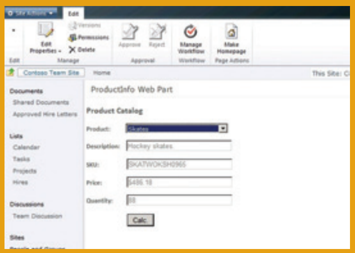
Visual Studio 2010 for SharePoint ◆ Open XML and Content Controls ◆ Event Tracing for Windows

MSDN Magazine

OCTOBER 2009 Vol 24 No 10

The Microsoft Journal for Developers

msdn® magazine



ProductInfo Web Part

Product Catalog

Product: Skates
Description: Hockey skates
SKU: SKATWOKSH0965
Price: $486.18
Quantity: 88

OCTOBER 2009 VOL 24 NO 10

msdn

Microsoft®

Microsoft msdn magazine OCTOBER 2009 VOLUME 24 NUMBER 10

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

EDITOR'S NOTE

The Capability-Delivery Business

In this month's issue of *MSDN Magazine*, we begin taking a look at some of the new features and tools in Visual Studio 2010 and Microsoft .NET Framework 4 that will continue improving the experience of creating business applications that run on the Microsoft Office platform. This is an area of software development that I have always believed holds tremendous potential in terms of ensuring that we are focused on solving the right problems—those being the business problems. And with each new release, it's nice to see improvements made to the platform rather than just to the applications that help to realize this potential. We still have a good distance to travel, but we're getting closer. For example, Steve Fox provides a first look at SharePoint tooling in Visual Studio 2010 (page 44), while Eric White walks through techniques for dynamically generating documents from SharePoint using XML content controls (page 52).

Office productivity applications have been around for a while now—after all, wasn't office productivity one of the main drivers behind the personal computer revolution? However, if you look at the features that have been added to various products as they have matured, with a couple notable exceptions such as OLE, it seems as though at the core, office applications haven't really changed all that much since their inception. For example, I'm writing this document using Word 2010, and while I really like the new version (a lot, actually), in the end, it is still a word processing application that I use in much the same way that I used WordPerfect 5.1 back around 1990. It seems as though these applications help us perform the same kinds of tasks more efficiently—rather than change the tasks that we need to perform. However, looking at the current and emerging versions of the Office platform, I think that this is now changing in a dramatic (and much needed) way.

This larger change is the result of two major categories of smaller shifts in the Office platform itself. First, there is an increased focus around the end-to-end collaborative experiences and business workflows rather than simply on tasks such as creating a document or e-mailing a file. As products like SharePoint continue to mature alongside the more long-standing Office client products, I'm confident that these kinds of experiences will only continue to deepen. Second, there seems to be a continued focus on data and more specifically on providing the capabilities to extract intelligence from that data. To group both of these thoughts together in a single philosophical statement, I would say that traditional office client applications are less about being really good stovepipe applications and more about being a rich presentation tier for a deeply interconnected network of systems in an enterprise architecture. To use the iceberg analogy, the office client applications simply flatten the tip a bit so that the iceberg is more comfortable to sit on.

To get to the next step of really transforming the manner in which business is conducted, highly domain-specific functionality must be developed below the metaphorical surface—and thus can only be meaningfully implemented by you. When you think about it, the Office platform lays a tremendous foundation for providing these sorts of meaningful capabilities—what other platform gets you halfway there out of the box with respect to meeting your users' expectations around things like user experience? We, as developers, simply need to adjust our mission a bit: we aren't in the application-development business as much as we are in the capability-delivery business. Seeing our role through that lens will, I believe, help to sharpen our focus on whom we build software for, what those people really need and how they can best use it. And in a majority of cases, the people that need the capabilities we develop can best leverage them when they are delivered in an integrated, frictionless manner, via the productivity tools that they are already using.

[illegible]

THANKS TO THE FOLLOWING MICROSOFT TECHNICAL EXPERTS FOR THEIR HELP WITH THIS ISSUE: *Kent Brown, Chance Coble, Steve Fox, Kit George, John Langdon, Don McGrady, Scott Morrison, Mike Morton, Boris Rivers-Moore and Stephen Toub.*

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

User Interfaces, Podcasts, Object-Relational Mappings and More

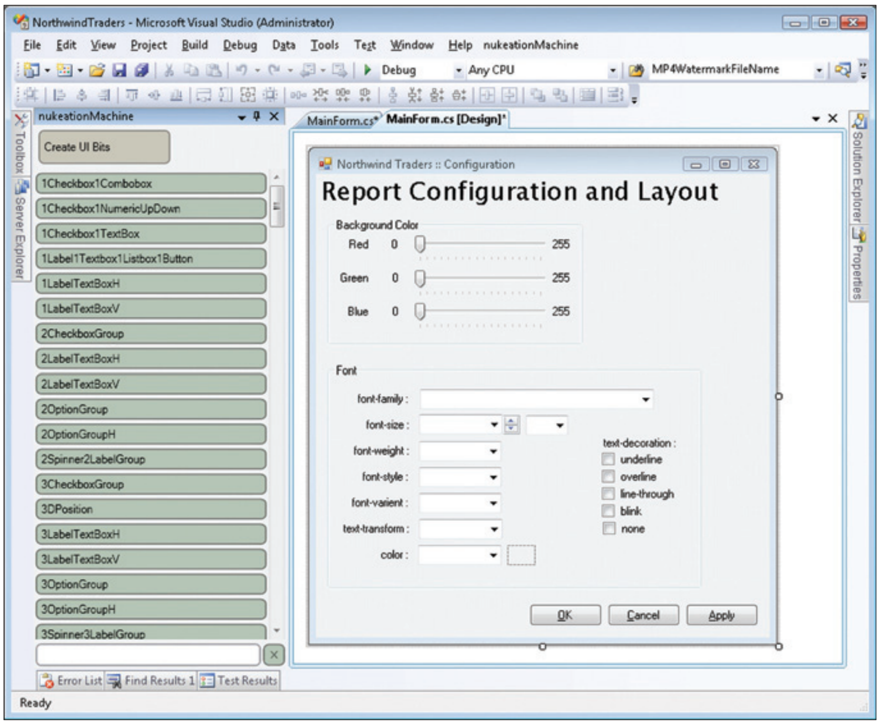
Quickly Create Common User Interface Groupings

I recently started work on an order-management application for the printing operations of a large university. Each day, hundreds of professors and administrators submit production orders ranging from 100 copies of an upcoming exam to 50,000 copies of next semester's course catalog. Like most line-of-business applications, this one requires a variety of data-entry screens, each one containing dozens of user-input elements. Many of these screens share common user interface groupings. For example, the order placement screen contains a series of drop-down lists from which the user can select the paper type, the binding, and other options. This same grouping of drop-down lists is repeated in the order management screen. Many screens also share common button groupings, such as Save Progress, Submit, and Cancel.

Implementing such user interfaces, one button and one drop-down list at a time, is a tedious and time-consuming endeavor—but it doesn't have to be that way. **nukeationMachine**, by Nukeation Studios, can greatly reduce the time it takes to build user interfaces for Windows Presentation Foundation (WPF), WinForms, and ASP.NET applications. nukeationMachine is a Visual Studio add-in that ships with over 1,600 UI Bits—common groupings of user interface elements that can be added to the design surface in Visual Studio with a single click of the mouse. For instance, there is a UI Bit named IgnoreRetryCancel that defines a grouping of three buttons laid out horizontally and titled Ignore, Retry and Cancel. To add this grouping to your application, simply locate the UI Bit in the nukeationMachine window and click it, and those three buttons are instantly added to the design surface. To better appreciate the time savings, check out the screenshot, which shows a user interface I designed with nukeationMachine in under 30 seconds.

You can think of nukeationMachine's UI Bits as macros that define the elements to add to the design surface; selecting the UI Bit from the nukeationMachine window simply executes the macro, adding those elements to the design surface. Consequently, using nukeationMachine to implement your user interface does not add any code to your project or any dependency to nukeationMachine. It's just a lightning-fast way to add common UI groupings. What's more, if you have a UI grouping that's not already included in the 1,600 UI Bits that ship with nukeationMachine, you can package your custom grouping into a UI Bit with just a few mouse clicks.

You can buy a Single Technology Edition of nukeationMachine, which supports UI Bits for only one technology—WPF, WinForms, or ASP.NET—for $140. The



A Sample UI Design Using nukeationMachine

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.

Complete Edition includes support for all three and costs $280.

Price: $140 for a Single Technology Edition; $280 for the Complete Edition

nukeation.com

.NET Podcasts

Podcasts are audio or video productions that are available for download over the Internet and are typically played on portable MP3 players. They are ideal listening material for your daily commute, for airplane flights, or for other downtimes. The podcast format allows for interactive discussions among multiple participants as well as the opportunity to explore a specific topic in more depth and in a more conversational way than is possible in a blog entry or magazine article.

There are an increasing number of high-quality, developer-focused podcasts, such as Scott Hanselman's Hanselminutes podcast, which was reviewed in the July 2009 issue (msdn.microsoft.com/magazine/dd943051.aspx). Another excellent podcast is **.NET Rocks**, a weekly talk show hosted by Carl Franklin and Richard Campbell. Each episode is about an hour long and focuses on a particular technology, product, or person of interest. For instance, in show #370 the duo interviewed Microsoft Senior Vice President S. Somasegar. Phil Haack discussed the ASP.NET MVC Framework in show #433. And author Julia Lerman shared her insights on the Entity Framework in show #319.

Franklin and Campbell are both natural interviewers and do a great job picking the brains of their guests while maintaining a smooth and natural flow to the discussion. Listening to a .NET Rocks podcast is a lot like overhearing a conversation among experienced developers at a user group meeting or conference. You're bound to learn something new, hear an interesting anecdote or two, and discover how other knowledgeable developers are using .NET and related technologies in their daily jobs.

The .NET Rocks show was originally started by Franklin in 2002, and there are now more than 460 episodes. From the show's Web site you can download previous shows (as well as each week's new episode), view complete transcripts, and send in your questions or comments.

Price: Free

dotnetrocks.com

Configure NHibernate Using C#

NHibernate is a popular object-relational mapper (ORM) for .NET applications. Like other ORM tools, NHibernate provides a framework for persisting and retrieving data between an object-oriented domain model and a relational database, freeing developers from having to write tedious data-access code. (NHibernate was reviewed in the October 2006 issue, available at msdn.microsoft.com/magazine/cc163540.aspx.) For NHibernate to work, a mapping between the domain model and database must be defined. Such mappings have traditionally been defined through XML files, but these mappings have some drawbacks. For instance, there are no compile-time checks to ensure that the XML mapping and the properties in the domain model line up; if there is a mismatch, it is not caught until runtime. And for many developers, XML's syntax and verbosity can seem excessive and less readable than clean, terse source code.

Fluent NHibernate is an interesting open-source project that makes it possible to move the NHibernate mappings out of XML and into source code. Each entity is mapped by creating a class and defining the mapping rules in its constructor. **Figures 1** and **2** show Fluent NHibernate in action. **Figure 1** contains the class definitions for two objects in the domain model, Category and Product. **Figure 2** shows the CategoryMap class, which defines the mapping rules between the Category object and the underlying Categories database table. Be sure to note the syntax in **Figure 2**. It is an example of a fluent interface, an API design style that aims at maximizing readability through the use of descriptive names and method chaining. With the NHibernate mapping rules in code, any misalignments between the mapping rules and the domain model are caught at compile time. For example, altering the Category class by renaming its CategoryId property to Id would result in a compile-time error in the CategoryMap class.

Figure 1 **Category and Product Class Definitions**

```
public class Category
{
  public int CategoryId { get; private set; }
  public string CategoryName { get; set; }
  public string Description { get; set; }
  public IList<Product>
    Products { get; private set; }
}

public class Product { ... }
```

Figure 2 **CategoryMap Class**

```
public class CategoryMap : ClassMap<Category>
{
    public CategoryMap()
    {
        WithTable("Categories");
        Id(x => x.CategoryId);
        Map(x => x.CategoryName)
          .WithLengthOf(50)
          .Not.Nullable();
        Map(x => x.Description);
        HasMany(x => x.Products)
          .Cascade.All()
          .WithTableName("Products");
    }
}
```

Show #459 | 6/30/2009 (59 minutes)
Michael Stiefel on Cloud Computing

Michael Stiefel is back to talk about the realities of Cloud Computing including offerings from Amazon, Google, and Microsoft Azure.

Michael Stiefel, principal of Reliable Software, Inc. is a consultant on software architecture and development, and the alignment of information technology with business

This episode is brought to you by

.NET Rocks Podcasts Web Site

If the structure of your object and relational models follows a set of conventions defined by Fluent NHibernate, you can use its auto-mapping feature, which removes the need to explicitly spell out the mapping rules in code. Instead, Fluent NHibernate automatically determines the mappings on your behalf. What's more, Fluent NHibernate includes capabilities to move database configuration information out of NHibernate's XML files and into source code using the same fluent interface style.

Price: Free

FluentNHibernate.org

The Bookshelf

Working as a consultant can be an exciting, fast-paced career choice. Unlike full-time staff, consultants are usually brought on board to solve a particular problem or implement a specific technology. When that's done, you're off to a new client and a new job, and, perhaps, new technologies. Of course, consulting has its drawbacks, the main one being job security—consultants are easier to let go than full-time employees.

If you currently work for a technology consulting company, or are weighing the pros and cons of doing so, then Aaron Erickson's book "The Nomadic Developer" (Addison-Wesley, 2009) is for you. This book is divided in two: the first half gives prospective consultants an insider's look at the consulting business; the second is geared toward those already employed by a consulting firm and explores various career paths and offers advice on how to best thrive as a consultant.

"The Nomadic Developer" should be required reading for anyone currently seeking a job at a consulting company, especially those who have not previously worked as consultants. The book's second chapter, "The Seven Deadly Firms," describes seven dysfunctional traits that, if present, will negatively impact your time at the company. For each dysfunction, Erickson supplies a detailed description, explains what life is like at a consulting firm when that trait is present, and provides tips for spotting the dysfunction during a job interview. There are also chapters on the top 10 traits a technology consulting firm is looking for in applicants, and another chapter that suggests questions applicants should ask during the interview process.

In addition to Erickson's own viewpoints, "The Nomadic Developer" includes insights and anecdotes from other experienced consultants in the form of annotations and in a chapter of essays. These annotations and essays, along with Erickson's conversational writing style, make "The Nomadic Developer" an enjoyable and educational read.

Price: $39.99

nomadic-developer.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.*

Profiling the .NET Garbage-Collected Heap

In the *MSDN Magazine* June 2009 article "Memory Usage Auditing for .NET Applications" (msdn.microsoft.com/magazine/dd882521.aspx), we discussed monitoring memory consumption using tools like Task Manager, PerfMon and VADump. These tools help in monitoring the overall memory consumption of your application. Typically, if a .NET application is consuming a large amount of memory, it is either because the application loads a large number of DLLs or the application is creating a large number of long-lived objects on the .NET garbage-collected (GC) heap. If an application loads many DLLs, the only recourse (other than to avoid unnecessary dependencies) is to run less code. However, if an application suffers from a large GC heap footprint, one could identify that the GC heap was contributing significantly to memory consumption, as discussed in the June article. In this article, we complete the picture for GC-heap-related memory issues by providing step-by-step instructions on using the CLR Profiler for .NET GC heap memory investigations.

Recap: Audit All Applications

The .NET runtime abstracts certain operations (such as allocating memory or using a large class library), and it is not always simple to foresee the impact of these operations on application performance. This is especially true for memory. The important takeaway point in the last article was that every application deserves a periodic memory audit. This article goes through this step-by-step using a real program as our example (as opposed to a demo). This was a simple program, and we could have chosen not to audit the application's memory usage. However, when we completed a memory audit, the small effort of performing the audit paid off handsomely, as we found significant memory inefficiencies.

The program we use in this article is called XMLView, which is a simple viewer for XML files that is highly scalable. XMLView can easily handle XML files that are gigabytes in size (try that in IE!) because this application was designed to be memory efficient from the outset. Despite the careful design, in some scenarios, tools like Task Manager showed that the application was using significantly more memory than we would have expected. This article walks you through the investigation that uncovered a performance bug.

Big-Picture Memory Usage

The first step in a memory audit is to launch the application and look at the OS task manager to check the application's overall memory usage. As we described previously, the metric that is most important is the "Memory - Private Working Set" column. This refers to the application memory that cannot be shared with other processes running on the machine (see previous article for details).

Once Task Manager is open, you can monitor the private memory usage of your application in real time, run through various user scenarios and observe any impact to memory consumption. There are two anomalies that warrant your attention. One is memory consumption that is disproportionate to the scenario or task being performed. For example, opening a menu entry or performing simple operations should not cause large spikes in memory consumption. Memory leaks are the second type of anomaly that necessitates attention. For some operations, such as opening a new file, the expectation is that new data structures will be created each time the operations is performed. However, many operations (for example, searches) do not logically allocate new long-lived memory. It is not unusual for these stateless operations to allocate memory the first time they are performed (as lazy data structures are populated). But if memory usage continues to increase with subsequent iterations of the scenario, memory is leaking and needs to be investigated. This technique of using Task Manager is by design very coarse (only large memory problems will be evident), but for many applications, only the large ones are worth pursuing.

For XMLView, the scenario is launching the viewer on a large XML file and performing various navigation operations, such as opening tree nodes or searching. This experiment initially turned up no leaks (the memory usage stabilized after a while; see **Figure 1**). Memory usage spiked rather substantially (8MB) when the search functionality was used (see **Figure 2**) from 16812KB private working set to 25648KB. This represents one of the anomalies we were looking for: the resource consumption did not fit our mental model of the program's memory behavior, because search has no auxiliary data structures and thus should not consume extra memory. We had a mystery to solve.

CLR Profiler

A number of thirdparty tools are available for doing analysis of the .NET GC heap; however, we will concentrate on CLR Profiler, which has the advantage of being available as a free Web download from

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

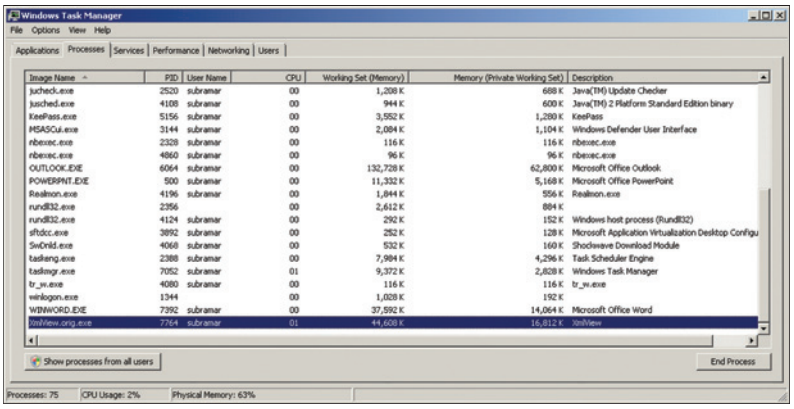


Image Name	PID	User Name	CPU	Working Set (Memory)	Memory (Private Working Set)	Description
jucheck.exe	2520	subramar	00	1,208 K	688 K	Java(TM) Update Checker
jusched.exe	4108	subramar	00	944 K	600 K	Java(TM) 2 Platform Standard Edition binary
KeePass.exe	5156	subramar	00	3,552 K	1,280 K	KeePass
MSASCui.exe	3144	subramar	00	2,084 K	1,104 K	Windows Defender User Interface
nbexec.exe	2328	subramar	00	116 K	116 K	nbexec.exe
nbexec.exe	4860	subramar	00	96 K	96 K	nbexec.exe
OUTLOOK.EXE	6064	subramar	00	132,728 K	62,800 K	Microsoft Office Outlook
POWERPNT.EXE	500	subramar	00	11,332 K	5,168 K	Microsoft Office PowerPoint
Realmon.exe	4196	subramar	00	1,844 K	556 K	Realmon.exe
rundll32.exe	2356		00	2,612 K	884 K	
rundll32.exe	4124	subramar	00	292 K	152 K	Windows host process (Rundll32)
sftdcc.exe	3892	subramar	00	252 K	128 K	Microsoft Application Virtualization Desktop Configu
SwDnld.exe	4068	subramar	00	532 K	160 K	Shockwave Download Module
taskeng.exe	2388	subramar	00	7,984 K	4,296 K	Task Scheduler Engine
taskmgr.exe	7052	subramar	01	9,372 K	2,828 K	Windows Task Manager
tr_w.exe	4080	subramar	00	116 K	116 K	tr_w.exe
winlogon.exe	1344		00	1,028 K	192 K	
WINWORD.EXE	7392	subramar	00	37,592 K	14,064 K	Microsoft Office Word
XmlView.orig.exe	7764	subramar	01	44,608 K	16,812 K	XmlView

Processes: 75 CPU Usage: 2% Physical Memory: 63%

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

Figure 1 **XMLView Memory Usage in Task Manager Before Search Operation**

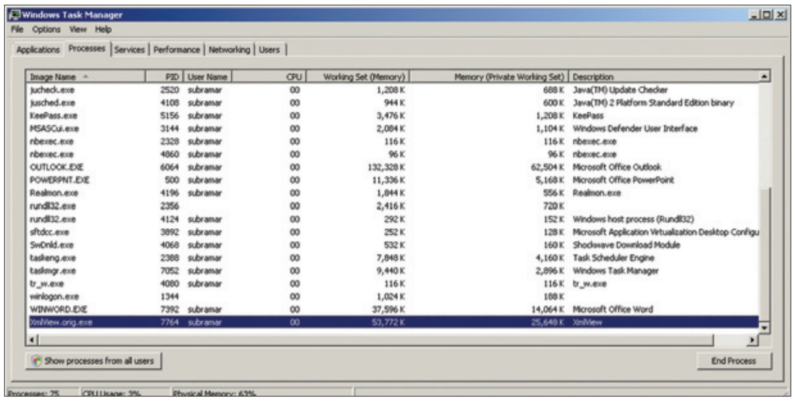


Image Name	PID	User Name	CPU	Working Set (Memory)	Memory (Private Working Set)	Description
jucheck.exe	2520	subramar	00	1,208 K	688 K	Java(TM) Update Checker
jusched.exe	4108	subramar	00	944 K	600 K	Java(TM) 2 Platform Standard Edition binary
KeePass.exe	5156	subramar	00	3,476 K	1,208 K	KeePass
MSASCui.exe	3144	subramar	00	2,084 K	1,104 K	Windows Defender User Interface
nbexec.exe	2328	subramar	00	116 K	116 K	nbexec.exe
nbexec.exe	4860	subramar	00	96 K	96 K	nbexec.exe
OUTLOOK.EXE	6064	subramar	00	132,328 K	62,504 K	Microsoft Office Outlook
POWERPNT.EXE	500	subramar	00	11,336 K	5,168 K	Microsoft Office PowerPoint
Realmon.exe	4196	subramar	00	1,844 K	556 K	Realmon.exe
rundll32.exe	2356		00	2,416 K	720 K	
rundll32.exe	4124	subramar	00	292 K	152 K	Windows host process (Rundll32)
sftdcc.exe	3892	subramar	00	252 K	128 K	Microsoft Application Virtualization Desktop Configu
SwDnld.exe	4068	subramar	00	532 K	160 K	Shockwave Download Module
taskeng.exe	2388	subramar	00	7,848 K	4,160 K	Task Scheduler Engine
taskmgr.exe	7052	subramar	00	9,440 K	2,896 K	Windows Task Manager
tr_w.exe	4080	subramar	00	116 K	116 K	tr_w.exe
winlogon.exe	1344		00	1,024 K	188 K	
WINWORD.EXE	7392	subramar	00	37,596 K	14,064 K	Microsoft Office Word
XmlView.orig.exe	7764	subramar	00	53,772 K	25,648 K	XmlView

Processes: 75 CPU Usage: 3% Physical Memory: 63%

Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Show processes from all users
End Process

Figure 2 **XMLView Memory Usage in Task Manager After Search Operation**

Microsoft. A Web search on "CLR Profiler" will locate the download site, and the installation consists simply of unzipping the files (there is no formal installation step). Included as part of the package, the file CLR Profiler.doc provides the documentation. In this article, we will focus on the parts that are used commonly.

How Does CLR Profiler Work?

In order to use a profiler properly, you need to understand how the data was collected and the limitations of the tool. The CLR Profiler uses a special interface to the runtime that allows it to get callbacks when particular events happen within the runtime (msdn.microsoft.com/library/ms404511.aspx). Currently, this interface can be used only if the process being profiled is started with certain environment variables set that inform the runtime regarding the contact mechanism to the profiler. Thus, there are some limitations, such as the inability to attach to an existing process. CLR Profiler must start the process with the right environment variables set. For services that are typically started by the OS directly, like ASP.NET, special procedures have to be followed (see CLR Profiler.doc from the download for details). As an aside, for certain scenarios we have added API support for attach in the .NET Framework 4 (but only the API support; support for attaching the CLR Profiler tool will follow soon after).

When CLR Profiler starts up the application, it provides options on the type of information it collects. By default, it gets callbacks every time a managed method is called or returns from a call, as well as on every GC heap allocation. These callbacks are called often, resulting in the program slowing down substantially and creating large log files more than 100 MB) for many real-world scenarios. CLR Profiler does not measure execution time and limits itself to keeping track of only GC heap allocations and the number of times each different method was called. CLR Profiler uses the *method-call* and *method-return* events to keep track of what methods are currently active on the stack, so that each call and heap allocation can be attributed with a complete call stack of where the event happened. This identifies where the objects are being created.

Unfortunately, the detailed information that CLR Profiler collects by default can make the data collection process very slow (minutes). However, this detailed information is rarely needed. One of the most useful techniques does not require any additional information to be collected during normal execution. While it is important to know where memory allocation occurs, many bugs can be fixed only by understanding who is keeping memory alive. To answer this latter question, one does not need call stacks or allocation events; rather, only a simple snapshot of the GC heap is required. Thus, for most investigations, one can turn off all normal event logging and obtain a good experience (reasonable profiling performance as well as identifying the issue). This is the technique used here.

Using CLR Profiler

If you download CLR Profiler from the Web and extract it to the default location, the program will end up in C:\CLR Profiler\Binaries\x86\ClrProfiler.exe (there is an X64 version, too). Launching CLR Profiler brings up the form shown in **Figure 3.**

This start-up dialog controls the collection of profile data. The check boxes indicate that when the application is executed, profiling will be active and both allocation and call data will be collected (the log file will be verbose). Since we are interested in only heap snapshots, only the Profiling active box has to remain selected (the Allocations and Calls boxes should be unchecked). Thus, the CLR Profiler will log only the bare minimum during program execution, keeping responsiveness up and log-file sizes down.

