows. Creating the endpoint requires specifying the address on which the endpoint is going to listen. This is provided by the WS_STRING structure, which is a length-prefixed Unicode string and is not required to be null terminated. Because the endpoint is responsible for fulfilling requests, you need to provide a table of function pointers mapping to the operations exposed by the service. The generated CalculatorBindingFunctionTable structure is used for this purpose. Finally, the endpoint itself is represented by the WS_SERVICE_ENDPOINT structure and is allocated in the provided heap.

The next step is to create the service host. WS_SERVICE_HOST* represents a handle to a service host object and WsCreateServiceHost is the function that creates it. The object is freed by calling the WsFreeServiceHost function. The WsCreateServiceHost function creates the service host object given a list of endpoints. At this point, you can start the listeners on all the endpoints using the WsOpenServiceHost function. Stop the communication using the WsCloseServiceHost function. As with the service proxy, be sure to close the service host prior to freeing it. Again, all of this housekeeping can be nicely wrapped up in a simple class provided in **Figure 11**. Given this, you can start the Web service as follows:

```
const WS_SERVICE_ENDPOINT* endpoints[] = { endpoint };

WsServiceHost serviceHost;

HR(serviceHost.Create(endpoints,
                      _countof(endpoints),
                      0, // properties
                      0, // property count
                      error));

HR(serviceHost.Open(0, // async context
                    error));
```

In this case, there is only a single endpoint but you can see how easy it would be to add additional endpoints to the service host. When it is time to stop the service, you just need to close the service host's communication channels.

```
HR(serviceHost.Close(0, // async context
                     error));
```

Actually implementing the Web service operation is about the easiest part:

```
HRESULT CALLBACK AddCallback(__in const WS_OPERATION_CONTEXT*,
                 __in double first,
                 __in double second,
                 __out double* result,
                 __in_opt const WS_ASYNC_CONTEXT* /*asyncContext*/,
                 __in_opt WS_ERROR* /*error*/)
{
    *result = first + second;
    return S_OK;
}
```

The signature for the AddCallback function is also provided by the generated source code, should you have any doubts about how to specify it.

And that's all I have space for this month, but you should now have a good idea of the features and benefits that the Windows Web Services API has to offer. As you can see, C++ developers finally have a modern SOAP stack right out of the box. It offers the best possible performance and memory usage and is a pleasure to use with a little help from C++. ■

Who's Using It?

Various teams at Microsoft have already started adopting the Windows Web Services (WWS) API within their own products or technologies. In many cases, this replaces custom-built SOAP stacks and some even chose to replace commercial implementations like Windows Communication Foundation (WCF) with WWS. Here are just a few examples.

The Microsoft Web Services on Devices (WSD) API enables developers to write clients and servers based on the Devices Profile for Web Services (DPWS). In Windows 7, the WSD API has started using WWS for the creation and canonicalization of XML in SOAP messages that the WSD runtime sends out. The WSD team plans to expand their use of WWS as they add new features and refactor existing code.

Windows CardSpace is Microsoft's implementation of an identity system based on Web service standards. Originally implemented with WCF, it is being rewritten using native code and WWS to meet very strict business requirements on the size of the downloadable installer and the runtime working set.

The Microsoft Forefront Threat Management Gateway (TMG) is a security platform providing firewall and caching features to secure and improve the performance of networks. Its URL filtering feature uses WWS to connect to the Microsoft Reputation Service to categorize URLs.

Finally, the Windows Public Key Infrastructure (PKI) client provides automatic lifecycle management of certificates with auto enrollment as well as user and application driven certificate enrollment. Windows 7 introduces a new set of Web services that allow for certificate enrollment to be completed without the traditional limitations of LDAP and DCOM. The PKI client makes use of WWS for all operations, including a new Certificate Enrollment Policy (MS-XCEP) protocol and a WS-Trust extension for certificate enrollment (MS-WSTEP). The WWS client communicates with a new set of Active Directory Certificate Services in Windows Server 2008 R2 that are implemented with WCF, as well as with Web services provided by public certificate issuers.

KENNY KERR *is a software craftsman specializing in software development for Windows. He has a passion for writing and teaching developers about programming and software design. You can reach Kerr at weblogs.asp.net/kennykerr.*

THANKS to the following technical expert for reviewing this article: *Nikola Dudar*