If the application you wish to profile does not need any command-line arguments or special start-up directory, you can simply select Start Application and CLR Profiler will bring up a file chooser dialog box to select the application to run. If you need to set command-line parameters or the current directory for the application, you should set them using the File -> Set Parameters menu before clicking on Start Application.

Taking a Heap Snapshot

After selecting the application executable, CLR Profiler will immediately start running the application. Because we have turned off

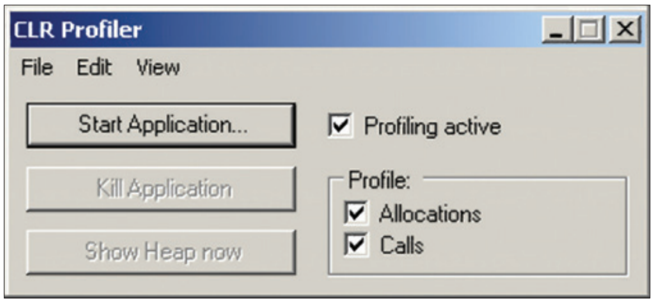


Figure 3 **CLR Profiler Start-up Screen**

allocation and call profiling, it should run very close to its normal speed. CLR Profiler is now attached to the application, and at any point you can select Show Heap Now and get a heap snapshot. The heap snapshot for the XMLView application happens to look like what is shown in **Figure 4**.

This is a representation of the entire GC heap (only live objects are shown). While the GC heap is actually an arbitrary graph, CLR Profiler determines links and forms a tree out of it, rooted in a pseudo-node called *root*. The size of each of the children can then be attributed to their parents so that at the root, all live objects have been accounted for. In the example above, the GC heap had 5.4MB of objects. This 5.4MB does not include free space in the GC heap that has not been allocated, and therefore, the amount of virtual memory allocated for the heap is guaranteed to be larger (the exact size depends, but as mentioned in the previous article, 1.6 times larger is a good estimate).

Often, there is more than a single path from the root to a particular object. CLR Profiler always chooses the shortest path for display in the tree. However, if there are multiple paths with the same length, the choice is made arbitrarily. Additionally, by default, CLR Profiler does not display individual objects but treats all objects that have the same parent types and the same child types as part of the same node. This technique generally does a good job of collapsing linked lists nodes and other recursive data structures, thereby simplifying the view. If this aggregation is not desired, you can turn it off by right-clicking and selecting Show Individual Instances.

At the top of the Heap Graph window are two radio button groups. The first is the Scale, which indicates the height of each box in the graph view. The second is the Detail button, which determines how large a node has to be to get displayed. It is useful to set this to something very coarse, to keep the graph uncluttered. Often, you are interested in only the graph that has something to do with a particular type (for example, only about objects of type XmlPositions). CLR Profiler allows you to filter on this and other criteria (Right click -> Filter).

In the XMLView example in **Figure 4**, the bulk of the GC heap (5.1MB) is a child of an instance of XmlView type. The XmlView instance has a child of type XmlPositions, which in turn has a 5MB XmlElementPositionInfo array. XmlElementPositionInfo is the large array that keeps track of the starting point in the file of every XML tag, and its size is proportional to the size of the XML file being read (because this is a very large file, it is expected that this array will be large). Thus, we concluded the heap size is reasonable.

Tracking Down GC Heap Growth

In the XMLView example, our problem was not with the initial size of the GC heap, but the fact that it grew dramatically when a search was done. Thus, what we are interested in is the change in the heap between two snapshots. CLR Profiler has special features just for doing this kind of investigation.

After taking one heap snapshot, we can then perform the search operation (causing the additional memory to be allocated), and then take another heap snapshot. The result is the following display:

The memory associated with the XmlView object has now expanded to 8.5MB (from 5.1MB). The nodes have two colors, with the lighter red representing the memory that was present in the previous snapshot. The darker red is the memory that is unique to the new snapshot. Thus, the new memory was added by a type called XmlFind (this is not surprising), as can be seen in **Figure 5**. Scrolling over to the right, we find that all of this extra memory was associated with an Int64 array that belonged to the type LineStream, as shown in **Figure 6**.

This was the information needed to identify the bug! It turns out that the .NET XML parser has the capability to get the line number column number of an XML tag, but not its file position. To work around this, the program needed a mapping from line number to file position. The Int64 array was used for this. However, the code needed this mapping only for the *current* XML element tag, which might span any number of lines, but usually just a few. This mapping was not being *reset* by the find code, allowing the mapping array to grow unnecessarily large. The fix was trivial, but this bug would probably have existed forever if the memory audit was not performed. Using the CLR Profiler, it took only a few minutes to find the information that was critical to fixing this bug.

Displaying the New Objects Alone

In the previous example, it was relatively easy to visually distinguish by their color the objects unique to the last snapshot. However, for large object graphs, this is difficult to do and it would be convenient to remove the old objects from the graph altogether, which can be done using CLR Profiler. This feature uses the allocation events for implementation, so after taking the first snapshot, you need to check the Allocations box (you can leave the Calls box unchecked). Following this, interact with the application until it

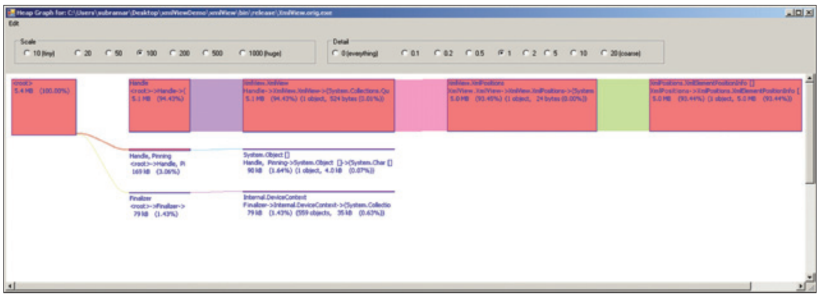
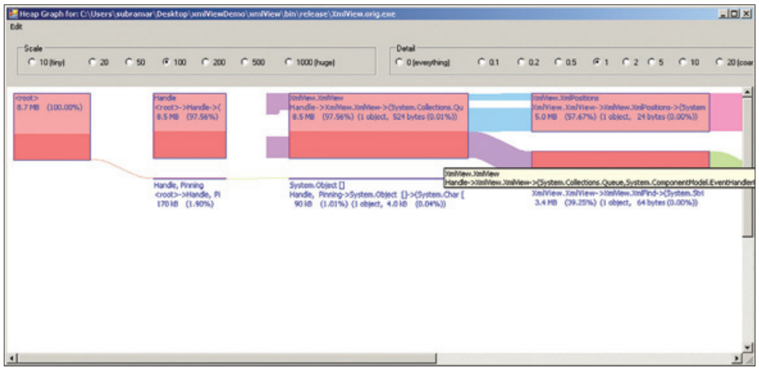

Heap Graph for: C:\Users\subramar\Desktop\xmlViewDemo\xmlView\bin\release\XmlView.orig.exe
Edit
Scale
10 (tiny)
20
50
100
200
500
1000 (huge)
Detail
0 (everything)
0.1
0.2
0.5
1
2
5
10
20 (coarse)
<root>
5.4 MB (100.00%)
Handle
<root>->Handle->(
5.1 MB (94.43%)
XmlView.XmlView
Handle->XmlView.XmlView->(System.Collections.Qu
5.1 MB (94.43%) (1 object, 524 bytes (0.01%))
XmlView.XmlPositions
XmlView.XmlView->XmlView.XmlPositions->(System
5.0 MB (93.45%) (1 object, 24 bytes (0.00%))
XmlPositions.XmlElementPositionInfo []
XmlPositions->XmlPositions.XmlElementPositionInfo [
5.0 MB (93.44%) (1 object, 5.0 MB (93.44%))
Handle, Pinning
<root>->Handle, Pi
169 kB (3.06%)
System.Object []
Handle, Pinning->System.Object []->(System.Char []
90 kB (1.64%) (1 object, 4.0 kB (0.07%))
Finalizer
<root>->Finalizer->
79 kB (1.43%)
Internal.DeviceContext
Finalizer->Internal.DeviceContext->(System.Collectio
79 kB (1.43%) (559 objects, 35 kB (0.63%))

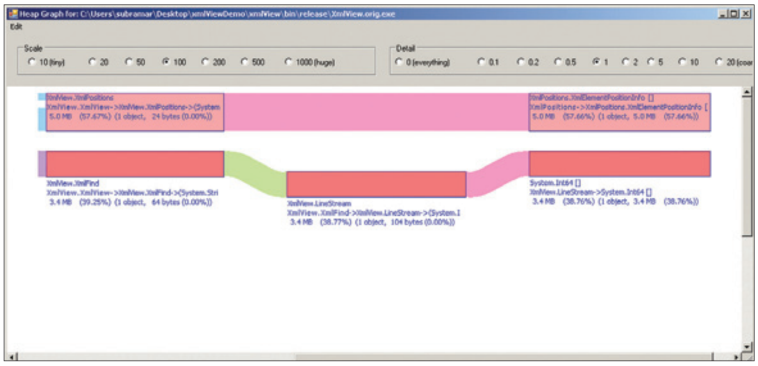
Figure 4 **CLR Profiler Snapshot Before Search Operation**


Heap Graph for: C:\Users\subramar\Desktop\xmlViewDemo\xmlView\bin\release\XmlView.orig.exe
Edit
Scale
10 (tiny)
20
50
100
200
500
1000 (huge)
Detail
0 (everything)
0.1
0.2
0.5
1
2
5
10
20 (coar

Scale: [✓ 100]; Detail: [✓ 1]

```
<root>
8.7 MB   (100.00%)

Handle
<root>->Handle->(
8.5 MB   (97.56%)

XmlView.XmlView
Handle->XmlView.XmlView->(System.Collections.Qu
8.5 MB   (97.56%) (1 object,  524 bytes (0.01%))

XmlView.XmlPositions
XmlView.XmlView->XmlView.XmlPositions->(System
5.0 MB   (57.67%) (1 object,  24 bytes (0.00%))

XmlView.XmlView
Handle->XmlView.XmlView->(System.Collections.Queue,System.ComponentModel.EventHandler

Handle, Pinning
<root>->Handle, Pi
170 kB   (1.90%)

System.Object []
Handle, Pinning->System.Object []->(System.Char [
90 kB   (1.01%) (1 object,  4.0 kB   (0.04%))

XmlView.XmlView->XmlView.XmlFind->(System.Stri
3.4 MB   (39.25%) (1 object,  64 bytes (0.00%))
```

Figure 5 **CLR Profiler Snapshot After Search Operation - Part 1**


Heap Graph for: C:\Users\subramar\Desktop\xmlViewDemo\xmlView\bin\release\XmlView.orig.exe
Edit
Scale
10 (tiny)
20
50
100
200
500
1000 (huge)
Detail
0 (everything)
0.1
0.2
0.5
1
2
5
10
20 (coar

Scale: [✓ 100]; Detail: [✓ 1]

```
XmlView.XmlPositions
XmlView.XmlView->XmlView.XmlPositions->(System
5.0 MB   (57.67%) (1 object,  24 bytes (0.00%))

XmlPositions.XmlElementPositionInfo []
XmlPositions->XmlPositions.XmlElementPositionInfo [
5.0 MB   (57.66%) (1 object,  5.0 MB   (57.66%))

XmlView.XmlFind
XmlView.XmlView->XmlView.XmlFind->(System.Stri
3.4 MB   (39.25%) (1 object,  64 bytes (0.00%))

XmlView.LineStream
XmlView.XmlFind->XmlView.LineStream->(System.I
3.4 MB   (38.77%) (1 object,  104 bytes (0.00%))

System.Int64 []
XmlView.LineStream->System.Int64 []
3.4 MB   (38.76%) (1 object,  3.4 MB   (38.76%))
```

Figure 6 **CLR Profiler Snapshot After Search Operation - Part 2**

allocates more memory, and select Show Heap Now again. It will show the heap with its two colors of red as before. Now, you can right-click on the window and select Show New Objects, and a new window will appear that shows you only the part of the GC heap that was allocated between the two snapshots.

Tracking Down All Roots for an Object

You might wonder how memory leaks can happen in managed code, given the garbage collector eventually cleans up all unused memory. However, there are situations where items can stay alive long after you might think they would be freed. For example, unintended references may keep managed objects alive.

The technique for tracking down memory leaks is very similar to that for understanding disproportionate memory consumption. The only real difference is that for disproportionate memory consumption, we do expect the object to be in memory (it just should not be as big), whereas for leaks, we do not expect the object to be in the GC heap at all. Using heap snapshots and the Show New Objects feature, it is easy to identify which objects should not be alive (using your understanding of the scenario). If an object is surviving in the heap, there must be some reference to the object. However, if there is more than a single reference, you may find removing one link does not fix the issue because another reference is keeping it alive. Thus, it would be convenient to find all paths from the root that keep a particular object alive in the GC heap. CLR Profiler has special support for doing this. First, the node of interest is selected. Then right-click and select Show References, and a new graph will appear that shows all paths from the root to the object in question. This will provide you with enough information to identify which references are keeping the objects alive in the heap. Eliminating the links to the object can fix your memory leak.

Getting Allocation Call Stacks

Viewing the GC heap is frequently sufficient to diagnose most high memory-consumption problems. However, on occasion, it is useful to understand the call stack where an allocation occurred. To obtain that information, you have to select the Allocations check box before executing the program. This causes CLR Profiler to log the stack trace whenever an allocation happens. Collecting this information will make the log file bigger, but it will still typically be less than 50MB for moderately complex applications. You should not turn on the Calls check box because this causes logging on every method entry, which is verbose (it is easy to get log files larger than 100MB). This method call information is needed only if you are using CLR Profiler to determine which methods are being called often, and this is not typically useful in a memory investigation.

From the heap graph, select a node of interest, right-click and select Show Who Allocated. A new window showing all calls stacks that allocated an object in that node will be displayed. You can also access the stack traces from the View -> Allocation Graph menu item. This shows all allocations made during the program attributed to the call stack that allocated them. For memory investigations, this view is not very useful because many of these allocations are for short-lived objects that quickly get reclaimed by the GC and, therefore, do not contribute to overall memory consumption. However, since these allocations do contribute to CPU usage (to allocate and then reclaim), this view is useful when your application is CPU-bound and the CPU profiler shows large amounts of time being spent in the GC. The obvious way to trim down time in the GC is to do fewer allocations, and this view shows you where your allocations are by call stack.

Part of the Development Lifecycle

In this article, we walked through a memory performance investigation for the common case where the memory consumption is dominated by the .NET GC heap. Using the free and downloadable CLR Profiler, we were easily able to get snapshots of the GC heap at various points in the application and compare the snapshots with one another. With CLR Profiler, it is easy to perform audits (it takes only a few minutes) and should be part of the development lifecycle of every application. ■

SUBRAMANIAN RAMASWAMY *is the program manager for CLR Performance at Microsoft. He holds a Ph.D. in electrical and computer engineering from the Georgia Institute of Technology.* **VANCE MORRISON** *is the partner architect and group manager for CLR Performance at Microsoft. He drove the design of the .NET Intermediate Language and has been involved with .NET since its inception.*

Collection and Array Initializers in Visual Basic 2010

Some of the language changes that have taken place during the Microsoft Visual Studio (VS) 2010 product cycle have been aimed at simplifying tasks that were previously possible only through the use of a lot of boilerplate code. This approach in Visual Studio 2010 improves the coding experience, making you more productive. One of the features new in Visual Studio for the 2010 release is Collection Initializers. This feature was available in C# in 2008 and as part of the general parity philosophy Microsoft has been adopting. Some of the changes made to Visual Basic 2010 may be less immediately obvious, and this article will identify some of the less obvious implementation details.

Collections and arrays are very common constructs in any modern application, and these collections often have to be initialized. Prior to the 2010 releases, Visual Basic was able to handle basic array initializers with some limitations, but collections were not able to be initialized in a similar one-line statement, which often resulted in boilerplate code like the following:

```
Dim x As New List(Of String)
x.Add("Item1")
x.Add("Item2")
```

Or it required creating a constructor for the class, using something like the following:

```
Dim x as new list(of string)({"Item1","Item2"})
```

Neither of these approaches were particularly elegant solutions.

While both worked, they involved writing additional code to initialize the collection. Although this code is often straightforward boilerplate code, it can bloat the source code, which can lead to associated maintenance costs.

If you look through almost any code that uses collections, you will see similar boilerplate code. In the preceding example, we are adding only two string elements and are using repetitive calls to the add method for each of the strings. If we used a custom "user-defined" collection or a collection of "user-defined" types, the standard code would increase to do the same task of initializing the collection.

New Syntax

With Visual Studio 2010, we are now able to handle this collection initialization task in a much more concise form, reducing down the multiple lines of code and repeated Add method calls into a single statement:

```
Dim x As New List(Of String) From {"Item1", "Item2"}
```

This is a simple example using the framework's generic list collection types to provide a type-safe collection. These collection types are now more commonly used than handcrafted collection types, which were used in the past for implementing type-safe collections.

The syntax allows for the collection members to be specified as a list of items contained within the braces { }, and each item is separated by commas using the "FROM" keyword to determine the initial member list.The preceding example works fine for a list type, but for more complex collection types such as dictionaries, where there may be a key-value pair provided for each member, we require an additional set of braces around each key-value pair, as follows:

```
Dim x As New Dictionary(Of Integer, String)
  From {{1, "Item1"}, {2, "Item2"}}
```

The syntax is clean and consistent. Each key-value pair is determined by the nested {}, and these arguments are used when a call to the collection type (Dictionary) Add method is made. So the preceding lines would equate to:

```
Dim x As New Dictionary(Of Integer, String)
x.Add(1, "Item1")
x.Add(2, "Item2")
```

Implementation and Usage in Other Collection Types

The preceding examples show how collections initialize usage for typical framework types. However, you may have implemented your own collection types or may want to use some types where you are not immediately able to use the initializer because they lack an Add method (examples include Stack, Queue).

To allow us to use the Collection Initializer feature in both these scenarios, it is important to understand a little about what is going on under the covers and how this feature is implemented. This understanding will then allow us to expand its use beyond the simple list and dictionary examples.

To use Collection Initializer syntax, we require two items to be true. The type must:

1. Implement the IEnumerable pattern, which can be either the IEnumerable interface or simply have a GetEnumerator method. This may be referred to as "duck" typing: if it looks like a duck and quacks like a duck, then it probably is a duck.
   In our case, if it contains a GetEnumerator method with an ap-

Column Disclaimer: This article is partly based on a prerelease version of Visual Studio. All information is subject to change.

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

propriate signature, then it is probably implementing similar behavior for the IEnumerable interface. This duck-typing behavior is already used to allow types to be used with the For Each construct.

2. Contain at least one accessible Add method with one parameter. This can be either an instance or an extension method.

If your collection class meets both these conditions, you can use the Collection Initializer syntax.

For Collection Initializers, we do not actually use the IEnumerable method for initializing the collection, but we do use it as a hint to determine that this type is, in fact, a collection type. If it implements IEnumerable or the IEnumerable pattern, then there is a high probability that you have a collection, although it's not guaranteed.

The Add method is called for each item in the Initializer List to add items to the collection. The fact that an Add method is present does not necessarily mean it has to implement functionality to Add items to the collection. But calls to an Add method with the arguments transposed from the "Initialize list" will be made.

To illustrate this point, the following code sample in **Figure 1** is valid and can use the Collection Initializer syntax, but in fact doesn't add any items to a collection.

Figure 1 **Collection Initializer Syntax**

```
Module Module1
    Sub Main()
        Dim NC As New TestNonCollectionClass From {1, 2, 3}
    End Sub
End Module

Class TestNonCollectionClass
    Public Function GetEnumerator() As System.Collections.IEnumerator
        'Does Nothing
    End Function

    Public Sub Add(ByVal x As Integer)
        If x = 1 Then
            Console.WriteLine("Add Item" & x.ToString)
        End If
    End Sub
End Class
```

Other Framework Classes—Use with Extension Methods

Some of the framework collection types don't meet both these requirements. Stacks and queues implement methods such as Pop and Push rather than Add. To allow you to use the concise Collection Initializer feature for these types, you can leverage the power of extension methods. It is simple to create an Add Extension method for these types, which then will permit you to initialize the methods with simple syntax.

```
Imports System.Runtime.CompilerServices

Module Module1
    Sub Main()
        Dim st1 As New Stack(Of Integer) From {1, 2, 3}
    End Sub
End Module

Module Extensions
    <Extension()> Sub Add(Of t)(ByVal x As Stack(Of t), ByVal y As t)
        x.Push(y)
    End Sub
End Module
```

When using extension methods, it is generally good practice to extend only the types that you have control over. Liberal use of extension methods can lead to conflicts or changes in behavior if the types are upgraded, so use them with caution.

Use with Implied Line Continuation

Visual Studio 2010 also contains another much-awaited feature: line continuation. This feature allows you to eliminate the pesky _ characters. Implicit line continuation is permitted for items in the "Initialize List", so it is possible to put each of these items on their own lines for clarity.

```
Dim st1 As New Stack(Of Integer) From {1,
                                       2,
                                       3}
```

This allows you to produce clean code that is simple to read, doesn't contain extra _ characters and avoids repetition. You can use it for your own collections and those already in the framework. The editor's IntelliSense support allows "FROM" support for all types that follow the preceding two rules, which means it will work for your collection types. If you have existing code that utilizes the _ character and prefer to remain consistent with old syntax, this choice is still supported.

Exception-Handling Behavior

There are a few interesting things to point out when calling the Add method for each of the "Initializer List" members.

For example, when a call to the Add method with the provided argument from the Initialize List results in an exception, the list will not be initialized with any members. What this means in reality is that the collection is initialized with all members or its not initialized at all.

This means that if your list has three items and the third one when called using the Add method results in an exception, you get an exception thrown and an uninitialized collection. The following-

Figure 2 **Changing the Initializer List**

```
Imports System.Runtime.CompilerServices

Module Module1
    Sub Main()
        Dim l1 As Stack(Of Integer)
        Try
            l1 = New Stack(Of Integer) From {1, 2}
        Catch ex As Exception
        End Try
        If l1 Is Nothing Then
            Console.WriteLine("Blank")
        Else
            Console.WriteLine("Initialized - Element Count:" & l1.Count)
        End If
    End Sub
End Module

Module Extensions
    <Extension()> Sub Add(Of t)(ByVal x As Stack(Of t), ByVal y As t)
        If y.ToString = "3" Then
            Throw New Exception("Intentional Exception")
        Else
            x.Push(y)
        End If
    End Sub
End Module
```

Figure 3 **Use of Extension Methods**

```
Imports System.Runtime.CompilerServices

Module Module1
    Sub Main()
        'Shortend Syntax through the use of extension methods
        Dim CustomerList As New List(Of Customer)
          From {{"Spotty", 39}, {"Karen", 37}}

    End Sub
End Module

Module Extension
    <Extension()> Public Sub add(ByVal x As List(Of Customer), _
      ByVal Name As String, ByVal Age As Integer)
        x.add(New Customer With {.Name = Name, .Age = Age})
    End Sub
End Module

Class Customer
    Public Property Name As String = ""
    Public Property Age As Integer = 0
End Class
```

example intentionally generates an exception to demonstrate this exact scenario and results in "Blank" being written to the console. However, changing the Initializer List to remove the value three will result in "Initialized:Element Count 2," as shown in **Figure 2**.

If this didn't happen, then it would require crafting additional code to determine whether no initialization exceptions occurred and what state the collection was in.

Using Extension Methods to Shorten Syntax

The standard syntax for initializing a collection of complex objects results in having to repeat "New <type>" for each of the items in the Initializer List.

```
Module Module1
    Sub Main()
        Dim CustomerList As New List(Of Customer)
        From {New Customer With {.Name = "Spotty", .Age = 39},
              New Customer With {.Name = "Karen", .Age = 37}}
    End Sub
End Module

Class Customer
    Public Property Name As String = ""
    Public Property Age As Integer = 0
End Class
```

The syntax requires that each item in the Initializer List is instantiated. Through the use of extension methods, it is possible to shorten this syntax still further. The functionality relies on the fact that the collection type supports the IEnumerable interface and also has an Add method, which will result in the enclosed Initializer List items being mapped to the add method parameters, as shown in **Figure 3**.

Add Method Overloading

The Add method condition is critical to this feature and can be overloaded. Each method will call its own overload based upon the calling arguments. There is no magic here and the Add method overload resolution works just as overload resolution works for any method.

This again is best demonstrated by a simple example. In **Figure 4**, we have overloads for different data types—we will call the one appropriate for the calling argument.

Why Does the Syntax Use 'FROM' and not '='?

A common question asked of the product team for this feature is the use of the "FROM" keyword, because the array initializers already are using the "=" for assignment.

In Visual Basic, it is possible to instantiate a collection class in three different ways:

```
Dim x1 As New List(Of Integer)
Dim x2 = New List(Of Integer)
Dim x3 As List(Of Integer) = New List(Of Integer)
```

Add to this the fact that two of the syntaxes for instantiating a collection already include an = character in the syntax. Using an additional = character would result in confusion in syntax between that of assignment of a new instance of the collection type and initialization of members to the collection.

So using a keyword rather than a = character to determine the result of a Collection Initializer action avoids this issue and allows all existing syntax conventions to declare and initialize collection types. Yet the way to determine the Initializer List remains familiar to the existing array initializer syntax.

The FROM keyword was chosen after much discussion. To those familiar with LINQ queries, it might seem a strange choice as there is already a FROM keyword used in LINQ. It may appear that this choice could lead to potential ambiguities when used within LINQ queries but as we will see, this is not the case.

Doesn't 'FROM' Clash with 'FROM' in Queries?

Even in queries that contain Collection Initializers, there is no ambiguity of the keyword. This can be demonstrated by the following code:

```
Module Module1
    Sub Main()
        'The first from is the query,
        'the second is the collection initializer.
        'The parser can always successfully identify the difference
        Dim x = From i In New List(Of Integer) From {1, 2, 3, 4}
                Where i <= 3
                Select i
        For Each i In x
            Console.WriteLine(i)
        Next
        Stop
    End Sub
End Module
```

Limitations Using Collection Initalizer with Object Initializer

For the 2008 product, the Visual Basic team implemented object initializers that enabled the developer to initialize object fields/properties when they were instantiating an object instance:

```
Dim x As New List(Of Integer) With {.Capacity = 10}
```

However, it is not possible to initialize both the object and the collection on the same declaration, so the following will produce a syntax error:

```
Dim x as New List(Of Integer) from {1,2,3} With {.Capacity = 10}
```

There are some other less obvious changes that occurred in 2010 relating to arrays that may not be immediately noticeable but could have impact on code using arrays.

Figure 4 **Overloads for Different Data Types**

```
Imports System.Runtime.CompilerServices

Module Module1
    Sub Main()
        Dim l1 As New Stack(Of Integer) From {1, 2.2, "3"}
        Console.WriteLine("Element Count:" & l1.Count)
        Stop
    End Sub
End Module

Module Extensions
    <Extension()> Sub Add(ByVal X1 As Stack(Of Integer), _
      ByVal x2 As Integer)
        Console.WriteLine("Integer Add")
        X1.Push(x2)
    End Sub
    <Extension()> Sub Add(ByVal X1 As Stack(Of Integer), _
      ByVal x2 As Double)
        Console.WriteLine("Double Add")
        X1.Push(CInt(x2))
    End Sub
    <Extension()> Sub Add(ByVal X1 As Stack(Of Integer), _
      ByVal x2 As String)
        Console.WriteLine("String Add")
        X1.Push(CInt(x2))
    End Sub
End Module
```

Array Initializer Changes

Although prior to the 2010 version it was possible to initialize simple arrays, there were still some limitations. One of the recent changes allows array types to be initialized and inferred in more concise ways.

Previously, to identify an item in Visual Basic as an array required you to specify a set of parentheses on the identifier:

```
Dim a() As Integer = {1, 2, 3}
```

Now these parentheses are not required and the type will be inferred as an array type with Option Infer On, resulting in a more concise syntax:

```
Dim a = {1, 2, 3}
```

Array Type Inference

In 2008, the type-inference feature was implemented, which enabled data types to be inferred from their assignment values. This worked for single objects declared within method bodies, but arrays did not infer a type and therefore were always arrays of Objects.

```
    Sub Main()
        Dim a() = {1, 2, 3}
          'Type would previously result in Object Array prior to 2010
    End Sub
```

This has been improved, and now array types are inferred, as follows:

```
    Sub Main()
        Dim a = {1, 2, 3} 'Type now infers Integer Array
    End Sub
```

The inferred type of the array is determined by the dominant type functionality added in 2008. This type inference still works only for code declared within method bodies, so fields will still be object arrays unless specifically typed.

It is possible to initialize multiple array dimensions, but all the nested pairs must contain the same number of members. So two-dimension examples will work.

The following two examples will both create a two-dimensional integer array:

```
Dim TwoDimension1(,) = {{1, 2}, {3, 4}}
Dim TwoDimension2 = {{1, 2}, {3, 4}}
```

For single-dimension arrays, the syntax is very similar to previous code, and all existing code will continue to work with a few exceptions.

```
Dim a() = {1, 2, 3}
```

In the preceding statement, the array type will be inferred from the initializer list. This will result in an integer array being inferred. In the past, these would have simply been Object arrays as we would not have inferred any array type from the Initializer list. If you are using any code that checks the type, uses reflection or contains multiple overloads including object array types, this may now produce different results. Examples of this may include late-bound methods.

Multidimensional Arrays

For you to initialize a multidimensional array, the dimensions must match the item count within the Initializer List. If these are not the same and all nested items in the initialize list do not have a consistent item count, then a syntax error will occur:

```
Dim TwoDimension2 = {{1, 2}, {3, 4}}
  'Valid 2 dimension integer(,) array inferred
Dim TwoDimension2Invalid(,) = {{1}, {3}}
  'Invalid - dimension and element count mismatch
Dim TwoDimension2Invalid1(,) = {{1, 2}, {3}}
  'Invalid:element count not consistent in Initializer List
```

This means that using the similar syntax as before, it is possible to initialize standard fixed dimensional arrays.

Jagged Array Initialization

However, this means that for a jagged array (an array of arrays), this syntax won't work as is. To allow for jagged array initializers, it is required that you wrap each of the member lists in parentheses:

```
    Sub Main()
        'Incorrect Jagged Array syntax
        Dim JaggedDimensionIncorrect()() = {{1,2},{3,4,5}}

        'Correct Jagged Array syntax
        Dim JaggedDimension1()() = {({1,2}),({3,4,5})}
    End Sub
```

The preceding example results in the following:

- JaggedDimension1(0) containing Integer array with elements 1,2
- JaggedDimension1(1) containing Integer array with elements 3,4,5

The array type-inference will work for the nested array types. Here is a slightly more complex example:

```
Dim JaggedDimension1() = {({1, 2}), ({3.1, 4.2, 5.3})}
```

This will result in JaggedDimension1 being inferred as Object() and the members being of type Integer() and Double().

Array Inference and Expected Behavior

So you look on the Initializer List as an inferred array. Don't ! The Initializer List is not a concrete type as you might think; it is a syntactical representation of a list of members, which becomes a specific type depending upon the context it is used in.

The following code sample demonstrates that we can use the {, , ,} syntax to:

1. Infer and initialize an integer array:

```
Dim a = {1, 2, 3}  'Infers Integer Array
```

This works as no target type is specified and it infers its type based upon the dominant type of the Initializer List members.

2. Initialize an array of different type:

```
Dim b As Single() = {1, 2, 3}
  'Will convert each integer value into single
```

This will work as we will initialize a single array with values one, two and three. The Initializer List has no intrinsic type by itself.

But the following is expected to fail:

```
Dim a = {1, 2, 3}  'Infers Integer()
Dim c As Single() =
  a 'No conversion between Single() and Integer()
```

This may seem a little strange but variable a is inferred as an integer array and initialized with values one, two and three. Variable c is declared as a Single array. However, there is no conversion between a Single array and an Integer array, which will cause the syntax error.

Standalone Array Initializer Usage

The array Initializer List also can be used in a standalone context, in which case, it will have not have a specified target type and will work as an array of the dominant type of the members of the initialize list.

This allows some interesting scenarios not previously available. An example could be a method that takes in an array of items that you want to use to initialize a data structure. Where the method is called determines different default values, which are used as call arguments, as shown in **Figure 5**.

In this case, we create an array local for each call, simply to pass an array type to the method. With the new standalone usage feature, it is possible to avoid having to create unnecessary locals for this type of scenario and you can simply use the standalone Initializer List to pass array types.

The Initializer List in the standalone usage scenario snaps to an inferred array type. So in a standalone usage scenario, the initialize list can be thought of as an array literal. This saves having to declare items that are only going to be used in limited instances, such as individual method calls.

Figure 5 **Different Default Values Used as Call Arguments**

```
Module Module1

    Sub InitializeMethod(ByVal x As String())
        '....
    End Sub

    Sub Main()
        Dim i As Integer = 1

        Select Case I
            Case 1
                Dim Array1 As String() = {"Item1", "Item2"}
                InitializeMethod(Array1)
            Case 2
                Dim Array1 As String() = {"Item11", "Item12"}
                InitializeMethod(Array1)
        End Select
    End Sub
End Module
```

Use with Other 2010 Features

One of the features of the 2010 version of the product is multi-targeting. Although this feature existed in the 2008 version, some improvements have been made that allow many of the language features in the 2010 version of the product to be used on down-targeted versions (2.0, 3.0 and 3.5 targets). Therefore, you can use this improved functionality for collection and array initializers for your existing applications if you are using the 2010 version of the product. This enables you to simplify your existing source code and take advantage of the new language functionality.

So when you use this new functionality, is there potential for breaking existing code? The answer to this is yes, there is some potential for backward-compatibility breaks, but this is limited to a few scenarios where the benefits outweigh the original breaking change and the existing functionality can be retained easily:

```
Dim x1() = {1, 2, 3}
```

In previous versions, the preceding line of code would have resulted in an Object array, but with Visual Studio 2010, this will be type-inferred to the dominant type of the elements—in this case Integer(). This will be the case regardless of whether you are targeting 4.0 or down-targeting to an earlier version. If you have code that is expecting a specific type of Object array, then this code may fail. It can easily be corrected by explicitly specifying the target type:

```
Dim x1() As Object = {1, 2, 3}
```

The exception to this behavior is if there is no dominant type for the elements:

```
Dim x1() = {1, 2.2, "test"}
```

This will retain the earlier behavior, resulting in x1 being an object array.

Easy Application

Collection Initializers are a great addition to the language. They allow for a much more concise syntax in order to initialize both framework and user-defined collection types. With minimal changes, this syntax can be applied to existing code easily.

The functionality is ideally suited to be able to use other language features such as extension methods, enabling the reduction of the use of syntax for any collection types. This reduction in boilerplate code previously used makes the code smaller and easier to maintain.

The array initializer behavior has changed slightly, resulting in implementing array type inference and improved functionality, allowing standalone usage of arrays and improved ability to initialize multi-dimensional and jagged arrays over previous versions.

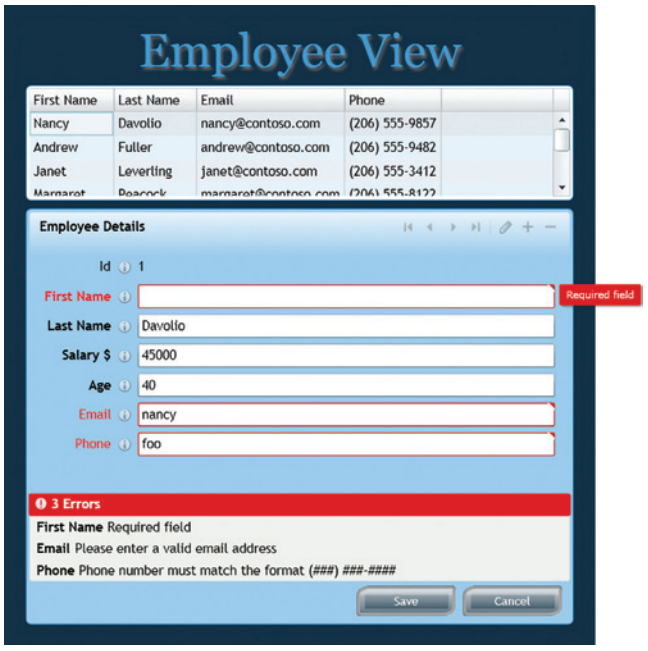
Virtually every application uses arrays or collections and this feature can be used in almost any project. ■

ADRIAN SPOTTY BOWLES *has developed using every version of Visual Basic and has managed to find his way to Redmond, Wash., where he works on the Visual Basic product team as a software design engineer tester focused on the Visual Basic compiler. During the Visual Basic 2008 release, he worked on many of the language features, including Extension Methods. You can reach Bowles at Abowles@microsoft.com.*

Data Validation with Silverlight 3 and the DataForm

Building robust Silverlight applications that edit and validate data got a whole lot easier with the introduction of the DataForm control in Silverlight 3. The DataForm control contains several features to help create forms that allow adding, editing, deleting and navigating through sets of data. The greatest benefits of the DataForm are its flexibility and customization options.

In this month's column, I will show how the DataForm control works and how it can be customized, and I'll demonstrate it in action. I'll begin by presenting a sample application that uses several features to bind, navigate, edit and validate data using the DataForm. Then I'll walk through how the sample application works, while pointing out the basic requirements to use the control and customizations I recommend. Though the DataAnnotations are not required by the DataForm , they can influence the appearance, validation aspects and behavior of the DataForm. The DataAnnotations can also indicate a custom validation method to invoke for a property or an entire entity. I will show how to implement the DataAnnotations on an entity and how to write your own custom validation methods. All code for this article can be downloaded from the MSDN Code Gallery (ambc,msdn.microsoft.com/mag200910DataPoints).



First Name	Last Name	Email	Phone
Nancy	Davolio	nancy@contoso.com	(206) 555-9857
Andrew	Fuller	andrew@contoso.com	(206) 555-9482
Janet	Leverling	janet@contoso.com	(206) 555-3412
Margaret	Peacock	margaret@contoso.com	(206) 555-8122

Employee Details

Id	1
First Name	
Last Name	Davolio
Salary $	45000
Age	40
Email	nancy
Phone	foo

Required field

3 Errors
First Name Required field
Email Please enter a valid email address
Phone Phone number must match the format (###) ###-####

Figure 1 **Validation and Editing with the DataForm**

What's the Endgame?

Before diving into the code, let's take a look at the sample application that demonstrates the navigation, validation and editing features of the DataForm. **Figure 1** shows a DataGrid displaying a list of employees and a DataForm where each employee record can be edited. The employees in the DataGrid can be navigated, changing which is the currently selected employee. The DataForm is bound to the same set of employees and receives that employee.

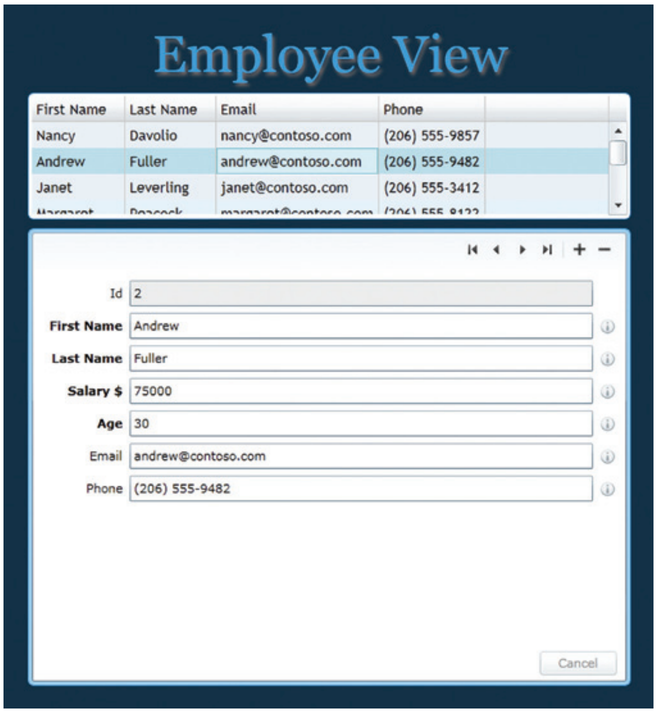
Toolbar The DataForm shown in **Figure 1** has been placed in edit mode so the user can make changes and either save or cancel them. The toolbar at the upper right of the DataForm shows icons for editing (pencil), adding a new record (plus sign) and deleting a record (the minus sign). These buttons can be styled and disabled as needed. The state of the buttons is determined by the interfaces implemented by the entity, while their visibility can be customized by setting properties on the DataForm.

Labels The captions for each TextBox in the DataForm can be placed beside or above the control. The content of the labels can be metadata-driven using the Display DataAnnotations attribute. The DataAnnotations attributes are placed on the properties of the entity, in this case the Employee entity, where they help feed the DataForm information, such as the text to display in the field labels.

DescriptionViewer When the DataForm is in edit mode (as shown in **Figure 1**), the user can hover over the icon next to the caption beside each TextBox control to see a tooltip that provides more information on what to enter in the field. This feature is implemented by the DataForm when you place a DescriptionViewer control beside the bound TextBox control. The text displayed in the tooltip of the DescriptionViewer is also fed from the DataAnnotations Display attribute.

Cancel and Commit The Cancel button at the bottom of the DataForm is enabled when in edit mode. The Save button is enabled when in edit mode and a user changes a value in the DataForm. The text and styles for these buttons can be customized.

Column Disclaimer: This article is based on prerelease versions of Silverlight 3.
Code download available at msdn.microsoft.com/mag200910DataPoints.
Send your questions and comments for John to mmdata@microsoft.com.



First Name	Last Name	Email	Phone
Nancy	Davolio	nancy@contoso.com	(206) 555-9857
Andrew	Fuller	andrew@contoso.com	(206) 555-9482
Janet	Leverling	janet@contoso.com	(206) 555-3412

Id	2
First Name	Andrew
Last Name	Fuller
Salary $	75000
Age	30
Email	andrew@contoso.com
Phone	(206) 555-9482

Figure 2 **DataForm out of the Box**

Validation Notification The DataForm in the sample is shown in the edit state where the user has typed in an invalid e-mail address and phone number. Notice that the field captions for the invalidated fields are in red and the textboxes are bordered in red with a red flag in the upper-right corner. When a user puts the cursor in an invalidated control, a tooltip appears with a message describing what has to be done to correct the problem. **Figure 1** also shows a list of the validation errors at the bottom of the screen. All of these features are supported by the DataForm and DataAnnotations and can use custom styles to give a custom look and feel. The validation rules are fed from several of the DataAnnotations to indicate canned rules, such as required fields, or custom rules.

Designing the DataForm

The DataForm, found in the Silverlight Toolkit, can be added to a project once the assembly System.Windows.Controls.Data.DataForm.Toolkit.dll has been referenced. The DataForm can be created with very little setup. Once dragged onto the design surface in Blend or created in XAML—by simply setting the ItemsSource of the DataForm to a collection of entities—the DataForm will automatically generate the fields to display and then enable appropriate editing features. The following XAML will generate the DataForm shown in **Figure 2**:

```
<dataFormToolkit:DataForm x:Name="dataForm" ItemsSource="{Binding
Mode=OneWay}" />
```

Note: The binding mode is set to OneWay in the previous example only for explicit purposes. It is not required. However, I find it useful for clarity and for supportability to explicitly set the binding mode.

Additional settings can be hooked up to the DataForm to refine its functionality. Common settings include the ones shown in **Figure 3.**

Often, it is beneficial to style the DataForm's visual features to coordinate them with the visual appearance of your application. Several aspects of the DataForm can be styled using resources in the document, app.xaml or a resource dictionary. There are several styles that can be set, as shown in **Figure 4** in the partial screenshot from Expression Blend. Notice that CancelButtonStyle and the CommitButtonStyle are both set to a value. This is what gives the

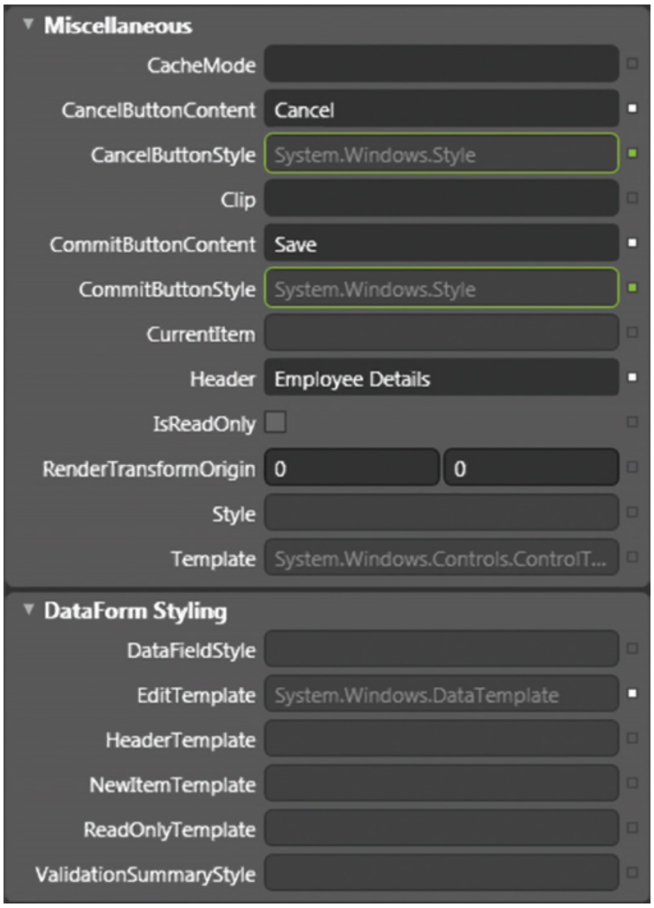
Figure 3 **Common DataForm Customizations**

Properties	Description
ItemsSource	The source of the data for the DataForm.
CommitButtonContent	The content to place in the Commit button. Can be text or other, more complex XAML content.
CancelButtonContent	The content to place in the Cancel button. Can be text or other, more complex XAML content.
Header	The title above the DataForm.
AutoEdit	Boolean value. True indicates that the DataForm should be placed in edit mode upon navigating to a record. False indicates that the DataForm is placed in read-only mode upon navigating to a record and the user must click the edit button on the toolbar to put the DataForm in edit mode.
AutoCommit	Boolean value. True indicates that navigating away from the current record will automatically commit the record on the DataForm. False indicates that the user must explicitly click the commit button.
AutoGenerateFields	Boolean value. True indicates that the DataForm should examine the DataAnnotations on the bound entity and display all fields not explicitly excluded. False indicates that the DataForm should only display fields added to the DataForm in the ReadOnlyTemplate, EditTemplate and NewItemTemplates. If no DataAnnotations are found, all public properties will be generated in the DataForm.
CommandButtonsVisibility	Indicates the buttons that should be visible on the DataForm. Multiple buttons may be specified if separated by commas, or all buttons can be specified by using the keyword All. Possible values are None, Edit, Add, Delete, Commit, Cancel, Navigation and All.
DescriptionViewerPosition	Indicates the position of the DescriptionViewer control. Possible values are Auto, BesideLabel and BesideContent.
LabelPosition	Indicates the position of the label control in relation to the data displayed in the DataForm. Possible values are Auto, Left and Top.

buttons the blue appearance as shown in the bottom of the screen in **Figure 1**. The DataForm itself can be styled by setting the Style property. Individual aspects of the DataForm, such as the DataField or ValidationSummary, can also be styled.

By setting some of these more common properties of the DataForm as well as some basic styles, we achieve the appearance and functionality displayed in **Figure 1**. The settings for the sample application included with this article are shown in **Figure 5**. Notice that AutoEdit and AutoCommit are both set to false, thus requiring that the DataForm be explicitly put into edit mode by the user clicking on the edit button on the toolbar, and that the user click the Save button to commit the changes.

The navigation button is omitted from the CommandButtonsVisibility property, which means that the toolbar will not show the navigation options. If we include this option either by adding the keyword Navigation to the CommandButtonsVisibility or by changing the setting to the keyword All, the DataForm would also show the navigation toolbar buttons. Since the Commit and Cancel buttons are included in the CommandButtonsVisibility property, they will be visible. While the DataForm has not been edited yet, the Cancel button will be disabled as shown in **Figure 6**. The navigation buttons allow the user to move around the collection of records. However, in this sample application, the users can also navigate using the data-bound DataGrid, since it is also bound to the same set of data as the DataForm. Therefore, I turned off the navigation buttons in the toolbar purely as a design choice, because the navigation functionality can already be achieved through the DataGrid.


▾ Miscellaneous

CacheMode	
CancelButtonContent	Cancel
CancelButtonStyle	System.Windows.Style
Clip	
CommitButtonContent	Save
CommitButtonStyle	System.Windows.Style
CurrentItem	
Header	Employee Details
IsReadOnly	☐
RenderTransformOrigin	0 0
Style	
Template	System.Windows.Controls.ControlT...

▾ DataForm Styling

DataFieldStyle	
EditTemplate	System.Windows.DataTemplate
HeaderTemplate	
NewItemTemplate	
ReadOnlyTemplate	
ValidationSummaryStyle	

Figure 4 **Styling the DataForm**

DataAnnotations

A lot of the functionality of the DataForm is driven by the DataAnnotations attributes placed on the entity data bound to the DataForm. The DataAnnotations are available once you reference the assembly System.ComponentModel.DataAnnotations.dll. The DataForm examines the attributes and applies them accordingly.

Figure 5 **Customizing the DataForm**

```
<dataFormToolkit:DataForm x:Name="dataForm"
        ItemsSource="{Binding Mode=OneWay}"
        BorderThickness="0"
        FontFamily="Trebuchet MS" FontSize="13.333"
        CommitButtonContent="Save"
        CancelButtonContent="Cancel"
        Header="Employee Details"
        AutoEdit="False"
        AutoCommit="False"
        AutoGenerateFields="True"
        CommandButtonsVisibility="Edit, Add, Delete, Commit, Cancel"
        CommitButtonStyle="{StaticResource ButtonStyle}"
        CancelButtonStyle="{StaticResource ButtonStyle}"
        DescriptionViewerPosition="BesideLabel"
        LabelPosition="Left" />
```

Figure 7 shows several of the DataAnnotations attributes that can decorate properties of an entity and describes their effect.

The Display attribute feeds the DataForm's Label and DescriptionViewer with the Name and Description parameters, respectively. If omitted, the DataForm automatically generates these values for the controls using the name of the property. Notice in **Figure 8** that the Employee entity's FirstName property is decorated with 3 DataAnnotations attributes. The Display attribute indicates that the Label should display the text "First Name" and the DescriptionViewer should show a tooltip of "Employee's first name."

The ModelBase class's Validate method is invoked in every property setter so each property can be validated when set.

The Required attribute indicates that a value must be entered for the FirstName property. The Required attribute, like all validation attributes, accepts an ErrorMessage parameter. The ErrorMessage parameter indicates the message that will be displayed both in a tooltip for the invalidated control and in the ValidationSummary control in the DataForm. The StringLength attribute for the FirstName property is set to indicate that the field cannot exceed 40 characters, and if it does, an error message will be displayed.

The code in **Figure 8** shows that when a value is set on the property, the Validate method is invoked. Validate is a method I created in the base class ModelBase I used for all entities in this sample. This method validates the property by checking it against all of its DataAnnotations attributes. If any of them fail, it throws an exception, does not set the property to the invalid value, and directs the DataForm to notify the user. **Figure 1** shows the results of emptying the FirstName field, since it is required.

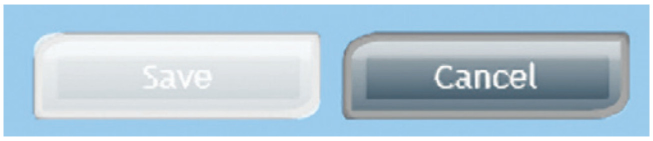


Figure 6 **Customizing the Buttons**

The ModelBase class's Validate method is invoked in every property setter so each property can be validated when set. The method accepts the new value and the name of the property being validated. These parameters are in turn passed to the Validator class's ValidateProperty method, which does the actual validation on the property value:

```
protected void Validate(object value,
  string propertyName) {
  Validator.ValidateProperty(value,
    new ValidationContext(this, null, null) {
      MemberName = propertyName });
}
```

The Validator is a static class that allows you to perform validation using several different techniques:

- The ValidateProperty method checks a single property's values and throws DataAnnotations.ValidationException if the value is invalid.
- The ValidateObject method checks all properties on the entire entity and throws DataAnnotations.ValidationException if any value is invalid.
- The TryValidateProperty method performs the same validation checks as ValidateProperty, except it returns a Boolean value instead of throwing an exception.

Figure 7 **Common DataAnnotations**

Properties	Description
Display	Indicates how the property should be displayed in the various controls in the DataForm. [Display(Name = "Age", Description = "Age of the employee")] indicates the display settings for the field label and description viewer.
Key	Specifies that the property (or properties) decorated with this attribute uniquely identify the entity. [Key] indicates that the property is a key field for the entity.
Editable	Boolean parameter can be passed indicating that the property should be editable or not in the DataForm. If not specified, the default behavior is for the DataForm to assume the property is editable (as long as it is a public property). [Editable(false)] indicates the property value is not editable in the DataForm.
Required	[Required] indicates that a value must be specified for the property.
StringLength	Specifies that the value specified for the property must not exceed the given length. [StringLength(40)] indicates a maximum length of 40 characters.
Range	Specifies that the value set for the property must not exceed the given minimum and maximum values. Range[0, 100] indicates that the values must be between 1 and 100.
RegularExpression	Specifies that the value for the property must match the given regular expression. [RegularExpression(@"^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$", ErrorMessage = "Please enter a valid email address")] indicates that the value must match the specified regular expression.
EnumDataType	Specifies that the value of the property must equal one of the given enumerator values.
CustomValidation	Indicates that the value of the property must pass a custom validation method. [CustomValidation(typeof(EmployeeValidator), "ValidateAge")] indicates that the value must pass the test expressed in the EmployeeValidator class's ValidateAge method.

Figure 8 **FirstName Property with DataAnnotations**

```
private string firstName;
[Required(ErrorMessage = "Required field")]
[StringLength(40, ErrorMessage = "Cannot exceed 40")]
[Display(Name = "First Name", Description = "Employee's first name")]
public string FirstName
{
    get { return firstName; }
    set
    {
        if (firstName == value) return;
        var propertyName = "FirstName";
        Validate(value, propertyName);
        firstName = value;
        FirePropertyChanged(propertyName);
    }
}
```

- The TryValidateObject method performs the same validation checks as ValidateObject, except it returns a Boolean value instead of throwing an exception.
- The ValidateValue method accepts a value and a collection of validation attributes. If the value fails any of the attributes, a ValidationException is thrown.
- The TryValidateValue method performs the same validation checks as ValidateValue, except that it returns a Boolean value instead of throwing an exception.

I also created a method in the ModelBase class named IsValid, which calls the TryValidateObject method, as shown below. This method can be called on the entity at any time to determine if it is in a valid state:

```
public bool IsValid() {
  return Validator.TryValidateObject(this,
    new ValidationContext(this, null, null),
      this.validationResults, true);
}
```

Figure 9 **Custom Property Validation**

```
public static class EmployeeValidator
{
  public static ValidationResult ValidateAge(int age)
  {
      if (age > 150)
          return new ValidationResult("Employee's age must be under 150.");
      else if (age <= 0)
          return new ValidationResult(
            "Employee's age must be greater than 0.");
      else
          return ValidationResult.Success;
  }
}
```

Custom Validation

When a canned attribute from the set of DataAnnotations does not suffice, you can create a custom validation method and apply it to an entire object or to a property value. The validation method must be static and return a ValidationResult object. The ValidationResult object is the vehicle that contains the message that will be bound to the DataForm. **Figure 9** shows the ValidateAge custom validation method, which verifies that the value is between 0 and 150. This is a simple example, and a Range attribute could have been used. However, creating a specific method allows it to be reused by multiple entities for validation.

Custom validation methods can also be created to validate an entire object. The ValidateEmployee method accepts the entire Employee entity and examines several properties in determining if the entity is in a valid state. This is very helpful for enforcing entity-level validation, as shown below:

```
public static ValidationResult ValidateEmployee(Employee emp) {
  string[] memberNames = new string[] { "Age", "AnnualSalary" };
    if (emp.Age >= 50 && emp.AnnualSalary < 50000)
      return new ValidationResult(
        "Employee is over 50 and must make more than $50,000.",
        memberNames);
    else
      return ValidationResult.Success;
}
```

The Validation Summary control is displayed by the DataForm when one or more ValidationExceptions are thrown (see **Figure 10**). The name of the field is highlighted in bold lettering while the validation message is displayed beside it. The style for the ValidationSummary can also be customized to use a resource, if desired.

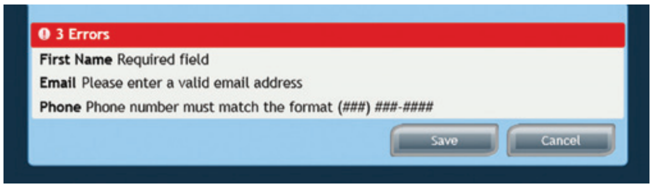

3 Errors
First Name Required field
Email Please enter a valid email address
Phone Phone number must match the format (###) ###-####

Figure 10 **Validation Summary**

Influencers

The DataForm takes its cue from DataAnnotations as well as from the interfaces that its bound data source implements. For example, the Employee entity implements the IEditableObject interface. This interface forces the implementation of the BeginEdit, CancelEdit and EndEdit methods that are invoked by the DataForm at the appropriate times while the DataForm is being edited. In the sample application, I used these methods to implement a cancel behavior so the user can press the Cancel button and restore the original entity values.

This behavior is achieved by creating an instance of the entity in the BeginEdit method (named cache) and copying the original values from the entity. The EndEdit method clears the cache object while the CancelEdit method copies all values form the cache entity back to the original entity, thus restoring its original state.

When the DataForm sees that the collection it is bound to is a CollectionViewSource, it will support editing, sorting, filtering and tracking the current record (also known as currency). The currency feature is important, as it also ensures that the DataForm does not allow a user to navigate away from an invalid entity state without fixing it and committing or canceling the changes. Binding the DataForm to a PagedCollectionView also provides these same features and also offers deleting of an item, adding an item and grouping items. Binding to an ObservableCollection will disable some of these features, as on its own the ObservableCollection does not implement the interfaces that the DataForm looks for.

If your entities are already contained within an ObservableCollection, you can create a PagedCollectionView by passing the ObservableCollection instance into the collection via its constructor, as shown below:

```
var employees = new DataService().GetPeople();
PagedCollectionView view = new PagedCollectionView(employees);
this.DataContext = view;
```

Alternatively, you can create a CollectionViewSource by passing the ObservableCollection of employees into the CollectionViewSource's collection initializer and setting the Source property, as shown below:

```
var employees = new DataService().GetPeople();
CollectionViewSource view = new CollectionViewSource
  {Source = employees};
this.DataContext = view;
```

Wrapping Up

The DataForm adds a lot of value to applications and to the normally code-intensive and monotonous process of navigating, editing and viewing data. Its features are customizable to suit your needs, and it can be styled visually to blend in with your application's design. ■

JOHN PAPA *(johnpapa.net) is a senior consultant and a baseball fan who spends summer nights rooting for the Yankees with his family. Papa, a Silverlight MVP, Silverlight Insider and INETA speaker, has authored several books, including his latest, "Data-Driven Services with Silverlight 2" (O'Reilly, 2009). He often speaks at conferences such as VSLive!, DevConnections and MIX.*

DINO ESPOSITO

Data Binding in ASP.NET AJAX 4.0

Stop beating around the bush: AJAX is possible only with a strong JavaScript engine that runs within the client browser and provides the foundation for more advanced and asynchronous features. The JavaScript library currently incorporated in ASP.NET 3.5 Service Pack 1 is a necessary, but insufficient, attempt to deliver such a library. A more powerful ASP.NET AJAX platform is required, and it is just being introduced as part of ASP.NET AJAX 4.0.

Abstractly speaking, an AJAX-based front-end is a presentation layer that combines rich UI capabilities with the implementation of some application logic. The application logic is essentially the code behind all the use-case diagrams that resulted from the design and analysis phases. The application logic expresses the intended behavior of the application and how a user is expected to interact with the whole system.

What makes an AJAX front-end fairly unique, at least when compared to a classic Web or smart-client front-end, is the pressing need to mix elements of a rich user experience with low-level programming tools. Because an AJAX front-end runs within the client browser, it can rely only on HTML to produce the user interface and only on JavaScript to spice up the visual elements with special effects, drag-and-drop, asynchronous data fetch and partial updates.

To meet expectations, a modern and effective AJAX platform must provide two key capabilities. First, it must enable developers to place asynchronous calls to a HTTP façade of ad hoc server modules. Second, it must enable developers to incorporate any received raw data into the existing page Document Object Model (DOM). Both capabilities, though, would lose much of their inherent appeal if implemented in a way that isn't that easy and effective.

In ASP.NET 3.5 Service Pack 1, developers find a powerful and reliable API to connect asynchronously to a layer of HTTP-based Web services. ASP.NET AJAX 3.5 makes it possible, and easy overall, for you to reference a Web service from the client page. When you do so, the framework also automatically generates a JavaScript proxy class that mirrors the service contract. The existing AJAX framework, both on the server and client, works to shield developers from all details of data serialization. From the perspective of the JavaScript developer, a remote Web service (still subject to the well-known same-origin policy) is like a local JavaScript object exposing behavior through methods.

ASP.NET 3.5 Service Pack 1 doesn't offer the same virtues as far as the building of the user interface is concerned. It makes it very easy to fetch raw data from the server, but it doesn't offer much in the way of a powerful interface to display this raw data in a user interface. The major weakness of the AJAX support in ASP.NET 3.5 Service Pack 1 is the lack of effective tools for client-side data binding and HTML templates. That is why an engine for client-side template rendering and a made-to-measure data binding syntax are the most compelling features you find in ASP.NET AJAX 4.0.

In this article, I'll review the pillars of real-world AJAX development as supported in ASP.NET AJAX 4.0. In doing so, I'll mostly focus on client-side templates and data binding but won't ignore other goodies, such as ADO.NET Data Services proxy classes and programming facilities.

Column Disclaimer: This column is based on a prerelease version of ASP.NET 4.0. Details are subject to change.

Code download available at msdn.microsoft.com/mag1009CutEdge.

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

Pillars of Real-World AJAX Development

Real-world AJAX development is about building rich user interfaces over the Web, and it requires the application of new design patterns and employment of new programming tools.

For a long time, any Web user interface represented a whole step backward in terms of usability and responsiveness if compared to any desktop user interface. For a long time, Web developers just ignored (because it was not relevant to their work) a number of UI patterns and programming features, including predictive fetch, caching, monitoring of remote tasks, context-sensitive and drill-down display, subviews, partial UI disablement, and modality.

In classic Web development, the building of the user interface was entirely delegated to the server side and effectively implemented using server-side data binding and ad hoc controls. The advent of the AJAX paradigm made this mode obsolete and unappealing for increasing numbers of applications.

Data binding, though, is too powerful a feature to overlook in an AJAX programming model. Also, object-orientation is hard to refuse when the complexity of code grows beyond a given threshold. At the same time, a Web application remains a pretty unique combination of small footprints, cacheable downloads and rich capabilities.

On the way to real-world AJAX development, JavaScript libraries are the only affordable way to add programming power. Through JavaScript libraries, you provide the foundation of object-orientation in a non-OOP language; you offer rich and off-the-shelf UI widgets; and you can offer programming tools to effectively code data binding entirely on the client-side.

Without a powerful model for client-side data binding, you can't have a powerful platform for real-world AJAX development.

Requirements for Client-Side Data Binding

There are two fundamental patterns for implementing data binding functionalities. One is the HTML Message pattern, and the other is the Browser-side Template pattern. The former entails making a remote service call to a component that returns prearranged HTML markup ready for display. The latter is all about setting up machinery to download raw data and decide on the client how to render it out.

The HTML Message pattern is similar to a smart form of partial rendering, except that it doesn't involve any view state and can be configured to be an autonomous operation not bound to any other postback operations occurring in the same application. In an implementation of the HTML Message pattern, everything takes place on the server; any data binding is essentially a form of classic server-side data binding involving controls such as DataGrid and ListView and managed containers of fetched data.

The HTML Message pattern can be easily implemented with the tools available in ASP.NET 3.5. All that it requires on the client is the binding of returned markup to the page DOM. The code snippet below shows what's really required from a coding perspective:

```
grid.innerHTML = markup;
```

In the example, grid indicates the HTML element to contain the markup—typically this is a DIV tag. The variable named markup, on the other hand, indicates any chunk of HTML obtained as a response from a service method call.

It goes without saying that the service used to implement the HTML Message pattern must incorporate both the logic to retrieve or calculate the data to return, plus any logic required to format the data to HTML.

In general, a solution based on the HTML Message pattern requires more bandwidth as the average size of the response for each remote method call increases.

The Browser-side Template pattern (BST) requires more coding on your side but can also deliver better results both in terms of flexibility and bandwidth optimization. The BST pattern is based on the idea that you place a remote call to retrieve data. Data is then downloaded on the client in a format that allows manipulation with JavaScript. Finally, data is merged with the existing page DOM and produces any complex interface you need.

Too often, the power of the AJAX paradigm is mistakenly represented with the possibility of asynchronously updating small portions of the user interface. It is one thing to get a scalar value asynchronously (say, the current balance of a bank account) and insert that into the existing page DOM; it is quite another thing to refresh asynchronously an array of data that changes frequently and requires a gridlike infrastructure to display.

The server side is simple with server controls, as below:

```
Collection<StockInfo> stocks = GetLatestQuotes(...);
DataGrid1.DataSource = stocks;
DataGrid1.DataBind();
```

What would be the equivalent of such code for the client side? The first part can be easily mapped to the features of ASP.NET 3.5. All you do is instantiate and use a client proxy for a remote service that is capable of getting you up-to-date values, like so:

```
var service = new Samples.Services.FinanceService();
var stocks = service.GetLatestQuotes(...);
```

The stocks variable is a JavaScript array of objects that represents the data you received. How would you fit this chunk of raw data into an existing HTML layout? The BST pattern is here to help. It requires that you define the following elements: your own syntax for HTML templates and related data placeholders; your own syntax for binding actual data to placeholders; an HTML factory that takes templates and data and produces updated markup; and glue code to tie it up all together while offering a manageable programming interface.

ASP.NET AJAX 4.0 provides an implementation of the BST pattern out of the box. The scaffolding for ASP.NET AJAX templates is defined in the MicrosoftAjaxTemplates.js file. You need to reference this file via the ScriptManager control (or ScriptManagerProxy if you use master pages). If you use ASP.NET MVC or prefer to reference script files via the traditional <script> tag, then you must also add a preliminary reference to MicrosoftAjax.js.

Syntax for HTML Templates

Years of ASP.NET programming have proven beyond any reasonable doubt that HTML templates are an excellent way to create a Web user interface from data. An HTML template is a piece of HTML that contains literals, ASP.NET controls and placeholders for binding data. Bound to raw data and processed by an ad hoc engine, an HTML template morphs into plain HTML for the browser to render. A template exists to bind data and originate a chunk of markup to display; until binding occurs, the template is hidden from view.

In spite of a relatively simple description, an HTML template is quite difficult to implement in a real-world AJAX framework. A few attempts have been made by popular libraries, such as Prototype JS, to formalize an HTML template. While there's common agreement on the features one should expect from an HTML template, a common model for defining an HTML template in a browser doesn't exist yet.

A template must be able to render XHTML-compliant markup. A template must be processed as quickly as possible by the underlying rendering engine and should let the engine render a large percentage of the markup before the user realizes there's a delayed response from the application. A template must support a very simple syntax for binding that is easy to read, while not being limited to simple cases only. You should be able to mix markup and code in a template. Ideally, the code that triggers the rendering of the template should be declarative and not particularly intrusive.

Let's review the characteristics of the HTML template model supported by ASP.NET AJAX 4.0.

In ASP.NET AJAX 4.0, an HTML template is a DIV tag, or any other container tag, decorated with the sys-template class attribute, as shown below:

```
<div>
   <ul id="MyItem" class="sys-template">
       <li>
         {{ Symbol }}, {{ Quote }}, {{ Change }}
       </li>
   </ul>
</div>
```

The sys-template CSS class is a convention that marks the element, and its content, as initially invisible. Note that the sys-template must be defined explicitly in the master page, or in every page, as follows:

```
<style type="text/css">
   .sys-template { display:none; visibility:hidden; }
</style>
```

When rendered, a template is given a data context and the body of the template can host bindings to public fields and properties of the data context object. Likewise, any elements in the template can reference any JavaScript expression that evaluates to a string.

Syntax for Data Bindings

The syntax to express a binding between a placeholder in the template and external data is as follows:

```
{{ expression }}
```

As mentioned, the expression can be the name of a public member of the data context object or a JavaScript expression that returns a string. Such expressions can appear anywhere in the template and can also be used to assign a value to an attribute, as shown below:

```
<div>
  <ul id="MyItem" class="sys-template">
     <li>
       <asp:Label runat="server"
            Text="{{CompanyName}}" />
     </li>
  </ul>
</div>
```

An HTML template doesn't have to be made of plain HTML literals; you can use ASP.NET markup as well. Given the preceding code snippet, here's the markup emitted by the Label control:

```
<span>{{ CompanyName }}</span>
```

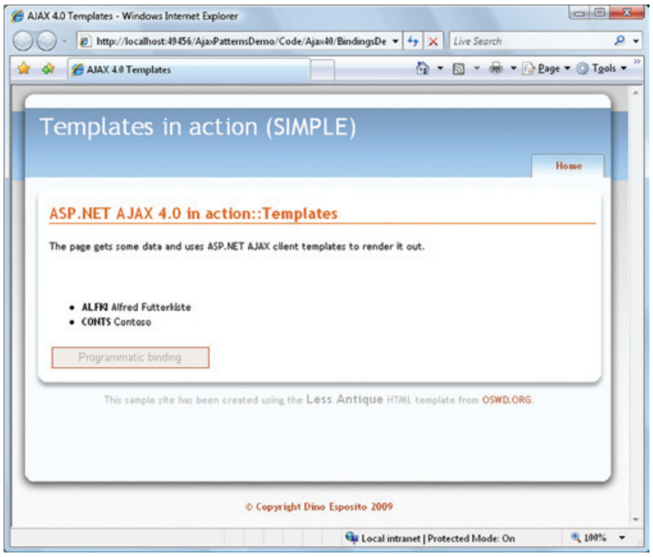
As you can see, the use of a server control in the source code of the page doesn't hinder client-side rendering of templates.

Client events can be defined within a template using the familiar onXXX syntax or via the $addHandler function in the Microsoft AJAX Library.

DataView Control and Template Instantiation

To display data, a template must be instantiated, bound to data and rendered within a container. The Sys.UI.DataView client control can be used to automate and simplify all these tasks.

The DataView control is essentially a component that takes some input data and the ASP.NET AJAX template and produces HTML markup to be displayed within the page. The DataView is also referenced as a behavior component. In general, a behavior is a script-based component that, once attached to a DOM element, changes the way in which the HTML element works within the client browser. You can work with a behavior in either of two ways. You can declaratively attach the behavior to its target DOM


AJAX 4.0 Templates - Windows Internet Explorer
http://localhost:49456/AjaxPatternsDemo/Code/Ajax40/BindingsDe
Live Search
AJAX 4.0 Templates
Page
Tools
Templates in action (SIMPLE)
Home
ASP.NET AJAX 4.0 in action::Templates
Programmatic binding
© Copyright Dino Esposito 2009
Local intranet | Protected Mode: On
100%

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

- ALFKI Alfred Futterkiste
- CONTS Contoso

This sample site has been created using the Less Antique HTML template from OSWD.ORG.

Figure 1 **The DataView Behavior in Action**

element, or you can create an instance of the behavior and configure it programmatically. In the latter case, the association between the behavior and the template is just part of the configuration. Let's tackle the declarative approach first.

Before I go any further, though, let me clarify that the DataView is just one possible behavior component. Anything you read later on regarding declarative instantiation and attachment applies to any behavior you may run across in ASP.NET AJAX.

Attaching Behaviors Declaratively

To attach behaviors to a DOM element, you use the sys:attach custom attribute. As you can see, the attribute is associated with a namespace URI that makes it XHTML compliant. You declare the sys prefix in the <body> element:

```
<body xmlns:sys="javascript:Sys" ...>
```

The sys prefix maps to the javascript:Sys namespace URI defined in the Microsoft AJAX library. Using the sys:attach attribute serves only the purpose of establishing an association between an existing behavior and an HTML element. You still need to instantiate the behavior component. You do that by defining another custom namespaced attribute in the <body> element. The value of the attribute will reference the JavaScript object to instantiate:

```
<body xmlns:sys="javascript:Sys"
      xmlns:dataview="javascript:Sys.UI.DataView" ...>
```

The name of the attribute—dataview, in the preceding code snippet—is arbitrary and can be changed to anything else you like. Whatever name you pick, however, it must be maintained in the remainder of the code to reference the behavior.

The effective instantiation of any attached script behavior occurs when the page is loaded and the DOM element gets processed. The browser that is loading the page may not know anything about how to handle such library-defined behaviors. The ASP.NET AJAX framework is ultimately responsible for the instantiation of its own behaviors. However, the ASP.NET AJAX framework requires ad hoc instructions from you to proceed. In particular, you want the framework to kick in

Figure 2 **Controlling a DataView Object Programmatically**

```
<script type="text/javascript">

// Define a global instance of the DataView
var theDataView = null;

// This function can be called from anywhere in the page to
// fill the template with passed data and update the page.
function renderTemplate(dataSource)
{
    // Ensure we have just one copy of the DataView.
    // The DataView's constructor gets a DOM reference to the template.
    if (theDataView === null)
        theDataView = new Sys.UI.DataView($get("MyTemplate"));

    // Bind data to the template associated with the DataView
    theDataView.set_data(dataSource);

    // Force the template to render
    theDataView.refresh();
}

</script>
```

during the DOM parsing process and have it look in the content of any parsed element to take care of any sys:attach attributes.

For performance reasons, the ASP.NET AJAX framework is not designed to automatically take care of any DOM element the browser encounters along the way. It is therefore up to you to explicitly indicate which DOM elements need to be scanned for attached behaviors that support declarative instantiation. You enable a DOM element to contain declaratively instantiated behaviors by using the sys:activate attribute. You use the attribute in the <body> element and set it to a comma-separated list of element IDs:

```
<body xmlns:sys="javascript:Sys"
      xmlns:dataview="javascript:Sys.UI.DataView"
      sys:activate="customerList">
```

The preceding code instructs the framework to automatically instantiate any behavior that may be attached to the customerList DOM element.

You can also use the wildcard symbol (*) if you want to activate the whole document. Use this option with care, especially on large pages because it may introduce a rendering delay. With the DataView behavior fully configured in the <body> element, all that remains to do is to bind fresh data to the template:

```
<div id="customerList">
    <ul class="sys-template"
        sys:attach="dataview"
        dataview:data="{{ theCustomers }}">
        <li>
            <span ><b>{{ ID }}</b></span>
            <asp:label runat="server"
                 Text="{{ CompanyName }}"></asp:label>
        </li>
    </ul>
</div>
```

The DataView component has a rich programming interface that, among other things, includes a data property. The data property represents the data context for the bound template. You can set the data property both declaratively and programmatically. In the code snippet below, the data property is declaratively set to the content of a global array named theCustomers:

```
<ul class="sys-template"
    sys:attach="dataview"
    dataview:data="{{ theCustomers }}">
...
</ul>
```

In general, you can set the data property to any JavaScript expression that evaluates to a bindable object or an array of objects. You opt for a declarative approach when you intend to bind the template to some global data.

The code download sample titled A Sample Page Using Declarative Binding shows the full listing of a sample page. **Figure 1** shows the sample page in action.

For programmatic binding, you need to obtain a reference to the DataView object and call the setter method of the data property. You retrieve the DataView instance via the $find helper defined in the Microsoft AJAX library. The ID of the DataView component matches the ID of the DOM element it is attached to. Consider the following code snippet:

```
<ul class="sys-template"
    sys:attach="dataview"
    id="DataView1">
...
</ul>
<input type="button"
       value="Perform binding"
       onclick="bind()" />
```

You can perform the data binding only when the user explicitly clicks the button. Here's the code you need to put in the bind JavaScript function:

```
<script type="text/javascript">
    function bind() {
        theDataView = $find("DataView1");
        theDataView.set_data(theCustomers);
    }
</script>
```

You first retrieve the DataView instance by ID and then assign new content to its data property.

Tips for Master Pages

When you use a master page, you usually leave the <body> tag of the page template in the master. It is then necessary that you edit the master page to attach any required behavior. Alternatively, you can put the <body> tag in a content placeholder and force developers to set it explicitly in any content page.

The <body> tag is required to register attachable behaviors, such as DataView, and to enable DOM elements to recognize and instantiate behaviors through the sys:activate attribute.

It should be noted that an alternative exists to editing the <body> tag. You can use the new ClientElementsToActivate property on the ScriptManager control, as follows:

```
<asp:ScriptManager runat="server"
                   ID="ScriptManagerProxy1"
                   ClientElementsToActivate="*">
    <Scripts>
        <asp:ScriptReference Name="MicrosoftAjaxTemplates.js" />
    </Scripts>
    ...
</asp:ScriptManager>
```

Note that only a few of the properties defined on the ScriptManager control are mirrored on the ScriptManagerProxy control, which is the wrapper control you use to replicate ScriptManager functionality in a content page. In ASP.NET 4.0, however, the ClientElementsToActivate is among the few properties you can access on both controls.

You might be wondering how a string array property like ClientElementsToActivate can affect the behavior of a JavaScript framework, such as the ASP.NET AJAX framework. When the ClientElementsToActivate property is set, the script manager control emits an extra line of script within the page that adds one or more entries to an internal array on the Sys.Application object in the Microsoft AJAX Library.

Using the DataView Control Programmatically

So far, we've examined a scenario in which the HTML template is attached to a DataView behavior for mixing together layout and data into fresh HTML. This is not the only possible approach to client-side data binding.

You can also create an instance of the DataView component programmatically and pass the template to it to use as an argument. **Figure 2** provides a brief but illustrative listing.

The constructor of the DataView takes a DOM reference to the template to use internally. The setter method of the data property gets fresh data to bind. Finally, the refresh method forces an update of the HTML template that displays freshly bound data.

To create an instance of the DataView, you can also resort to the $create helper method, as shown below:

```
<script type="text/javascript">
function pageLoad() {
   $create(
       Sys.UI.DataView,
       {},
       {},
       {},
       $get("MyTemplate")
   );
}
</script>
```

A very common scenario in which using a DataView definitely smoothes your programming efforts is to populate a complex HTML template with data provided by a remote Web service. Let's investigate this point further.

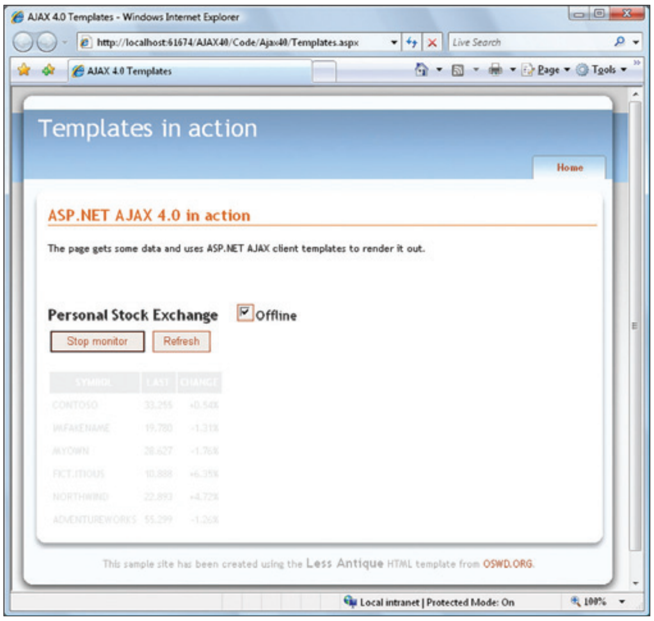
Fetching Data from Web Services

The programming interface of the DataView component features a number of members specifically designed to serve the scenario in which the content of the DataView comes from a remote URI. The download includes a list of members of the DataView that work with Web Services.

Note that the content of the code download sample titled An Auto-Fetch DataView to Consume Data from a Remote URI doesn't fully comprehend the entire API of the DataView component. Many more members exist to serve other scenarios that I'll cover in depth in a future column.

The code download sample titled An Auto-Fetch DataView to Consume Data from a Remote URI shows the JavaScript code that creates a DataView when the page loads. The DataView is configured to fetch its data immediately by placing a call to the specified Web service method.

The dataProvider property indicates the data source, whereas the fetchOperation sets the method to invoke. Finally, fetchParameters is an dictionary object that contains parameter information for the method. You set it to a JavaScript object where each


AJAX 4.0 Templates - Windows Internet Explorer
http://localhost:61674/AJAX40/Code/Ajax40/Templates.aspx
Live Search
AJAX 4.0 Templates
Page
Tools
Templates in action
Home
ASP.NET AJAX 4.0 in action
Personal Stock Exchange
Offline
Stop monitor
Refresh
Local intranet | Protected Mode: On
100%

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

SYMBOL	LAST	CHANGE
CONTOSO	33.255	+0.54%
IMFAKENAME	19.780	-1.31%
MYOWN	28.627	-1.76%
FICT.ITIOUS	10.888	+6.35%
NORTHWIND	22.893	+4.72%
ADVENTUREWORKS	55.299	-1.26%

This sample site has been created using the Less Antique HTML template from OSWD.ORG.

Figure 3 **A Rich Data-Bound Web Page Using AJAX and jQuery**

property matches the name of a formal parameter on the method definition. In particular, in the previous example, the method GetQuotesFromConfig has the following prototype:

```
[OperationContract(Name="GetQuotesFromConfig")]
StockInfo[] GetQuotes(bool isOffline);
```

The DataView receives the HTML template to populate via the $create helper function and once data has been successfully fetched, it fills up any placeholders in the template.

The sample code also contains a custom timer object that periodically refreshes the template with a bit of jQuery animation. When the timer ticks, a new fetch operation is ordered using any current value for the "offline" checkbox. The "offline" checkbox indicates whether the Web service is expected to return fake stocks and values or connect to a real-world finance service to grab quotes for true stocks. The service gets the URL of the finance service and the list of stocks from the configuration file. (See the source code for details.)

Next Month

Figure 3 shows a rich and animated data-bound ASP.NET AJAX page that uses client-side rendering to bind raw data to a relatively complex HTML template. In ASP.NET AJAX 4.0, you find powerful programming tools to make coding such pages no more difficult than doing traditional server-side data binding.

The page shown in **Figure 3** is only the first, and the simplest, step of a longer path. Next month, I'll discuss more features of rich templates and how to put logic in HTML templates, and I'll explore live data binding. Stay tuned! ■

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, Esposito is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos.*

Visual Studio 2010 Tools for SharePoint Development

Steve Fox

SharePoint development has been a bit of a mystery to many developers, who have felt that developing for the platform was cumbersome and out of their reach. The developer community has also been split over what tool set to use. For example, some developers have used a combination of class libraries, manual project folders with XML configuration files, and post-build output events to generate features and solutions for SharePoint. Other developers have used STSDEV, a community tool, or Visual Studio Extensions for Windows SharePoint Services (VSeWSS) to build different applications and solutions and deploy them to SharePoint. In other words, developers could follow numerous paths to deploying features and solution packages to SharePoint. Challenges notwithstanding, the SharePoint developer community has grown to a significant number—roughly 600,000 developers—and continues to grow. Looking forward, Visual Studio 2010 will offer developers a great entry into SharePoint development with the new SharePoint tools that will ship in the box.

Disclaimer: This article is based on prerelease versions of Visual Studio 2010 and SharePoint 2010. All information is subject to change.

Technologies discussed:

Visual Studio 2010, SharePoint 2010

This article discusses:

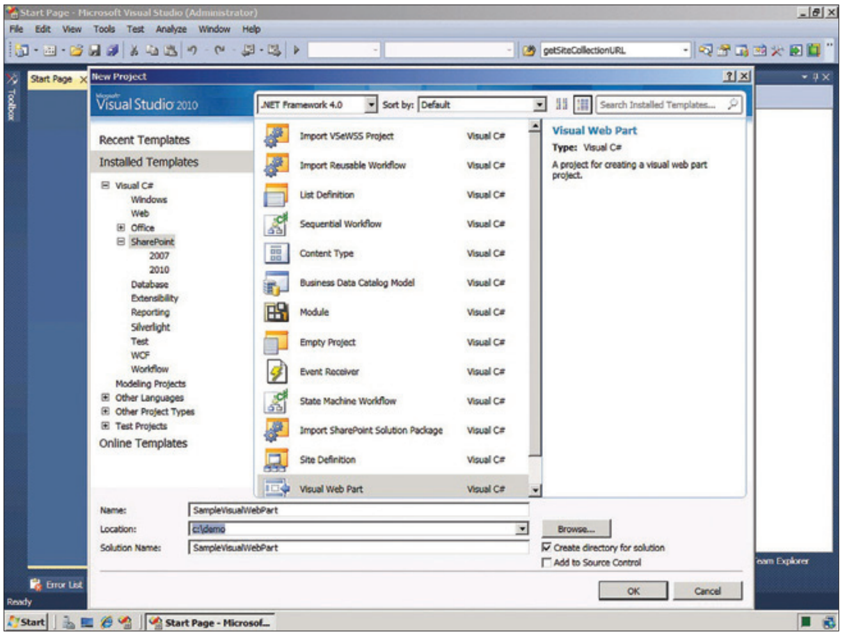
SharePoint tooling in Visual Studio 2010, Developing a visual Web Part, Deploying a SharePoint feature

SharePoint 2010 is a major step forward as a development platform, not only because of the rich set of features the platform supports, but also because significant investments have been made in the suite of tools designed to make the development process more productive and more accessible to developers of all skill levels. The two core developer tools for SharePoint 2010 are SharePoint Designer 2010 and Visual Studio 2010. (The companion tool set for designers is the Expression suite.) This article provides a first look at SharePoint 2010 development, introducing you to SharePoint tooling in Visual Studio 2010 (including a glimpse at the new project templates) and illustrating how to create and deploy a sample visual Web Part.

SharePoint Tools in Visual Studio 2010

A number of areas for SharePoint developers in Visual Studio 2010 are worth mentioning. First, you get SharePoint project templates in the box, so you can start right away on solution development. Second, tooling has standardized on the Windows SharePoint Package (WSP) packaging standard, so when you import or deploy a solution to SharePoint, Visual Studio treats it as a solution package. Third, some great deployment and packaging features, such as solution retraction and custom deployment configurations, ship with the SharePoint tools in Visual Studio 2010. And last, the new SharePoint Explorer provides a view into native and custom artifacts (for example, lists and workflows) that exist on your SharePoint server. This is, of course, a short list of the features that represent a major extension of a Visual Studio tool set designed to reach into a community and make it easier for SharePoint developers to get up and running.

Also worth mentioning are a few of the SharePoint 2010 enhancements, which can definitely be used in the context of Visual Studio 2010. For example, the new client object model enables you to access SharePoint objects through a referenced DLL as opposed to Web service calls. (In SharePoint 2007, you access SharePoint list data, for example, by using an ASP.NET Web service.) Also, LINQ for SharePoint brings the power of LINQ to SharePoint, letting you treat lists, for example, as strongly typed objects. Further, Silverlight (especially in combination with the client object model) is supported natively in SharePoint 2010—no more messing around with the web.config to get started with this development. And sandboxed solutions also offer a way to build SharePoint Web Parts and deploy them to a site without needing administrative intervention—that is, you can deploy a Web Part to a SharePoint site and have it run in the context of that site either in an on-premises instance of SharePoint or in the cloud using the hosted version of SharePoint. Finally, external data lists make interacting with line-of-business systems a read/write process, and while seemingly small, this is a huge leap forward given the tools support that enables you to build line-of-business integrations quickly and efficiently. For each of these innovations in SharePoint 2010, Visual Studio 2010 provides some measure of support, whether through project templates or APIs, for professional developers. If there's a time to pick up SharePoint development, it's now.



Visual Web Part
Type: Visual C#
A project for creating a visual web part project.

Name: SampleVisualWebPart
Location: c:\demo
Solution Name: SampleVisualWebPart

Figure 1 **New SharePoint Project Templates**

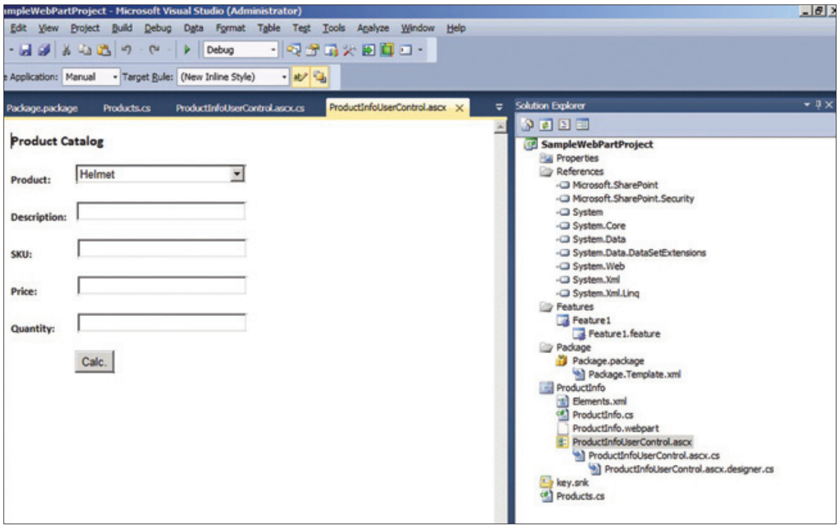


Figure 2 **Designer View for a Visual Web Part**

Developing a Visual Web Part Project

One of the most common artifacts that developers build and deploy in SharePoint is the Web Part. This makes sense given that Web Parts are one of the core building blocks for SharePoint. Because SharePoint is built on top of ASP.NET, the Web Part inherits its key features from the ASP.NET Web Part architecture.

One of the new project templates in Visual Studio 2010 is the Visual Web Part project template, which enables developers to visually design a Web Part that can be deployed to SharePoint. If you're new to SharePoint, this is a great way to get started building custom applications for SharePoint 2010. The visual Web Part I'll demonstrate has some self-contained code that calculates product costs and lists information in a simple Web Part UI.

Make sure you have the beta 2 of Visual Studio 2010 and the beta 2 of SharePoint 2010 installed on 64-bit Windows Server 2008. Open Visual Studio 2010, click File, New Project, and then navigate to the SharePoint node in the Installed Templates section. **Figure 1** shows the different types of project templates that are available. For example, the Import VSeWSS Project template provides an upgrade path from your current VSeWSS projects; the workflow templates enable you to create and deploy workflow projects to SharePoint; the Site Definition template provides site-level infrastructure that you can build out and deploy; and the Import SharePoint Solution Package is the template that enables you to import WSPs for redeployment to a local server instance. For this walk-through, select the Visual Web Part project template, provide a name (for example, SampleWebPartProject) and location for your project, and then click OK.

After you create a project, Visual Studio 2010 creates a number of default files. Expand the project nodes in the Solution Explorer to see the files. The key files you'll work with in this article are in the SampleWebPartProject node. Note that the default visual Web Part is called VisualWebPart1. To change this, right-click the VisualWebPart1 node in the Solution Explorer, select Rename and then enter a new name for your Web Part.

When you create a project, you configure it to be associated with a particular server instance.

Also note in the Solution Explorer the presence of the Features and Package nodes. These are new infrastructural parts to Visual Studio 2010 that package a SharePoint solution using a SharePoint feature. For developers new to SharePoint, a feature organizes your application in a way that SharePoint understands. Features can be deployed to SharePoint at the site or Web level, for example. A feature is structured through a set of XML configuration files, and it also references (depending on the level of trust for your application) the assembly from the global assembly cache (GAC). Specifically, each feature has its own folder in the SharePoint folder hierarchy, and the configuration files live within that folder and provide the necessary metadata for the feature. The package contains features and other assets and is used when you deploy solutions to SharePoint. The package is also where the assembly deployment location is determined. Visual Studio 2010 introduces a package designer, which makes viewing and managing packages much easier. If you double-click the Package node, the designer opens. The designer provides the ability for you to add and remove features from your deployable package. This designer represents a significant step forward in helping developers shape their SharePoint solutions through the addition of features.

Switch back to the Solution Explorer view, right-click the ProductInfoUserControl.ascx file, and then choose View in Designer. This opens a view in which you can drag and drop controls from the toolbox onto the Web Part designer surface. You'll notice three views: Design, Split, and Code. In this example, I added (by typing) a title and some controls, including text boxes and a button to calculate the cost of the product. I also typed in labels for the controls that were added to the page (see **Figure 2**).

After you complete your visual Web Part layout, you can add event handlers for the button. But before we do that, let's quickly take a look at the source code for the visual Web Part. As you can see from the code excerpt in **Figure 3,** Visual Studio adds some automatic styling to the UI in the form of CSS syntax. You can also see the actual controls (and in the case of the drop-down list, the collection of items) that make up the UI. Note that for brevity, I've removed the directives that are autogenerated and included at the top of the source.

To add event handlers to the Web Part, double-click the button. This takes you to the code behind. It also adds an onClick event to the ASCX control design. For example, in **Figure 3** note the onclick="btnCalcPrice_Click" event that is included within btnCalcPrice. The code behind, which is listed in **Figure 4**, contains some simple code that enables you to calculate the price of the product that is selected in the list box. Key parts of the code are the class-level variables (the doubles), which represent the longhand way I used to calculate the product cost; the List of Products collection (which holds a number of Products objects that are added to the

Figure 3 **Source Code for SalaryCalcWebPartUserControl.ascx**

```
<style type="text/css">
    .style1
    {
        font-family: Calibri;
        font-size: medium;
        font-weight: bold;
    }
    .style2
    {
        font-family: Calibri;
        font-size: small;
        font-weight: bold;
    }
</style>

<p class="style1">
    Product Catalog</p>
<p class="style2">
    Product:&nbsp;&nbsp;
    <asp:DropDownList ID="dropdwnProducts" runat="server" Height="20px"
        style="margin-left: 21px" Width="200px">
        <asp:ListItem>Helmet</asp:ListItem>
        <asp:ListItem>Stick</asp:ListItem>
        <asp:ListItem>Skates</asp:ListItem>
        <asp:ListItem>Elbow Pads</asp:ListItem>
        <asp:ListItem>Kneepads</asp:ListItem>
    </asp:DropDownList>
</p>
<p class="style2">
    Description:     <asp:TextBox ID="txtbxDescription" runat="server"
        Width="200px" Enabled="False"></asp:TextBox>
</p>
<p class="style2">
    SKU:
    <asp:TextBox ID="txtbxSKU" runat="server" style="margin-left: 48px"
        Width="200px" Enabled="False"></asp:TextBox>
</p>
<p class="style2">
    Price:<asp:TextBox ID="txtbxPrice" runat="server"
      style="margin-left: 48px"
        Width="200px" Enabled="False"></asp:TextBox>
</p>
<p class="style2">
    Quantity:
    <asp:TextBox ID="txtbxQuantity" runat="server" Width="200px"
Enabled="False"></asp:TextBox>
</p>
<p class="style1">
    <asp:Button ID="btnCalcPrice" runat="server"
      onclick="btnCalcPrice_Click"
        Text="Calc." />
</p>
```

list box); and the btnCalcPrice_Click event. When the page is loaded in SharePoint, the code calls the generateProductList method, which populates the list box. The btnCalcPrice_Click event then calculates the cost of a specific product—depending on what the user selected—and displays the information in the list box in the UI.

When a user clicks the button, the Web Part performs a postback to execute the event—here, calculating the product cost. What's probably more interesting than the code in **Figure 4**, which in essence is fairly simple, is how the Web Part surfaces this code in the actual Web Part. Given that what we've done is create an ASP user control for our Web Part that includes a skin and code behind, the project structure still has the actual Web Part that must surface this control. To do this, Visual Studio creates a string called _ascxPath, which represents the path to the ASCX user control that is located within the SharePoint 2010 folder hierarchy. Notice also that in the CreateChildControls method, an instance of a control is created and set to the path to the user control (using the LoadControl method). It is then added to the Controls collection

Figure 4 **Source Code for ProductInfoUserControl.ascx.cs**

```
using System;
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Collections.Generic;
using System.Data;

namespace SampleWebPartProject.ProductInfo
{
    public partial class ProductInfoUserControl : UserControl
    {
        double tax = .11;
        double totalCost = 0.0;
        List<Products> lstOfProducts = new List<Products>();

        protected void Page_Load(object sender, EventArgs e)
        {
            generateProductList();
        }

        private void generateProductList()
        {

            lstOfProducts.Add(new Products()
            { strName = "Helmet", strDescr = "Hockey helmet.", strSKU =
                "KLSONHELMT1224", dblPrice = 59.00, intQuantity = 28 });
            lstOfProducts.Add(new Products()
            { strName = "Skates", strDescr = "Hockey skates.", strSKU =
                "SKATWOKSH0965", dblPrice = 438.00, intQuantity = 88 });
            lstOfProducts.Add(new Products()
            { strName = "Stick", strDescr = "Composite hockey stick.",
                strSKU = "STIK82910JJKS", dblPrice = 189.99, intQuantity =
                35 });
            lstOfProducts.Add(new Products()
            { strName = "Elbow Pads", strDescr = "Hockey elbow pads.",
                 strSKU = "ELBOP563215NN", dblPrice = 34.00, intQuantity =
                12 });
            lstOfProducts.Add(new Products()
            { strName = "Knee Pads", strDescr = "Hockey knee pads.",
                strSKU = "KPDS7827NNJS1", dblPrice = 47.99, intQuantity =
                44 });
        }


        protected void btnCalcPrice_Click(object sender, EventArgs e)
        {
            double dblCost = 0;
            string strPrice = "";

            if (dropdwnProducts.SelectedValue == "Helmet")
            {
                dblCost = lstOfProducts[0].dblPrice;
                totalCost = dblCost + (dblCost * tax);
                System.Math.Round(totalCost, 2);
                strPrice = "$" + totalCost.ToString();

                txtbxDescription.Text = lstOfProducts[0].strDescr.
                    ToString();
                txtbxSKU.Text = lstOfProducts[0].strSKU.ToString();
                txtbxPrice.Text = strPrice;
                txtbxQuantity.Text = lstOfProducts[0].intQuantity.
                    ToString();

            }
            else if (dropdwnProducts.SelectedValue == "Skates")
            {
                dblCost = lstOfProducts[1].dblPrice;
                totalCost = dblCost + (dblCost * tax);
                System.Math.Round(totalCost, 2);
                strPrice = "$" + totalCost.ToString();

                txtbxDescription.Text = lstOfProducts[1].strDescr.
                    ToString();
                txtbxSKU.Text = lstOfProducts[1].strSKU.ToString();
                txtbxPrice.Text = strPrice;
                txtbxQuantity.Text = lstOfProducts[1].intQuantity.
                    ToString();
            }
            else if (dropdwnProducts.SelectedValue == "Stick")
            {
                dblCost = lstOfProducts[2].dblPrice;
                totalCost = dblCost + (dblCost * tax);
                System.Math.Round(totalCost, 2);
                strPrice = "$" + totalCost.ToString();

                txtbxDescription.Text = lstOfProducts[2].strDescr.
                    ToString();
                txtbxSKU.Text = lstOfProducts[2].strSKU.ToString();
                txtbxPrice.Text = strPrice;
                txtbxQuantity.Text = lstOfProducts[2].intQuantity.
                    ToString();
            }
            else if (dropdwnProducts.SelectedValue == "Elbow Pads")
            {
                dblCost = lstOfProducts[3].dblPrice;
                totalCost = dblCost + (dblCost * tax);
                System.Math.Round(totalCost, 2);
                strPrice = "$" + totalCost.ToString();

                txtbxDescription.Text = lstOfProducts[3].strDescr.
                    ToString();
                txtbxSKU.Text = lstOfProducts[3].strSKU.ToString();
                txtbxPrice.Text = strPrice;
                txtbxQuantity.Text = lstOfProducts[3].intQuantity.
                    ToString();
            }
            else if (dropdwnProducts.SelectedValue == "Knee Pads")
            {
                dblCost = lstOfProducts[4].dblPrice;
                totalCost = dblCost + (dblCost * tax);
                System.Math.Round(totalCost, 2);
                strPrice = "$" + totalCost.ToString();

                txtbxDescription.Text = lstOfProducts[4].strDescr.
                    ToString();
                txtbxSKU.Text = lstOfProducts[4].strSKU.ToString();
                txtbxPrice.Text = strPrice;
                txtbxQuantity.Text = lstOfProducts[4].intQuantity.
                    ToString();
            }
        }
    }
}
```

Figure 5 **Source Code for ProductInfo.cs**

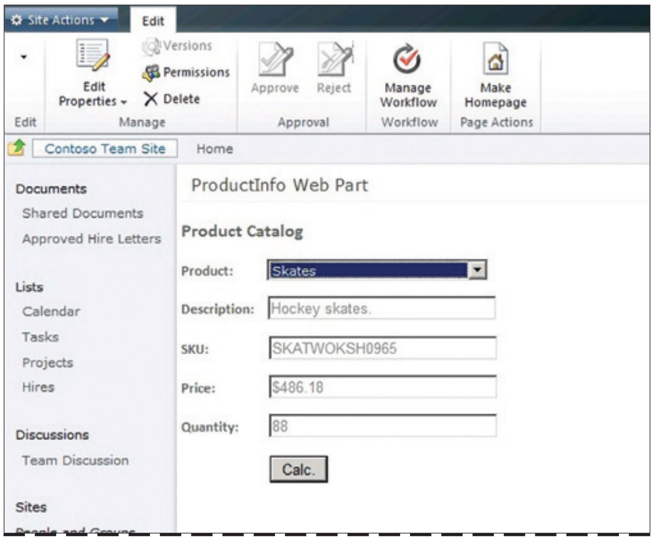
```
using System;
using System.ComponentModel;
using System.Runtime.InteropServices;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;

namespace SampleWebPartProject.ProductInfo
{
    public class ProductInfo : WebPart
    {
        private const string _ascxPath =
                  @"~/CONTROLTEMPLATES/SampleWebPartProject/ProductInfo/" +
                  @"ProductInfoUserControl.ascx";

        public ProductInfo()
        {
        }

        protected override void CreateChildControls()
        {
                Control control = this.Page.LoadControl(_ascxPath);
                Controls.Add(control);
                base.CreateChildControls();

        }

        protected override void Render(HtmlTextWriter writer)
        {
            base.RenderContents(writer);
        }
    }
}
```

Figure 6 **ProductInfo.webpart XML File**

```
<?xml version="1.0" encoding="utf-8"?>
<webParts>
  <webPart xmlns="http://schemas.microsoft.com/WebPart/v3">
    <metaData>
      <type name="SampleWebPartProject.ProductInfo.ProductInfo,
       SampleWebPartProject, Version=1.0.0.0, Culture=neutral,
       PublicKeyToken=db3a9f914308c42a" />
      <importErrorMessage>
        $Resources:core,ImportErrorMessage;
      </importErrorMessage>
    </metaData>
    <data>
      <properties>
        <property name="Title" type="string">
          Product Info Web Part</property>
        <property name="Description" type="string">Provides some
             information about hockey products.</property>
      </properties>
    </data>
  </webPart>
</webParts>
```

by using the Add method. This allows the Web Part to surface the ASP user control within the Web Part in SharePoint. **Figure 5** shows the code.

Now that you've built the visual Web Part, you can deploy it to your SharePoint server. When you created the project, you configured it to be associated with a particular server instance. The implication here is that some programmatic stitch work exists that brings together the code you've just written with the SharePoint server. If you review the files in the Solution Explorer, you'll see a



ProductInfo Web Part

Product Catalog

Product: Skates

Description: Hockey skates.

SKU: SKATWOKSH0965

Price: $486.18

Quantity: 88

Figure 7 **Web Part on Web Part Page**

number of XML files that aid in this integration. For example, the Feature.xml file (see the following code) provides a definition of the feature. You can see in the XML that the file references a couple of other XML files that also provide specific information about the Web Part. Here you can see that Elements.xml and ProductInfo.webpart are referenced:

```
<?xml version="1.0" encoding="utf-8"?>
<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
Id="416172c1-cfa7-4d7a-93ba-fe093b037fab" ImageUrl="" Scope="Site"
Title="SampleWebPartProject Feature1">
  <ElementManifests>
    <ElementManifest Location="ProductInfo\Elements.xml" />
    <ElementFile Location="ProductInfo\ProductInfo.webpart" />
  </ElementManifests>
```

Elements.xml provides information about the core assemblies that are included in the feature, and ProductInfo.webpart defines metadata about the Web Part, such as its title and description. For example, **Figure 6** shows the default Title and Description properties. You can update these properties to ensure that the Web Part metadata exposed in the Web Part Gallery is intuitive and meaningful. In the case of this Web Part, you'd likely want to amend the title to be Product Information Web Part and the description to read something like, "Web Part that provides calculated product pricing and information."

There are other XML configuration files, and if you're new to SharePoint development, I would encourage you to review each one in the project to better understand their purpose. Now let's move on to deploying the Web Part to your SharePoint server.

Deploying the Visual Web Part Project

Prior to SharePoint 2010, Stsadm, a command-line-driven administration tool, was commonly used to deploy applications to SharePoint. This need goes away with Visual Studio 2010 (and with the introduction of Window PowerShell, but this is a topic worthy of its own article). Because your project already has a relationship with your SharePoint server, and the association has a defined level of trust, you need only to right-click the project and select Build, make sure the solution builds, and then right-click and select Deploy. Of course, using the F5 key will also work when debugging your SharePoint solutions. By doing this, the debug experience includes steps such as attaching to the appropriate process and resetting IIS.

Visual Studio 2010 provides an amazing opportunity to engage in SharePoint development.

Once you've successfully deployed the Web Part, you need to open your SharePoint site and create a new Web Part page. If you clicked F5 to debug your application, the Create Web Part page is invoked by default. Otherwise, click View All Site Content, and then click Create. Click the Web Part Page option, and then provide the information requested about that particular Web Part page. For example, provide a name and layout template for the page. After you've entered this information, click Create, and SharePoint creates your Web Part page.

Now you need to add the visual Web Part you created and deployed to the server. To do this, navigate to the Web Part page, click Site Actions, and then click Edit Page. Click the Web Part zone in which you want to place the visual Web Part, click the Insert tab, and then click Web Part on the Insert tab.

After you've done this, SharePoint exposes a number of Web Part categories that you can browse to select a specific Web Part to add to the Web Part zone you selected on the page. Navigate to the Custom category, and in the Web Parts pane, you'll see the visual Web Part you created and deployed. If you followed along with the code in this article, click the ProductInfo Web Part, and then click the Add button.

The Web Part is now added to the Web Part zone on the Web Part page, shown in **Figure 7**. At this point, you can configure the Web Part options through the Tools pane, or you can simply accept the default options and click Stop Editing.

Engaging SharePoint Development

For SharePoint developers, Visual Studio 2010 provides not only a suite of native tools, but also an amazing opportunity to engage in SharePoint development. I would encourage you to check out these tools. There are some great options for developers who like to have control over their code and for those who like the design experience to build and deploy great solutions in SharePoint. ■

STEVE FOX *is a senior technical evangelist with the Developer and Platform Evangelism team at Microsoft. He spends most of his time working with customers on Office and SharePoint development. Fox has published a number of books and articles and regularly speaks at developer conferences around the world.*

Generating Documents from SharePoint with Open XML Content Controls

Eric White

It's often the case that a department manager needs to regularly send a nicely formatted status report to her general manager or that a team leader needs to send a weekly status report to a number of interested parties. To collaborate with others in their organizations, both the manager and the team leader can maintain status information in SharePoint lists. The question for developers is how to include the information in the lists in a document such as a status report.

Open XML, the default file format for Office 2007, is an ISO standard (documented in exacting detail in IS29500). Put simply, Open XML files are Zip files that contain XML, and it is very easy to generate or modify Open XML documents programmatically. All you need is a library to open a Zip file and an XML API. Using the programmability features of Open XML and SharePoint, you can put together a small document-generation system that takes advantage of Open XML content controls and puts people like the manager and team leader in charge of their reports. In this article, I'll present some guidance and sample code for creating a document-generation system that uses SharePoint lists to populate tables in an Open XML word-processing document.

This article discusses:

- Document generation
- Mining SharePoint lists
- Open XML content controls

Technologies discussed:

Open XML, LINQ to XML, Visual Studio 2008, Windows SharePoint Services

Code Download URL:

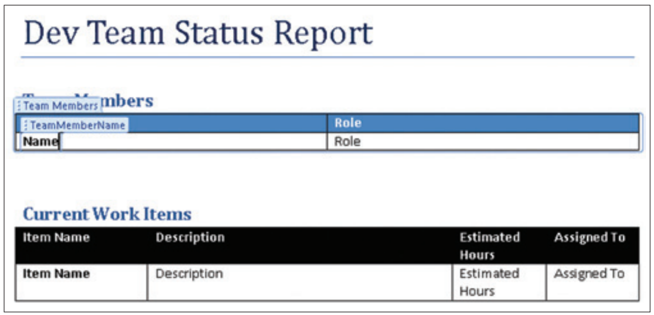
code.msdn.microsoft.com/mag200910SharePoint

Overview of the Example

The example in this article is a small SharePoint Web Part that generates an Open XML word-processing document from data that lives in SharePoint lists. I've created two custom SharePoint lists, shown in **Figure 1**, that contain the data we want to insert into the tables.

I've also created a template Open XML word-processing document that contains content controls defining the lists and columns that are the source of the data for the generated document. The controls are shown in **Figure 2**.

Finally, I created a Web Part that retrieves the list of template documents from a specific document library and presents the list to users. A user selects an item in the list and then clicks the Generate Report button. The Web Part creates an Open XML word-processing document, places it in the Reports document library, and redirects the user to that library so that she can open the report. The Web Part is shown in **Figure 3**, and the document it generates is shown in **Figure 4**.


Dev Team Status Report

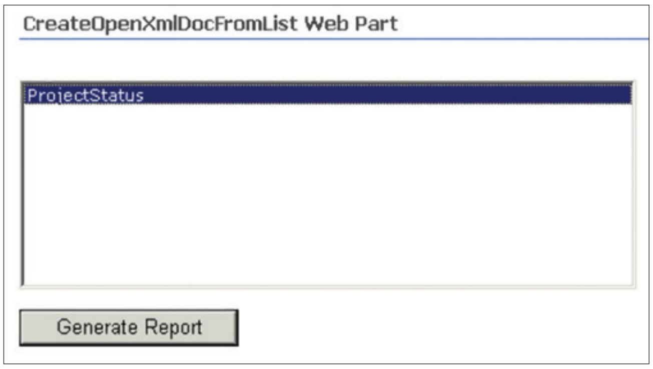
Team Members

TeamMemberName	Role
Name	Role

Current Work Items

Item Name	Description	Estimated Hours	Assigned To
Item Name	Description	Estimated Hours	Assigned To

Figure 2 **The Template Open XML Document That Contains Content Controls**


CreateOpenXmlDocFromList Web Part

ProjectStatus

Figure 3 **A SharePoint Web Part That Allows Users to Select a Template Document**

Open XML Content Controls

Before describing the SharePoint solution, I'll cover the basics of Open XML content controls. Open XML content controls provide a facility in word-processing documents that allows you to delineate content and associate metadata with that content. To use content controls, you must enable the Developer tab in Microsoft Office Word 2007. (Click Word Options on the Office menu; then, in the Word Options dialog box, select Show Developer Tab in Ribbon.)

To insert a content control, select some text and then click the button in the Controls area of the Developer tab that creates a plain text content control, shown in **Figure 5**.

You can set properties for a content control to give it a title and assign it a tag. Click in the content control, and then click the Properties button in the Controls area on the Developer tab. This displays a dialog box you can use to set the title and the tag.

Content controls use the w:sdt element in Open XML markup, which is shown in **Figure 6**. The contents of the content control are defined in the w:sdtContent element. In the figure, you can also see the title of the content control in the w:alias element and the content control tag in the w:tag element.

Programming for Open XML Using the .NET Framework

You can take a variety of approaches to programming for Open XML using the Microsoft .NET Framework:

- Use the classes in System.IO.Packaging
- Use the Open XML SDK with any of the XML programming technologies available in .NET, including XmlDocument, XmlParser, or LINQ to XML. My favorite is LINQ to XML.
- Use the strongly typed object model of the Open XML SDK version 2.0. You can find a number of articles that introduce how to program with this object model.

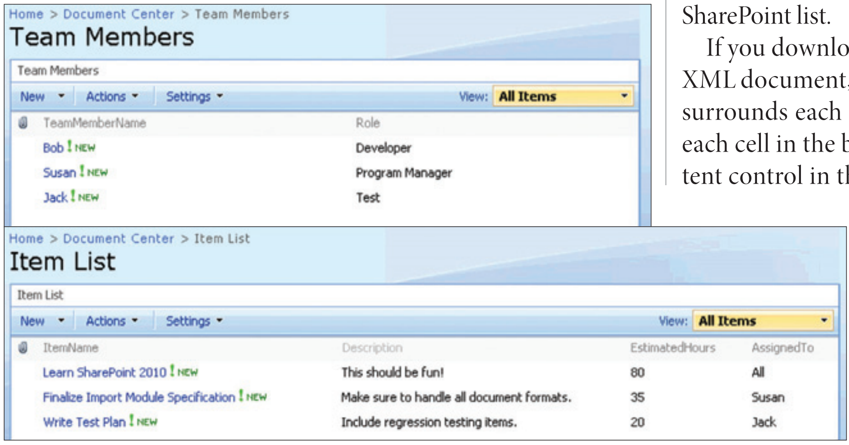
Here, I'm going to use the Open XML SDK (either version 1.0 or 2.0) with LINQ to XML. You can download the Open XML SDK from go.microsoft.com/fwlink/?LinkId=127912.

It is useful to encapsulate some Open XML functionality around content controls in a ContentControlManager class. When you approach the problem in this way, you can develop your Open XML functionality in a simple console application. After you have coded and debugged your Open XML functionality, you can then incorporate it in your SharePoint feature with a minimum of effort. It's pretty time-consuming to incur the overhead of deploying the SharePoint feature while debugging Open XML code.

For our SharePoint document-generation example, we want to write some code that retrieves a template document from a specific document library, queries the document for the content controls it contains and uses the metadata stored in each content control to populate the document with data from the appropriate SharePoint list.

If you download the sample and examine the template Open XML document, you'll see that it contains a content control that surrounds each table and that content controls are inserted in each cell in the bottom row of each table. The tag for each content control in the table cells specifies the name of a column in the SharePoint lists. For convenience, I've also set the title of each content control to the same value as the tag. Content controls display their title when the insertion point is inside the control.

When you write the code for a SharePoint feature that generates an Open XML document, the code should first query the document for these content controls. The query


Home > Document Center > Team Members

Team Members

Team Members

New ▾ Actions ▾ Settings ▾ View: All Items

TeamMemberName	Role
Bob ! NEW	Developer
Susan ! NEW	Program Manager
Jack ! NEW	Test

Home > Document Center > Item List

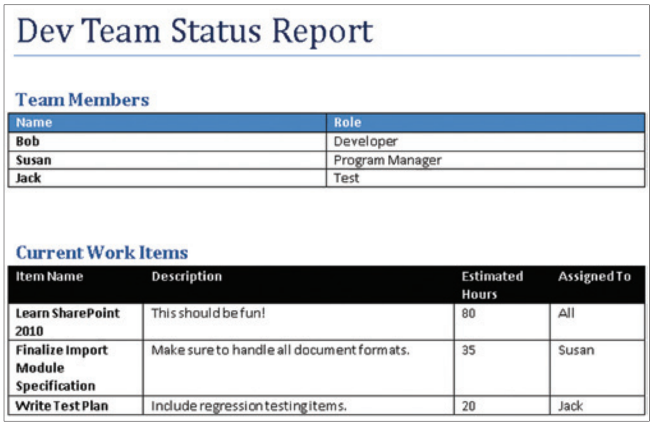
Item List

Item List

New ▾ Actions ▾ Settings ▾ View: All Items

ItemName	Description	EstimatedHours	AssignedTo
Learn SharePoint 2010 ! NEW	This should be fun!	80	All
Finalize Import Module Specification ! NEW	Make sure to handle all document formats.	35	Susan
Write Test Plan ! NEW	Include regression testing items.	20	Jack

Figure 1 **Two Custom SharePoint Lists**



Dev Team Status Report

Team Members

Name	Role
Bob	Developer
Susan	Program Manager
Jack	Test

Current Work Items

Item Name	Description	Estimated Hours	Assigned To
Learn SharePoint 2010	This should be fun!	80	All
Finalize Import Module Specification	Make sure to handle all document formats.	35	Susan
Write Test Plan	Include regression testing items.	20	Jack

Figure 4 **An Open XML Word-Processing Document That Contains the Generated Report**

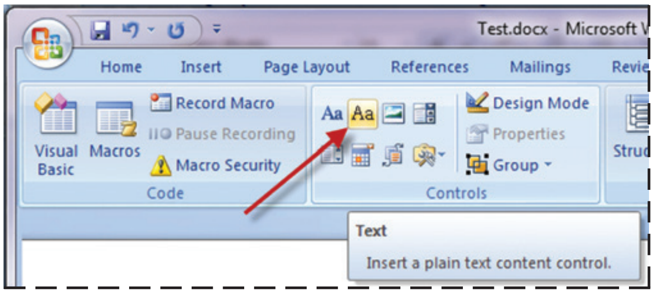


Figure 5 **Use This Button to Create a Plain Text Content Control**

returns an XML tree that describes the structure of the content controls, along with the tag for each. If you run this code on the sample document, it produces the following XML:

```
<ContentControls>
  <Table Name="Team Members">
    <Field Name="TeamMemberName" />
    <Field Name="Role" />
  </Table>
  <Table Name="Item List">
    <Field Name="ItemName" />
    <Field Name="Description" />
    <Field Name="EstimatedHours" />
    <Field Name="AssignedTo" />
  </Table>
</ContentControls>
```

This XML document shows which SharePoint lists our code needs to query. For each item in the list, you need to retrieve the values of the specified columns. The code to query the Open XML word-processing document, shown in **Figure 7**, is written as a LINQ to XML query that uses functional construction to form the returned XML.

To use functional construction, the code instantiates an XElement object using its constructor, passing a LINQ to XML query as an argument to the constructor. The LINQ to XML query uses axis methods to retrieve the appropriate elements in the body of the document and uses the Enumerable.Select extension method to form new XML from the results of the query. Functional construction takes a bit of study to understand, but as you can see, once you wrap your head around it, you can do an awful lot with just a little bit of code.

Figure 6 **Open XML Markup for a Content Control**

```
<w:p>
  <w:r>
    <w:t xml:space="preserve">Not in content control.  </w:t>
  </w:r>
  <w:sdt>
    <w:sdtPr>
      <w:alias w:val="Test"/>
      <w:tag w:val="Test"/>
      <w:id w:val="5118254"/>
      <w:placeholder>
        <w:docPart w:val="DefaultPlaceholder_22675703"/>
      </w:placeholder>
    </w:sdtPr>
    <w:sdtContent>
      <w:r>
        <w:t>This is text in content control.</w:t>
      </w:r>
    </w:sdtContent>
  </w:sdt>
  <w:r>
    <w:t xml:space="preserve">  Not in content control.</w:t>
  </w:r>
</w:p>
```

Preatomization of XName and XNamespace Objects

The code in **Figure 7** uses an approach called "preatomization" of the LINQ to XML names. This is just a fancy way of saying that you write a static class (see **Figure 8**) that contains static fields that are initialized to the qualified names of the elements and attributes you are using.

There is a good reason to initialize XName and XNamespace objects in this fashion. LINQ to XML abstracts XML names and namespaces into two classes: System.Xml.Linq.XName and System.Xml.Linq.XNamespace, respectively. The semantics of these

Figure 7 **Retrieving the Structure of the Content Controls in the Template Document**

```
public static XElement GetContentControls(
  WordprocessingDocument document)
{
    XElement contentControls = new XElement("ContentControls",
        document
            .MainDocumentPart
            .GetXDocument()
            .Root
            .Element(W.body)
            .Elements(W.sdt)
            .Select(tableContentControl =>
                new XElement("Table",
                    new XAttribute("Name", (string)tableContentControl
                        .Element(W.sdtPr).Element(W.tag).Attribute(
                          W.val)),
                    tableContentControl
                        .Descendants(W.sdt)
                        .Select(fieldContentControl =>
                            new XElement("Field",
                                new XAttribute("Name",
                                    (string)fieldContentControl
                                        .Element(W.sdtPr)
                                        .Element(W.tag)
                                        .Attribute(W.val)
                                    )
                                )
                            )
                    )
                )
            );
    return contentControls;
}
```

Figure 8 **A Static Class Containing Static Fields to Preatomize XName and XNamespace Objects**

```
public static class W
{
    public static XNamespace w =
        "http://schemas.openxmlformats.org/wordprocessingml/2006/main";

    public static XName body = w + "body";
    public static XName sdt = w + "sdt";
    public static XName sdtPr = w + "sdtPr";
    public static XName tag = w + "tag";
    public static XName val = w + "val";
    public static XName sdtContent = w + "sdtContent";
    public static XName tbl = w + "tbl";
    public static XName tr = w + "tr";
    public static XName tc = w + "tc";
    public static XName p = w + "p";
    public static XName r = w + "r";
    public static XName t = w + "t";
    public static XName rPr = w + "rPr";
    public static XName highlight = w + "highlight";
    public static XName pPr = w + "pPr";
    public static XName color = w + "color";
    public static XName sz = w + "sz";
    public static XName szCs = w + "szCs";
}
```

classes include the notion that if two XNames have the same qualified name (namespace + local name), they will be represented by the same object. This allows for fast comparison of XName objects. Instead of using a string comparison to select XElement objects of a given name, the code needs only to compare objects. When you initialize an XName object, LINQ to XML first looks in a cache to determine whether an XName object with the same namespace and name already exists. If one does, the object is initialized to the existing XName object from the cache. If one doesn't exist, LINQ to XML initializes a new one and adds it to the cache. As you might imagine, if this process is repeated over and over, it can result in performance issues. By initializing these objects in a static class, the work is done only once. In addition, by using this technique you reduce the possibility that an element or attribute name is misspelled in the body of the code. One other advantage is that by using this technique, you get support from IntelliSense, which makes writing Open XML programs using LINQ to XML easier.

Figure 9 **Extension Methods Use Open XML SDK Annotations to Minimize Deserialization of XML**

```
public static class AssembleDocumentLocalExtensions
{
    public static XDocument GetXDocument(this OpenXmlPart part)
    {
        XDocument xdoc = part.Annotation<XDocument>();
        if (xdoc != null)
            return xdoc;
        using (Stream str = part.GetStream())
        using (StreamReader streamReader = new StreamReader(str))
        using (XmlReader xr = XmlReader.Create(streamReader))
            xdoc = XDocument.Load(xr);
        part.AddAnnotation(xdoc);
        return xdoc;
    }

    public static void PutXDocument(this OpenXmlPart part)
    {
        XDocument xdoc = part.GetXDocument();
        if (xdoc != null)
        {
            // Serialize the XDocument object back to the package.
            using (XmlWriter xw =
                XmlWriter.Create(part.GetStream
               (FileMode.Create, FileAccess.Write)))
            {
                xdoc.Save(xw);
            }
        }
    }
}
```

The GetXDocument and PutXDocument Extension Methods

The example presented in this article also uses another little trick to ease programming and improve performance. The Open XML SDK has the ability to put annotations on parts in the document. This means that you can attach any .NET Framework object to an OpenXmlPart object and later retrieve it by specifying the type of the object that you attached.

We can define two extension methods, GetXDocument and PutXDocument, that use annotations to minimize deserialization of the XML from the Open XML part. When we call GetXDocument, it first looks to see whether an annotation of type XDocument exists on the OpenXmlPart. If the annotation exists, GetXDocument returns it. If the annotation doesn't exist, the method populates an XDocument from the part, annotates the part, and then returns the new XDocument.

The PutXDocument extension method also checks whether there is an annotation of type XDocument. If this annotation exists, PutXDocument writes the XDocument (presumably modified after the code calls GetXDocument) back to the OpenXMLPart. The GetXDocument and PutXDocument extension methods are shown in **Figure 9**. You can see the use of the GetXDocument extension method in the GetContentControls method listed earlier in **Figure 7**.

Replacing the Content Controls with Data

Now that we have a method that returns the structure of the content controls in the tables and cells, we need a method (SetContentControls) that creates an Open XML document with specific data (retrieved from the SharePoint lists) inserted in the tables. We can define this method to take an XML tree as an argument. The XML tree is shown in **Figure 10**, and **Figure 11** shows the document that SetContentControls creates when it is passed the XML tree.

You can see that the single row that contained content controls has been replaced by multiple rows, each containing the data from the XML tree that was passed as an argument to the method. By using an XML tree to pass the data to the code that manipulates the Open XML markup, you achieve a nice separation of the code that uses the SharePoint object model and the Open XML code.

The code to assemble the new document honors any formatting you've applied to the table. For example, if you've configured the table to show different colors for alternating rows, or if you've set the background color of a column, the newly generated document reflects your formatting changes.

Figure 10 **An XML Tree That Contains Data to Insert in the Word-Processing Document Tables**

```
<ContentControls>
  <Table Name="Team Members">
    <Field Name="TeamMemberName" />
    <Field Name="Role" />
    <Row>
      <Field Name="TeamMemberName" Value="Bob" />
      <Field Name="Role" Value="Developer" />
    </Row>
    <Row>
      <Field Name="TeamMemberName" Value="Susan" />
      <Field Name="Role" Value="Program Manager" />
    </Row>
    <Row>
      <Field Name="TeamMemberName" Value="Jack" />
      <Field Name="Role" Value="Test" />
    </Row>
  </Table>
  <Table Name="Item List">
    <Field Name="ItemName" />
    <Field Name="Description" />
    <Field Name="EstimatedHours" />
    <Field Name="AssignedTo" />
    <Row>
      <Field Name="ItemName" Value="Learn SharePoint 2010" />
      <Field Name="Description" Value="This should be fun!" />
      <Field Name="EstimatedHours" Value="80" />
      <Field Name="AssignedTo" Value="All" />
    </Row>
    <Row>
      <Field Name="ItemName" Value=
        "Finalize Import Module Specification" />
      <Field Name="Description" Value="Make sure to handle all document
               formats." />
      <Field Name="EstimatedHours" Value="35" />
      <Field Name="AssignedTo" Value="Susan" />
    </Row>
    <Row>
      <Field Name="ItemName" Value="Write Test Plan" />
      <Field Name="Description" Value=
        "Include regression testing items." />
      <Field Name="EstimatedHours" Value="20" />
      <Field Name="AssignedTo" Value="Jack" />
    </Row>
  </Table>
</ContentControls>
```

If you download and examine the ContentControlManager example, you can see that the code gets a copy of the row that contains content controls and saves it as a prototype row:

```
// Determine the element for the row that contains the content controls.
// This is the prototype for the rows that the code will generate from data.
XElement prototypeRow = tableContentControl
    .Descendants(W.sdt)
    .Ancestors(W.tr)
    .FirstOrDefault();
```

Then, for each item retrieved from the SharePoint list, the code clones the prototype row, alters the cloned row with the data from the SharePoint list, and adds it to a collection that is inserted into the document.

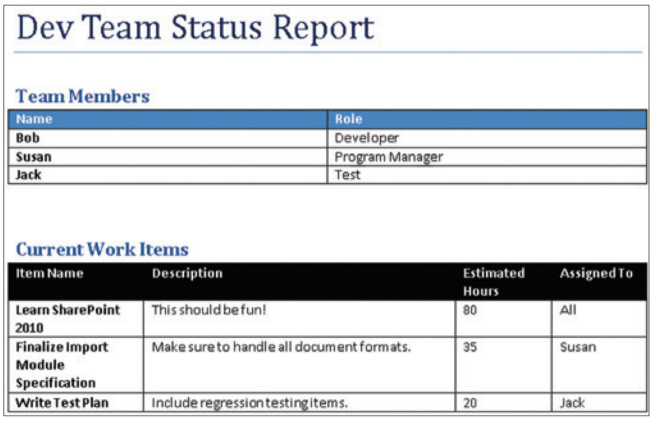
After creating the list of new rows, the code removes the prototype row from the list and inserts the collection of newly created rows, as shown here:

```
XElement tableElement = prototypeRow.Ancestors(W.tbl).First();
prototypeRow.Remove();
tableElement.Add(newRows);
```

Creating the SharePoint Feature

I used the February 2009 CTP release of the Visual Studio 2008 extensions for Windows SharePoint Services 3.0, v1.3 (microsoft.com/downloads/details.aspx?FamilyID=B2C0B628-5CAB-48C1-8CAE-C34C1CCBDC0A&displaylang=en) to build this example. I've built and run this example on both 32-bit and 64-bit versions of WSS. (Kirk Evans has some great webcasts that show how to use these extensions at blogs.msdn.com/kaevans/archive/2009/03/13/sharepoint-developer-series-part-1-introducing-vsewss-1-3.aspx.)

The example contains the code to create the Web Part controls. The code is pretty self-explanatory if you are accustomed to building SharePoint Web Parts. When a user clicks the Generate Report button, the code calls the CreateReport method, which assembles the new Open XML word-processing document from the template document by using the data in the SharePoint lists as configured in the tags of the content controls.There are a few points to note about the code for the CreateReport method. Files in a document library in SharePoint are returned as byte arrays. You need to con-



Dev Team Status Report

Team Members

Name	Role
Bob	Developer
Susan	Program Manager
Jack	Test

Current Work Items

Item Name	Description	Estimated Hours	Assigned To
Learn SharePoint 2010	This should be fun!	80	All
Finalize Import Module Specification	Make sure to handle all document formats.	35	Susan
Write Test Plan	Include regression testing items.	20	Jack

Figure 11 **The Generated Document**

Figure 12 **Writing a Byte Array from SharePoint to a MemoryStream**

```
private ModifyDocumentResults CreateReport(SPFile file, Label message)
{
    byte[] byteArray = file.OpenBinary();
    using (MemoryStream mem = new MemoryStream())
    {
        mem.Write(byteArray, 0, (int)byteArray.Length);
        try
        {
            using (WordprocessingDocument wordDoc =
                WordprocessingDocument.Open(mem, true))
            {
                // Get the content control structure from the template
                // document.
                XElement contentControlStructure =
                    ContentControlManager.GetContentControls(wordDoc);

                // Retrive data from SharePoint,
                   constructing the XML tree to
                // pass to the ContentControlManager.SetContentControls
                // method.
                ...
            }
        }
    }
}
```

SharePoint is a powerful technology that makes it easy for people in organizations to collaborate.

vert this byte array into a memory stream so that you can open and modify the document using the Open XML SDK. One of the MemoryStream constructors takes a byte array, and you might be tempted to use that constructor. However, the memory stream created with that constructor is a nonresizable memory stream, and the Open XML SDK requires that the memory stream be resizable. The solution is to create a MemoryStream with the default constructor and then write the byte array from SharePoint into the MemoryStream, as shown in **Figure 12**.

The remainder of the code is straightforward. It uses the SharePoint object model to retrieve document libraries and the contents of the libraries, retrieve lists and retrieve values of columns for each row in the list. It assembles the XML tree to pass to ContentControlManager.SetContentControls, and then it calls SetContentControls.

The code assembles the name of the generated report document as Report-yyyy-mm-dd. If the report already exists, the code appends a number to the report name to disambiguate the report from other reports that have already been generated. For instance, if Report-2009-08-01.docx already exists, the report is written to Report-2009-8-2 (1).docx.

Easy Customizations

You will probably want to customize this example to suit your own needs. One possible enhancement is to allow for a content control in the body of the template document that pulls boilerplate content from a specified document stored in SharePoint. You could write the code so that you could place the name of the document that contains the boilerplate text as text in the content control.

Also, this example hard-codes the names of the TemplateReports and the Reports document libraries. You could remove this constraint by specifying this information in a SharePoint list. The code would then know about the name of only this configuration list. The names of the TemplateReports and the Reports document libraries would be driven by data from your configuration list.

SharePoint is a powerful technology that makes it easy for people in organizations to collaborate. Open XML is a powerful emerging technology that is changing the way that we generate documents. Using the two technologies together enables you to build applications in which people can use documents to collaborate in new ways. ■

ERIC WHITE *is a writer at Microsoft specializing in the Office Open XML file formats, Office and SharePoint. Before joining Microsoft in 2005, he worked as a developer for a number of years and then started PowerVista Software, a company that developed and sold a cross-platform grid widget. He has written books on custom control and GDI+ development. Read his blog at blogs.msdn.com/ericwhite.*

Core Instrumentation Events in Windows 7, Part 2

Dr. Insung Park and Alex Bendetov

Welcome back for the second part of our two-part article series: Core Instrumentation Events in Windows 7. In the first article, we presented a high-level overview of the Event Tracing for Windows (ETW) technology and core OS instrumentation. We also discussed tool support to obtain and consume OS events. In this second part, we continue to provide more details about the events from various subcomponents in the core OS. We also explain how the different system events can be combined to produce a comprehensive picture of system behavior, which we demonstrate by using a set of Windows PowerShell scripts.

Disk, File, File Details and Driver Events

From a program's perspective, operations such as opening, reading, or writing files are the way to access the contents on the disk. Due to optimizations such as caching and prefetching, not all file IO requests result in immediate disk access. Furthermore, file contents may be scattered across disks, and certain disk devices support mirroring and striping, and so on. For such cases, reading one block of data from a file translates into multiple accesses to one or more disks. The events for file and disk access account for file IO start, file IO completion, disk access start, disk access end, split IO, driver activities and file (name to unique key) maps.

Disclaimer: This article is based on a prerelease version of Windows 7. Details are subject to change.

This article uses the following technologies:

Windows 7

This article discusses:

- Disk, File, File Details and Driver Events
- Network Events
- Ready Thread Events
- Simple Core OS Event Analysis Samples

A request from a user application to access a file and the corresponding completion of that request back to the user application travels through a stack of multiple components. In the Windows IO system, IO operations are tracked by an entity called an IO Request Packet (IRP). A user-initiated IO operation is turned into an IRP when it enters the IO Manager. As an IRP traverses a chain of components, each component performs necessary tasks to process the request, updates the IRP and passes it on, if necessary, to the component that will handle the request next. When all requirements of the IO request are satisfied (in a simple case, a requested block of a file is retrieved from a disk), registered completion routines are called to perform any additional processing of the data, and the requested data is returned to the user application.

At a higher layer in the core IO system, File IO events record the operations issued by an application. File IO events include the following types: Create, Read, Write, Flush, Rename, Delete, Cleanup, Close, Set Information, Query Information, Directory Enumeration, and Directory Change Notification. Operations such as Create, Read, Write, Flush, Rename and Delete are straightforward, and they contain data items such as file key, IO request packet (IRP) pointer, block size, and offset into the file, as necessary. Set Information and Query Information events indicate that file attributes were set or queried. A Cleanup event is logged when the last handle to the file is closed. A Close event specifies that a file object is being freed. Directory Enumeration and Directory Change Notification events are logged when a directory is enumerated or a directory change notification is sent out to registered listeners, respectively. File IO events are logged to ETW when the operation is requested. Those that are interested in the completion and duration of the file IO operations can enable File IO Completion events, which can be correlated to the original File IO events through IRP pointer. File IO Completion events record IRP pointer and return status.

Disk events are logged at a lower level in the IO stack, and they contain disk-access-specific information. Read and Write operations generate Disk Read and Write events containing disk number, transfer size, byte offset to the address being accessed, IRP pointer, and response time of the access. Flush events record disk flush operations. Unlike File IO events that are logged at the beginning of operations, Disk IO events are logged at the IO completion time. Users have the option to collect additional Disk IO Init events for all Disk IO events (ReadInit, WriteInit and FlushInit events). As mentioned earlier, not all File IO events have matching Disk IO events, if for instance the requested content is already available in the cache or a write to disk operation is buffered. Split IO events indicate that IO requests have been split into multiple disk IO requests due to the underlying mirroring disk hardware. Users without such hardware will not see Split IO events even if they enable them. It maps the original parent IRP into multiple child IRPs.

Disk IO, File IO and Split IO events contain unique file keys created for open files. This file key can be used to track related IO operations within the IO system. However, the actual file name for the file operation is not available in any File or Disk IO events. To resolve the name of the files, File Details events are needed. All open files are enumerated to record their file keys and names. In a simulated state machine, file objects are tracked in terms of file keys, to record file IO requests and actual disk accesses, and then names are updated in the objects when File Details events are encountered. For a historical reason, File Keys in Disk IO and File Details events are named FileObject. Most File IO events contain both file object and file key.

Driver events indicate activities in drivers, which, depending on the device type, may or may not overlap with disk IO activities. Driver events may be of interest to users familiar with the Windows Driver Model (WDM). The driver instrumentation adds events around driver IO function calls and completion routines. Driver events contain driver data such as file key, IRP pointer, and routine addresses (major and minor function and completion routine), as appropriate for individual event types.

IO events usually result in a very large volume of events, which may require increasing the number and/or size of the buffers for the kernel session (-nb option in logman). Also, IO events are useful in analyzing file usages, disk access patterns and driver activities. However, the process and thread id values of the IO events, with the exception of Disk IO events, are not valid. To correlate these activities correctly to the originating thread and thus to the process, one needs to consider tracking Context Switch events.

Network Events

Network events are logged when network activities occur. Network events from the kernel session are emitted at TCP/IP and UDP/IP layers. TCP/IP events are logged when the following actions take place: Send, Receive, Disconnect, Retransmit, Reconnect, Copy, and Fail. They all contain the packet size, source and destination IP addresses and ports, except for Fail events since no such information is available. Also, the originating process id for Send type events and the target process id for Receive type events are included because often these network operations are not performed under the originating/receiving process context. This means that Network events can be attributed to a process, but not to a thread. UDP/IP events are either Send or Receive events, which include the data items listed above. Finally, each operation type (Send, Receive and so on) corresponds to a pair of events: one for IPV4 protocol and the other for IPV6. Events for IPV6 protocol were added in Windows Vista.

Registry Events

Most registry operations are instrumented with ETW, and these events can be attributed to processes and threads through process and thread ids in the event header. The Registry event payload does not contain a full registry key name. Instead, it includes a unique key, noted as Key Control Block (KCB), for each open registry key. At the end of a kernel session, rundown events are logged that map these keys to full registry names. To resolve registry names, one needs to use a similar technique utilized for file name resolution,

Figure 1 **Get-ETW-PID-List Script for Printing Process Table**

```
<######################
 # Get-ETW-PID-List
 #
 # @brief Takes in an XML stream and prints the list of process names
 # with PIDs that were detected in the incoming stream.
 ######################>
function Get-ETW-PID-List([Switch] $print) {
    begin {
        $pidlist = new-object System.Collections.Hashtable
        $procid = 0
        $procname = ""
    }

    process {
        $xmldata = $_

        # For each Process event, grab process id and binary name
        # and add to the list.
        foreach ($e in $xmldata.Events.Event) {
            if ($e.RenderingInfo.EventName.InnerText -eq "Process") {
                foreach ($d in $e.EventData.Data) {
                    # Grab process id.
                    if ($d.Name -eq "ProcessId") {
                       $procid = $d.InnerText
                    # Grab the process name.
                    } elseif ($d.Name -eq "ImageFileName") {
                        $procname = $d.InnerText
                    }
                }
                if (!$pidlist.Contains($procid)) {
                    $pidlist.add($procid, $procname) | Out-Null
                }
            }
        }
    }

    end {
        remove-variable xmldata

        if ($print) {
            "{0,-29}| PID” -f "Binary Name"
            "-------------------------------------"
            foreach ($item in $pidlist.Keys) {
                “{0,-30}({1})” -f $pidlist.$item,$item
            }
        } else {
            return $pidlist
        }
    }
}
```

Figure 2 **Output of Get-ETW-PID-List Script**

```
PS C:\tests> $xmldata | Get-ETW-PID-List -print
Binary Name                   | PID
-------------------------------------
dwm.exe                       (2776)
powershell.exe                (2384)
svchost.exe                   (708)
notepad.exe                   (4052)
iexplore.exe                  (4284)
...
iexplore.exe                  (4072)
svchost.exe                   (3832)
smss.exe                      (244)
System                        (4)
spoolsv.exe                   (1436)
Idle                          (0)
```

in which these map events are used to update registry objects in the state machine. Registry events have been useful in analyzing access patterns and identifying redundant accesses for optimization. The Registry event payload contains the return status of the operation, which can be used to monitor registry operation failures and troubleshoot possible application problems.

Sample-Based Profile Events

Sample-Based Profile events (Profile events, hereafter) allow tracking where CPUs spend their time. Profile events can be used to compute a good approximation of CPU utilization. When Profile events are enabled, the Windows kernel turns on profiling interrupts, which causes Profile events to be logged from each processor at a fixed rate (the default is 1000 times a second on each processor). It should be noted that in low-power mode on some machines, Profile events may be turned off temporarily. The Profile event payload consists of the thread id of the thread currently running on that processor and the value of the instruction pointer register at the time of the profiling interrupt. In Windows 7, the Profile event payload also contains flags needed to identify execution context (thread/DPC/ISR).

CPU utilization can be approximated with Profile events, which is the percentage of Profile events with thread ids other than that of the idle thread. CPU usage distribution per process requires one more step of tracking thread ids in the Profile event payloads to individual processes. If a state machine is available with Process and Thread events as discussed in Part 1 of this two-part article series, it is straightforward to generate a per-process CPU usage report. It is also possible to trace CPU usage to a loaded module through Image Load events. Comparing the instruction pointer with the address range of loaded modules results in the location of instruction pointers to a binary image, and thus to a profile of CPU usage per module. If binary symbols are available, one can obtain function names from instruction pointers using the DbgHelp library. With call-stacks enabled for Profile events, one can even deduce how the function is invoked. This is very helpful when the instruction pointer points to a frequently used library function.

Ready Thread Events

There are several reasons why the system will switch out a running thread. One of the most common reasons is that it needs to wait for other threads to finish related tasks before it can continue. Such

Figure 3 **Get-ETW-PID-Info Script for Printing Process Details**

```
<####################
 # Get-ETW-PID-Info
 #
 # @brief Retrieves various information about a particular process id
 ####################>
function Get-ETW-PID-Info([Int] $p, [switch] $i, [switch] $t) {
    begin {
        $threadlist = New-Object System.Collections.ArrayList
        $imagelist = New-Object System.Collections.ArrayList
        $procname = ""
    }

    process {

        $xmldata = $_

        $sievedxml = $($xmldata | Get-ETW-PID-Events $p).$p

        foreach ($e in $sievedxml.Events.Event) {
            if ($e.RenderingInfo.EventName.InnerText -eq "Process") {
                foreach ($d in $e.EventData.Data) {
                    # Grab the process binary name
                    if ($d.Name -eq "ImageFileName") {
                        $procname = $d.InnerText
                    }
                }
            }
            if ($e.RenderingInfo.EventName.InnerText -eq "Image") {
                foreach ($d in $e.EventData.Data) {
                    # Grab the loaded image name and add it to the list
                    if ($d.Name -eq "FileName") {
                        if (!$imagelist.contains($d.InnerText)) {
                            $imagelist.add($d.InnerText) | Out-Null
                        }
                    }
                }
            }
            if ($e.RenderingInfo.EventName.InnerText -eq "Thread") {
                foreach ($d in $e.EventData.Data) {
                    # Grab thread id and add it to the list
                    if ($d.Name -eq "TThreadId") {
                        $tid = $d.InnerText
                        if (!$threadlist.contains($tid)) {
                            $threadlist.add($tid) | Out-Null
                        }
                    }
                }
            }
        }
    }

    end {
        "Process Name: $procname"
        if ($t) {
            "Thread List: ($($threadlist.Count))"
            $threadlist | Sort -descending
        } else {
            "Thread Count: $($threadlist.Count)"
        }
        if ($i) {
            "Image List ($($imagelist.Count)):"
            $imagelist | Sort
        } else {
            "Images: $($imagelist.Count)"
        }
    }
}
```

dependencies in execution surface as threads being blocked from execution while waiting for an object to be set, such as event, semaphore, timer and so on. The Windows OS scheduler (also known as dispatcher) keeps track of threads becoming unblocked by another thread, a DPC, or a timer. Dependencies between components and threads are not readily seen and predicted during development. When unforeseen delays take place, it becomes difficult to track it to the root cause of the problem.

Ready Thread (or Dispatcher) events were added to help diagnose such problems. When a thread is unblocked (or readied) and placed in the dispatcher ready queue, a Ready Thread event is logged, whose payload contains the thread id of the readied thread. By following up the chain of Ready Thread events, one can track the chain of execution dependency as it unfolds. Context Switch events, presented above, indicate when the readied thread actually is scheduled to run. Call-stacks enabled for Ready Thread events may lead to the point in the code responsible for unblocking of the waiting thread. Ready Thread events are newly available in Windows 7.

System Call Events

System Call events denote entries into and exits out of Windows core OS system calls. System calls are the interface into the Windows kernel, consisting of a number of APIs. This instrumentation was added to monitor system calls made by user mode applications and kernel mode components. There are two types of System Call events. A System Call Enter event indicates an invocation of a system call and logs the address of the routine corresponding to the invoked system service. A System Call Exit event indicates an exit from a system call, and its payload contains the return value from the API call. System Call events are useful for statistical analysis on system call activities and latencies, but like some of the IO and Memory events, they are logged without current process and thread id information in the header. To correlate system call activities with processes and threads, one must collect Context Switch events at the same time and, during state machine simulation, track the current thread running on the CPUs using the CPU id in the event header of the System Call events, as described in Part 1 of this two-part article series.

Advanced Local Procedure Call Events

Local Procedure Call (LPC) has been an efficient local Inter-Process Communication (IPC) mechanism on Windows platforms for years. Windows Vista provided a more efficient and secure means for IPC needs, called Advanced Local Procedure Call (ALPC). ALPC is also used as the transport mechanism for local Remote Procedure Call (RPC). The ALPC component is instrumented with ETW, emitting Send, Receive, Wait for New Reply, Wait for New Message and Unwait events.

System Configuration Events

When the kernel session ends, ETW logs several events that describe the system configuration of the machine on which the events are collected. In many performance analysis scenarios, knowing the underlying hardware and software configuration helps greatly in understanding the system behavior, especially when the machine

Figure 4 **Output from Get-ETW-PID-Info Script**

```
PS C:\tests> $xml | Get-ETW-PID-Info 4052 -i
Process Name: notepad.exe
Thread Count: 2
Image List (31):
\Device\HarddiskVolume2\Windows\System32\advapi32.dll
\Device\HarddiskVolume2\Windows\System32\dwmapi.dll
\Device\HarddiskVolume2\Windows\System32\gdi32.dll
\Device\HarddiskVolume2\Windows\System32\imm32.dll
\Device\HarddiskVolume2\Windows\System32\kernel32.dll
\Device\HarddiskVolume2\Windows\System32\KernelBase.dll
...
\Device\HarddiskVolume2\Windows\System32\version.dll
\Device\HarddiskVolume2\Windows\System32\winspool.drv
```

Figure 5 **Get-ETW-Top-VMops Script for Printing Top *n* Processes Invoking the Most VM Operations**

```
<#####################
 # Get-ETW-Top-VMops
 #
 # @brief Gets the top $num processes ranked by the number of virtual
   memory events
 #####################>
function Get-ETW-Top-VMops ([Int] $num) {
    begin {
        $hold = @{}
    }

    process {

        $xmldata = $_

        # Get a list of the PIDs
        $list = $xmldata | Get-ETW-PID-List

        # Iterate through each PID
        $eventxml = $xmldata | Get-ETW-PID-Events $list.Keys

        foreach ($pd in $list.Keys) {

            $vmops = 0

            # Count the number of VM events
            foreach ($e in $eventxml.$pd.Events.Event) {
                [String] $type = $e.RenderingInfo.EventName.InnerText
                [String] $opcode = $e.RenderingInfo.Opcode

                if ($type -eq "PageFault") {
                    if ($opcode -eq "VirtualAlloc" -or
                      $opcode -eq "VirtualFree") {
                        $ vmops++
                    }
                }
            }
            $hold.$pd = $vmops
        }
    }

    end {
        "{0,-20}|{1,-7}| VM Event Count" -f "Binar"," PID"
        "---------------------------------------------------------"

        $j = 0
        foreach ($e in ($hold.GetEnumerator() | Sort Value  -Descending)) {
            if ($j -lt $num) {
                $key = $e.Key
                "{0,-20} ({1,-6} {2}" -f $list.$key,"$($e.Key))",$e.Value
            } else {
                return
            }
            $j++
        }
    }
}
```

on which the events are analyzed is different from the machine where the events were collected. These System Configuration events provide information on CPU, graphics cards, network cards, PnP devices, IDE devices, physical and logical disk configuration, and services. The System Configuration event payload varies depending on the device or configuration that it describes but typically contains description strings and key specifications.

Applications instrumented with ETW can benefit even further from precise correlation of applicatiaon and OS activities.

Simple Core OS Event Analysis Samples

In this section, we present simple scripts to demonstrate a few basic accounting techniques on some of the OS events introduced previously. We will use Windows Powershell scripts for simplicity, but the underlying algorithms can be adopted in applications for more efficient processing. Once an XML dump is created by the tracerpt tool, one can import the events as objects in Windows Powershell using the following commands:

```
> $xmldata = New-Object System.Xml.XmlDocument
> $xmldata.Load(<xml dump file>)
```

The first Windows Powershell script in **Figure 1** simply prints all the processes in the log file by scanning for Process events. It updates a hash table with a process id and name pair and prints out the table at the end if the –print option is specified. By default, it passes along the array of pairs to a pipe so that other scripts can use its output. Please note that we skipped conventional handling of arguments, comments and error checks to simplify the sample scripts. We also assume process and thread ids are not recycled during the kernel session in this script, but they do get recycled in reality. Additional codes are needed for correct processing.

The output of this script, if the xml dump contains Process events, is as shown in **Figure 2**. The goal of the next script is to build a basic state machine and, given a process ID, to print the number of threads and a list of loaded images in that process. If the –i or –t options are given, the script prints the names of loaded images and the ids of threads in the process, instead of the counts of images and threads. **Figure 3** shows the script Get-ETW-PID-Info written for this functionality. This script calls another script called Get-ETW-PID-Events (that we do not show here) that picks up only the events relevant to the given process ID.

Looking for the notepad process from **Figure 2**, we get the following output from the Get-ETW-PID-Info script. **Figure 4** lists all the modules loaded by notepad.exe.

Finally, we are going to print a table with top n processes with the most virtual memory operations. In this script called Get-ETW-Top-VMops, shown in **Figure 5**, we run Get-ETW-PID-List to construct a list of process IDs. Then we filter events by each process ID and count VM events. Last, we sort and print the table with top n processes.

The output from this script with top 10 processes with the most virtual memory operations is given in **Figure 6**. Note that two instances of logman.exe show up in the table, one for starting and the other for stopping the kernel session.

Figure 6 **The Output From Get-ETW-Top-VMops**

```
PS C:\tests> $xml | Get-ETW-Top-Vmops 10
Binary              | PID    | VM Event Count
-------------------------------------------------------
svchost.exe          (536)   486
iexplore.exe         (3184)  333
svchost.exe          (1508)  206
logman.exe           (2836)  98
svchost.exe          (892)   37
sidebar.exe          (3836)  37
logman.exe           (1192)  19
svchost.exe          (2052)  18
audiodg.exe          (7048)  18
services.exe         (480)   13
```

Enhance Your Tool Capability

Windows 7 features hundreds of event providers from various components. In this two-part article series, we have presented the set of core OS ETW events available on Windows 7 and the analysis techniques that we have used for many years. Individual events indicate certain activities in the core OS, but if combined through context-sensitive analysis methods, they can be used to produce meaningful reports that provide insights into patterns and anomalies in resource usage. Moreover, we know of quite a few cases in which effective and careful examination of a subset of these events resulted in great findings in specialized fields of study. Applications instrumented with ETW can benefit even further from precise correlation of application and OS activities; for instance, if an application emits events indicating a beginning and an end of an important application activity, core OS events collected at the same time contains accurate OS resource usage information attributed to that activity. Management and performance tool developers can take advantage of the core system events and various analysis techniques to enhance their tool capability, which will in turn benefit IT workers. Application developers may be able to diagnose application problems better if such analysis is incorporated in their environments. We hope that our two-part article series will lead to the promotion of sound engineering practice, greater software quality and better user experiences. ■

DR. INSUNG PARK *is a development manager for the Windows Instrumentation and Diagnosis Platform Team. He has published a dozen papers on performance analysis, request tracking, instrumentation technology, and programming methodology and support. His e-mail address is insungp@microsoft.com.*

ALEX BENDETOV *is a development lead for the Windows Instrumentation and Diagnosis Platform Team. He works on both Event Tracing for Windows and the Performance Counter technologies. He can be reached at alexbe@microsoft.com.*

Functional Programming for Everyday .NET Development

What is the most important advance in the .NET ecosystem over the past three or four years? You might be tempted to name a new technology or framework, like the Windows Communication Foundation (WCF) or Windows Presentation Foundation (WPF). For me personally, however, I would say that the powerful additions to the C# and Visual Basic languages over the last two releases of the .NET Framework have had a far more significant impact on my day-to-day development activities. In this article I'd like to examine in particular how the new support for functional programming techniques in .NET 3.5 can help you do the following:

1. Make your code more declarative.
2. Reduce errors in code.
3. Write fewer lines of code for many common tasks.

The Language Integrated Query (LINQ) feature in all of its many incarnations is an obvious and powerful use of functional programming in .NET, but that's just the tip of the iceberg.

To keep with the theme of "everyday development," I've based the majority of my code samples on C# 3.0 with some JavaScript sprinkled in. Please note that some of the other, newer programming languages for the CLR, like IronPython, IronRuby and F#, have substantially stronger support or more advantageous syntax for the functional programming techniques shown in this article. Unfortunately, the current version of Visual Basic does not support multiline Lambda functions, so many of the techniques shown here are not as usable in Visual Basic. However, I would urge Visual Basic developers to consider these techniques in preparation for the next version of the language, shipping in Visual Studio 2010.

First-Class Functions

Some elements of functional programming are possible in C# or Visual Basic because we now have first-class functions that can be passed around between methods, held as variables or even returned from another method. Anonymous delegates from .NET 2.0 and the newer Lambda expression in .NET 3.5 are how C# and Visual Basic implement first-class functions, but "Lambda expression" means something more specific in computer science. Another common term for a first-class function is "block." For the remainder of this article I will use the term "block" to denote first-class functions, rather than "closure" (a specific type of first-class function I discuss next) or "Lambda," to avoid accidental inaccuracies (and the wrath of real functional programming experts). A closure contains variables defined from outside the closure function. If you've used the increasingly popular jQuery library for JavaScript development, you've probably used closures quite frequently. Here's an example of using a closure, taken from my current project:

```
// Expand/contract body functionality
var expandLink = $(".expandLink", itemElement);
var hideLink = $(".hideLink", itemElement);
var body = $(".expandBody", itemElement);
body.hide();

// The click handler for expandLink will use the
// body, expandLink, and hideLink variables even
// though the declaring function will have long
// since gone out of scope.
expandLink.click(function() {
    body.toggle();
    expandLink.toggle();
    hideLink.toggle();
});
```

This code is used to set up a pretty typical accordion effect to show or hide content on our Web pages by clicking an <a> element. We define the click handler of the expandLink by passing in a closure function that uses variables created outside the closure. The function that contains both the variables and the click handler will exit long before the expandLink can be clicked by the user, but the click handler will still be able to use the body and hideLink variables.

Lambdas as Data

In some circumstances, you can use the Lambda syntax to denote an expression in code that can be used as data rather than executed. I didn't particularly understand that statement the first several times I read it, so let's look at an example of treating a Lambda as data from an explicit object/relational mapping using the Fluent NHibernate library:

```
public class AddressMap : DomainMap<Address>
{
    public AddressMap()
    {
        Map(a => a.Address1);
        Map(a => a.Address2);
        Map(a => a.AddressType);
        Map(a => a.City);
        Map(a => a.TimeZone);
        Map(a => a.StateOrProvince);
        Map(a => a.Country);
        Map(a => a.PostalCode);
    }
}
```

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

Fluent NHibernate never evaluates the expression a => a.Address1. Instead, it parses the expression to find the name Address1 to use in the underlying NHibernate mapping. This technique has spread widely through many recent open-source projects in the .NET space. Using Lambda expressions just to get at PropertyInfo objects and property names is frequently called *static reflection*.

Passing Blocks

One of the best reasons to study functional programming is to learn how first-class functions allow you to reduce duplication in code by providing a more fine-grained mechanism for composition than the class. You will often come across sequences of code that are essentially identical in their basic form except for one step somewhere in the middle of the sequence. With object-oriented programming, you can use inheritance with the template method pattern to try to eliminate the duplication. More and more I find that passing blocks representing the variable step in the middle to another method that implements the basic sequence to be a cleaner way to eliminate this duplication.

One of the best ways to make an API easier to use and less prone to error is to reduce repetitive code. For example, consider the common case of an API designed to access a remote service or resource like an ADO.NET IDbConnection object or a socket listener that requires a stateful or persistent connection. You must typically "open" the connection before using the resource. These stateful connections are often expensive or scarce in terms of resources, so it is often important to "close" the connection as soon as you are done to release the resource for other processes or threads.

The following code shows a representative interface for the gateway to a stateful connection of some type:

```
public interface IConnectedResource
{
    void Open();
    void Close();

    // Some methods that manipulate the connected resource
    BigDataOfSomeKind FetchData(RequestObject request);
    void Update(UpdateObject update);
}
```

Every single time another class consumes this IConnectedResource interface, the Open method has to be called before using any other method, and the Close method should always be called afterward, as shown in **Figure 1**.

In an earlier article I discussed the idea of essence versus ceremony in our designs. (See msdn.microsoft.com/magazine/dd419655.aspx.) The "essence" of the ConnectedSystemConsumer class's responsibility is simply to use the connected resource to update some information. Unfortunately, most of the code in ConnectedSystemConsumer is concerned with the "ceremony" of connecting to and disconnecting from the IConnectedResource interface and error handling.

Worse yet is the fact that the "try/open/do stuff/finally/close" ceremony code has to be duplicated for each use of the IConnectedResource interface. As I've discussed before, one of the best ways to improve your design is to stamp out duplication wherever it creeps into your code. Let's try a different approach to the IConnectedResource API using a block or closure. First, I'm going to apply the Interface Segregation Principle (see objectmentor.com/resources/articles/isp.pdf for more information) to extract an interface strictly for invoking the connected resource without the methods for Open or Close:

Figure 1 **Using IConnectedResource**

```
public class ConnectedSystemConsumer
{
    private readonly IConnectedResource _resource;

    public ConnectedSystemConsumer(IConnectedResource resource)
    {
        _resource = resource;
    }

    public void ManipulateConnectedResource()
    {
        try
        {
            // First, you have to open a connection
            _resource.Open();

            // Now, manipulate the connected system
            _resource.Update(buildUpdateMessage());
        }
        finally
        {
            _resource.Close();
        }
    }
}
```

```
public interface IResourceInvocation
{
    BigDataOfSomeKind FetchData(RequestObject request);
    void Update(UpdateObject update);
}
```

Next, I create a second interface that is used strictly to gain access to the connected resource represented by the IResourceInvocation interface:

```
public interface IResource
{
    void Invoke(Action<IResourceInvocation> action);
}
```

Now, let's rewrite the ConnectedSystemConsumer class to use the newer, functional-style API:

```
public class ConnectedSystemConsumer
{
    private readonly IResource _resource;

    public ConnectedSystemConsumer(IResource resource)
    {
        _resource = resource;
    }

    public void ManipulateConnectedResource()
    {
        _resource.Invoke(x =>
        {
            x.Update(buildUpdateMessage());
        });
    }
}
```

This new version of ConnectedSystemConsumer no longer has to care about how to set up or tear down the connected resource. In effect, ConnectedSystemConsumer just tells the IResource interface to "go up to the first IResourceInvocation you see and give it these instructions" by passing in a block or closure to the IResource.Invoke method. All that repetitive "try/open/do stuff/finally/close" ceremony code that I was

Figure 2 **Concrete Implementation of IResource**

```
public class Resource : IResource
{
    public void Invoke(Action<IResourceInvocation> action)
    {
        IResourceInvocation invocation = null;

        try
        {
            invocation = open();

            // Perform the requested action
            action(invocation);
        }
        finally
        {
            close(invocation);
        }
    }

    private void close(IResourceInvocation invocation)
    {
        // close and teardown the invocation object
    }

    private IResourceInvocation open()
    {
        // acquire or open the connection
    }
}
```

complaining about before is now in the concrete implementation of IResource, as shown in **Figure 2**.

I would argue that we've improved our design and API usability by putting the responsibility for opening and closing the connection to the external resource into the Resource class. We've also improved the structure of our code by encapsulating the details of infrastructure concerns from the core workflow of the application. The second version of ConnectedSystemConsumer knows far less about the workings of the external connected resource than the first version did. The second design enables you to more easily change how your system interacts with the external connected resource without changing and potentially destabilizing the core workflow code of your system.

The second design also makes your system less error-prone by eliminating the duplication of the "try/open/finally/close" cycle. Every time a developer has to repeat that code, he risks making a coding mistake that could technically function correctly but exhaust resources and harm the scalability characteristics of the application.

Delayed Execution

One of the most important concepts to understand about functional programming is delayed execution. Fortunately, this concept is also relatively simple. All it means is that a block function you've defined inline doesn't necessarily execute immediately. Let's look at a practical use of delayed execution.

In a fairly large WPF application, I use a marker interface called IStartable to denote services that need to be, well, started up as part of the application bootstrapping process.

```
public interface IStartable
{
    void Start();
}
```

All the services for this particular application are registered and retrieved by the application from an Inversion of Control container (in this case, StructureMap). At application startup, I have the following bit of code to dynamically discover all the services in the application that need to be started and then start them:

```
// Find all the possible services in the main application
// IoC Container that implements an "IStartable" interface
List<IStartable> startables = container.Model.PluginTypes
    .Where(p => p.IsStartable())
    .Select(x => x.ToStartable()).ToList();

// Tell each "IStartable" to Start()
startables.Each(x => x.Start());
```

There are three Lambda expressions in this code. Let's say you attached the full source code copy of the .NET base class library to this code and then tried to step through it with the debugger. When you try to step into the Where, Select, or Each calls, you would notice that the Lambda expressions are not the next lines of code to execute and that as these methods iterate over the internal structures of the container.Model.PluginTypes member, the Lambda expressions are all executed multiple times. Another way to think about delayed execution is that when you invoke the Each method, you're just telling the Each method what to do anytime it comes across an IStartable object.

Memoization

Memoization is an optimization technique used to avoid executing expensive function calls by reusing the results of the previous execution with the same input. I first came into contact with the term memoization in regards to functional programming with F#, but in the course of researching this article I realized that my team fre-

The Map/Reduce Pattern

It turns out that many common development tasks are simpler with functional programming techniques. In particular, list and set operations in code are far simpler mechanically in languages that support the "map/reduce" pattern. (In LINQ, "map" is "Select" and "reduce" is "Aggregate".) Think about how you would calculate the sum of an array of integers. In .NET 1.1, you had to iterate over the array something like this:

```
int[] numbers = new int[]{1,2,3,4,5};
int sum = 0;
for (int i = 0; i < numbers.Length; i++)
{
    sum += numbers[i];
}

Console.WriteLine(sum);
```

The wave of language enhancements to support LINQ in .NET 3.5 provided the map/reduce capabilities common in functional programming languages. Today, the code above could simply be written as:

```
int[] numbers = new int[]{1,2,3,4,5};
int sum = numbers.Aggregate((x, y) => x + y);
```

or more simply as:

```
int sum = numbers.Sum();

Console.WriteLine(sum);
```

Figure 3 **Implementing an Inner IFinancialDataService Class**

```
public class MemoizedFinancialDataService : IFinancialDataService
{
    private readonly Cache<string, FinancialData> _cache;

    // Take in an "inner" IFinancialDataService object that actually
    // fetches the financial data
    public MemoizedFinancialDataService(IFinancialDataService
      innerService)
    {
        _cache = new Cache<string, FinancialData>(region =>
          innerService.FetchData(region));
    }

    public FinancialData FetchData(string region)
    {
        return _cache[region];
    }
}
```

quently uses memoization in our C# development. Let's say you often need to retrieve some sort of calculated financial data for a given region with a service like this:

```
public interface IFinancialDataService
{
    FinancialData FetchData(string region);
}
```

IFinancialDataService happens to be extremely slow to execute and the financial data is fairly static, so applying memoization would be very beneficial for application responsiveness. You could create a wrapper implementation of IFinancialDataService that implements memoization for an inner IFinancialDataService class, as shown in **Figure 3**.

The Cache<TKey, TValue> class itself is just a wrapper around a Dictionary<TKey, TValue> object. **Figure 4** shows part of the Cache class.

If you're interested in the internals of the Cache class, you can find a version of it in several open-source software projects, including StructureMap, StoryTeller, FubuMVC and, I believe, Fluent NHibernate.

Continuations

Roughly put, a continuation in programming is an abstraction of some sort that denotes "what to do next" or the "rest of the computation." Sometimes it is valuable to finish part of a computational process at another time, as in a wizard application in which a user can explicitly allow the next step or cancel the whole process.

Let's jump right into a code sample. Say you are developing a desktop application in WinForms or WPF. You frequently need to initiate some type of long-running process or access a slow external service from a screen action. For the sake of usability, you certainly do not want to lock up the user interface and make it unresponsive while the service call is happening, so you run it in a background thread. When the service call does finally return, you may want to update the user interface with the data coming back from the service, but as any experienced WinForms or WPF developer knows, you can update user interface elements only on the main user interface thread.

You can certainly use the BackgroundWorker class that's in the System.ComponentModel namespace, but I prefer a different approach based on passing Lambda expressions into a CommandExecutor object, represented by this interface:

```
public interface ICommandExecutor
{
    // Execute an operation on a background thread that
    // does not update the user interface
    void Execute(Action action);

    // Execute an operation on a background thread and
    // update the user interface using the returned Action
    void Execute(Func<Action> function);
}
```

The first method is simply a statement to perform an activity in a background thread. The second method that takes in a Func<Action> is more interesting. Let's look at how this method would typically be used within application code.

Figure 4 **The Cache Class**

```
public class Cache<TKey, TValue> : IEnumerable<TValue> where TValue :
  class
{
    private readonly object _locker = new object();
    private readonly IDictionary<TKey, TValue> _values;

    private Func<TKey, TValue> _onMissing = delegate(TKey key)
    {
        string message = string.Format(
          "Key '{0}' could not be found", key);
        throw new KeyNotFoundException(message);
    };

    public Cache(Func<TKey, TValue> onMissing)
        : this(new Dictionary<TKey, TValue>(), onMissing)
    {
    }

    public Cache(IDictionary<TKey, TValue>
      dictionary, Func<TKey, TValue>
      onMissing)
        : this(dictionary)
    {
        _onMissing = onMissing;
    }

    public TValue this[TKey key]
    {
        get
        {
            // Check first if the value for the requested key
            // already exists
            if (!_values.ContainsKey(key))
            {
                lock (_locker)
                {
                    if (!_values.ContainsKey(key))
                    {
                        // If the value does not exist, use
                        // the Func<TKey, TValue> block
                        // specified in the constructor to
                        // fetch the value and put it into
                        // the underlying dictionary
                        TValue value = _onMissing(key);
                        _values.Add(key, value);
                    }
                }
            }

            return _values[key];
        }
    }
}
```

First, assume that you're structuring your WinForms or WPF code with the Supervising Controller form of the Model View Presenter pattern. (See msdn.microsoft.com/magazine/cc188690.aspx for more information on the MVP pattern.) In this model, the Presenter class is going to be responsible for calling into a long-running service method and using the return data to update the view. Your new Presenter class will simply use the ICommandExecutor interface shown earlier to handle all the threading and thread marshalling work, as shown in **Figure 5**.

The Presenter class calls ICommandExecutor.Execute by passing in a block that returns another block. The original block invokes the long-running service call to get some data, and returns a Continuation block that can be used to update the user interface (the IView in this scenario). In this particular case, it's important to use the Continuation approach instead of just updating the IView at the same time because the update has to be marshaled back to the user interface thread.

Figure 6 shows the concrete implementation of the ICommandExecutor interface.

The Execute(Func<Action>) method invokes Func<Action> in a background thread and then takes the Continuation (the Action returned by Func<Action>) and uses a SynchronizationContext object to execute the Continuation in the main user interface thread.

I like passing blocks into the ICommandExecutor interface because of how little ceremonial code it takes to invoke the background processing. In an earlier incarnation of this approach, before we had Lambda expressions or anonymous delegates in C#, I had a similar implementation that used little Command pattern classes like the following:

```
public interface ICommand
{
    ICommand Execute();
}

public interface JeremysOldCommandExecutor
{
    void Execute(ICommand command);
}
```

The disadvantage of the former approach is that I had to write additional Command classes just to model the background operation and the view-updating code. The extra class declaration and constructor functions are a little more ceremony code we can eliminate with the functional approach, but more important to me is that the functional approach allows me to put all this closely related code in a single place in the Presenter rather than having to spread it out over those little Command classes.

Continuation Passing Style

Building on the Continuation concept, you can use the Continuation Passing style of programming to invoke a method by passing in a Continuation instead of waiting for the return value of the method. The method accepting the Continuation is in charge of deciding whether and when to call the Continuation.

In my current Web MVC project, there are dozens of controller actions that save updates from user input sent from the client browser via an AJAX call to a domain entity object. Most of these controller actions simply invoke our Repository class to save the changed entity, but other actions use other services to perform the persistence work. (See my article in the April issue of *MSDN Magazine* at msdn.microsoft.com/magazine/dd569757.aspx for more information about the Repository class.)

The basic workflow of these controller actions is consistent:

1. Validate the domain entity and record any validation errors.
2. If there are validation errors, return a response to the client indicating that the operation failed and include the validation errors for display on the client.

Figure 5 **The Presenter Class**

```
public class Presenter
{
    private readonly IView _view;
    private readonly IService _service;
    private readonly ICommandExecutor _executor;

    public Presenter(IView view, IService service, ICommandExecutor
      executor)
    {
        _view = view;
        _service = service;
        _executor = executor;
    }

    public void RefreshData()
    {
        _executor.Execute(() =>
        {
            var data = _service.FetchDataFromExtremelySlowServiceCall();
            return () => _view.UpdateDisplay(data);
        });
    }
}
```

Figure 6 **Concrete Implementation of ICommandExecutor**

```
public class AsynchronousExecutor : ICommandExecutor
{
    private readonly SynchronizationContext _synchronizationContext;
    private readonly List<BackgroundWorker> _workers =
      new List<BackgroundWorker>();

    public AsynchronousExecutor(SynchronizationContext
      synchronizationContext)
    {
        _synchronizationContext = synchronizationContext;
    }

    public void Execute(Action action)
    {
        // Exception handling is omitted, but this design
        // does allow for centralized exception handling
        ThreadPool.QueueUserWorkItem(o => action());
    }

    public void Execute(Func<Action> function)
    {
        ThreadPool.QueueUserWorkItem(o =>
        {
            Action continuation = function();
            _synchronizationContext.Send(x => continuation(), null);
        });
    }
}
```

Figure 7 **A Sample Controller Action**

```
public class SolutionController
{
    private readonly IPersistor _persistor;
    private readonly IWorkflowService _service;

    public SolutionController(IPersistor persistor, IWorkflowService
      service)
    {
        _persistor = persistor;
        _service = service;
    }

    // UpdateSolutionViewModel is a data bag with the user
    // input from the browser
    public CrudReport Create(UpdateSolutionViewModel update)
    {
        var solution = new Solution();

        // Push the data from the incoming
        // user request into the new Solution
        // object
        update.ToSolution(solution);

        // Persist the new Solution object, if it's valid
        return _persistor.Persist(solution, x => _service.Create(x));
    }
}
```

3. If there are no validation errors, persist the domain entity and return a response to the client indicating that the operation succeeded.

What we would like to do is centralize the basic workflow but still allow each individual controller action to vary how the actual persistence is done. Today my team is doing this by inheriting from a CrudController<T> superclass with plenty of template methods that each subclass can override to add or change the basic behavior. This strategy worked out well at first, but it is rapidly breaking down as the variations have increased. Now my team is going to move to using Continuation Passing style code by having our controller actions delegate to something like the following interface:

```
public interface IPersistor
{
    CrudReport Persist<T>(T target, Action<T> saveAction);
    CrudReport Persist<T>(T target);
}
```

A typical controller action would tell IPersistor to perform the basic CRUD workflow and supply a Continuation that IPersistor uses to actually save the target object. **Figure 7** shows a sample controller action that invokes IPersistor but uses a different service than our basic Repository for the actual persistence.

I think the important thing to note here is that IPersistor itself is deciding whether and when the Continuation supplied by SolutionController will be called. **Figure 8** shows a sample implementation of IPersistor.

Write Less Code

Frankly, I originally chose this topic because I was interested in learning more about functional programming and how it can be applied even within C# or Visual Basic. In the course of writing this article, I've learned a great deal about just how useful functional programming techniques can be in normal, everyday tasks. The most important conclusion I've reached, and what I've tried to convey here, is that compared with other techniques, functional programming approaches can often lead to writing less code and often more declarative code for some tasks—and that's almost always a good thing. ■

JEREMY MILLER, *a Microsoft MVP for C#, is also the author of the open source StructureMap (structuremap.sourceforge.net) tool for Dependency Injection with .NET and the forthcoming StoryTeller (storyteller.tigris.org) tool for supercharged FIT testing in .NET. Visit his blog, The Shade Tree Developer, at codebetter.com/blogs/jeremy.miller, part of the CodeBetter site.*

Figure 8 **An Implementation of IPersistor**

```
public class Persistor : IPersistor
{
    private readonly IValidator _validator;
    private readonly IRepository _repository;

    public Persistor(IValidator validator, IRepository repository)
    {
        _validator = validator;
        _repository = repository;
    }

    public CrudReport Persist<T>(T target, Action<T> saveAction)
    {
        // Validate the "target" object and get a report of all
        // validation errors
        Notification notification = _validator.Validate(target);

        // If the validation fails, do NOT save the object.
        // Instead, return a CrudReport with the validation errors
        // and the "success" flag set to false
        if (!notification.IsValid())
        {
            return new CrudReport()
            {
                Notification = notification,
                success = false
            };
        }


        // Proceed to saving the target using the Continuation supplied
        // by the client of IPersistor
        saveAction(target);


        // return a CrudReport denoting success
        return new CrudReport()
        {
            success = true
        };
    }

    public CrudReport Persist<T>(T target)
    {
        return Persist(target, x => _repository.Save(x));
    }
}
```

Building RESTful Clients

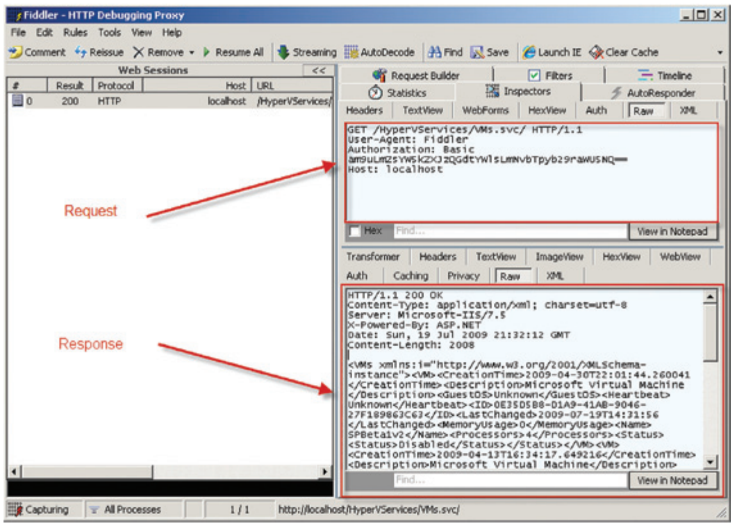
In this installment of the RESTful Service Station, I'll discuss building clients against RESTful services. Building clients is perceived as difficult (mostly because of the lack of automatic client generation from metadata, à la SOAP and WSDL), but in reality it's like any other type of code that you write: At first there is some ramp-up time as you get used to a particular programming paradigm. Then, once you have the hang of that paradigm, writing code against it becomes easier and easier. I've found this to be true for writing clients against RESTful services, as have many other developers I've interacted with.

Client Basics

I'm going to show the basics of interaction between a client and a service using REST. Just a quick recap about REST before we dig in. REST is an architectural style that builds on the same principles that provide the foundation of the World Wide Web. Clients interact with services by making HTTP requests, and services respond with HTTP responses, which often include a representation of a resource that client code can use to drive an application.

Imagine I have a service that exposes information and functionality related to Hyper-V (the virtualization technology built into Windows Server 2008). If the "entry" URI of this service was http://localhost/HyperVServices/VMs.svc/, as a client I'd make an HTTP GET request to retrieve a representation of the resource identified by this URI. In this case, the resource is formatted as XML and represents a list of all the virtual machines (VM)s installed on a particular machine.

As I showed in the first of my Service Station articles about REST, in the January 2009 issue of *MSDN Magazine* (msdn.microsoft.com/magazine/2009.01.servicestation.aspx), one of the small benefits of using REST is that you can use HTTP request tools to make your initial test requests against a service. (Using these tools for debugging problems is also very useful.) In this case, I'm using a free tool called Fiddler (fiddlertool.com) to make an HTTP GET request for the resource that represents all my virtual machines. See **Figure 1**. Of course, tools like Fiddler are valuable, but to build an application you need to write code against the service. I'm going to start with the basics of making an HTTP request and then get into the issue of actually reading the resource.


Fiddler - HTTP Debugging Proxy
File Edit Rules Tools View Help
Comment Reissue Remove Resume All Streaming AutoDecode Find Save Launch IE Clear Cache
Web Sessions
# Result Protocol Host URL
0 200 HTTP localhost /HyperVServices/
Request Builder Filters Timeline
Statistics Inspectors AutoResponder
Headers TextView WebForms HexView Auth Raw XML
Request
Response
Hex Find... View in Notepad
Transformer Headers TextView ImageView HexView WebView
Auth Caching Privacy Raw XML
Capturing All Processes 1 / 1 http://localhost/HyperVServices/VMs.svc/

```
GET /HyperVServices/VMs.svc/ HTTP/1.1
User-Agent: Fiddler
Authorization: Basic
am9uLmZsYW5kZXJzQGdtYWlsLmNvbTpyb29raWU5NQ==
Host: localhost
```

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Sun, 19 Jul 2009 21:32:12 GMT
Content-Length: 2008

<VMs xmlns:i="http://www.w3.org/2001/XMLSchema-
instance"><VM><CreationTime>2009-04-30T22:01:44.260041
</CreationTime><Description>Microsoft Virtual Machine
</Description><GuestOS>Unknown</GuestOS><Heartbeat>
Unknown</Heartbeat><ID>0E35D5B8-D1A9-41AB-9046-
27F189863C63</ID><LastChanged>2009-07-19T14:31:56
</LastChanged><MemoryUsage>0</MemoryUsage><Name>
SPBeta1v2</Name><Processors>4</Processors><Status>
<Status>Disabled</Status></Status></VM><VM>
<CreationTime>2009-04-13T16:34:17.649216</CreationTime>
<Description>Microsoft Virtual Machine</Description>
```

Figure 1 **A Simple HTTP GET Request**

The .NET Framework has always offered a basic HTTP API that can be used to interact with RESTful services. At the center of this API are the HttpWebRequest and HttpWebResponse types. To make an HTTP request, you create and configure an HttpWebRequest instance and then ask for the HttpWebResponse. **Figure 2** shows an example. Making a request to a RESTful service is a simple set of steps:

1. Make sure you are using the correct URI.
2. Make sure you are using the correct method (HTTP verb).
3. Process the response if the status code is 200 OK.
4. Deal with other status codes as appropriate (more on this later).

Because steps 1 and 2 are pretty easy, generally speaking step 3 (and sometimes step 4) end up being the more difficult of the steps.

The majority of the work of step 3 is processing the resource representation in some reasonable way. Since XML is still the most typical resource response, I'm going to cover that in this article. (The other most likely media type would be JSON.) When the resource is XML, you need to parse that resource and write the code that can extract the data you need from that resource. Using .NET, you have a few op-

Send your questions and comments for Jon to sstation@microsoft.com.

Code download available at code.msdn.microsoft.com/mag200910ServiceStat.

Figure 2 **Simple HttpWebRequest Code**

```
string uri = "http://localhost/HyperVServices/VMs.svc/";
var webRequest = (HttpWebRequest)WebRequest.Create(uri);

//this is the default method/verb, but it's here for clarity
webRequest.Method = "GET";
var webResponse = (HttpWebResponse)webRequest.GetResponse();

Console.WriteLine("Response returned with status code of 0}",
  webResponse.StatusCode);

if (webResponse.StatusCode == HttpStatusCode.OK)
    ProcessOKResponse(webResponse);
else
    ProcessNotOKResponse(webResponse);
```

tions for parsing XML. There is the tried and true XmlReader class. The XmlDocument class is also a possibility, and by using either of those types you can manually parse or use XPath to navigate your way around the XML. With the advent of .NET 3.0, you also get the XDocument class, which, when combined with LINQ to XML, has become the de facto choice for processing XML. **Figure 3** shows an example of code processing the list of VMs using XDocument.

LINQ to XML, along with anonymous types, provides a nice, easy way to deal with turning XML into objects that an application can process, and, of course, you can also use a predefined type instead of an anonymous type.

Another approach that is popular when programming against SOAP-based and REST-based services is to cause the responses to be automatically deserialized into .NET types. In the case of SOAP, this generally happens in the WSDL-generated proxy. With Windows Communication Foundation (WCF) and REST, this can be accomplished in a couple of ways. One way (which I don't really recommend but am mentioning for completeness) is to use the symmetrical nature of WCF to use a WCF service contract definition on the client. In fact, the WCF support for REST includes a type named WebChannelFactory that can be used to create a client channel against a service contract definition. I don't recommend this type of client programming for two reasons. First, creating the client becomes a very manual and error-prone operation. Second, using a strongly typed service contract creates a tight coupling between your client and the service. Avoiding tight coupling is one of the main reasons that the Web has succeeded, and we want to continue that trend when using the Web programmatically.

Another way to use XML serialization is to use the HttpWebResponse.GetResponseStream method and deserialize the XML

Figure 3 **Processing a RESTful Response Using XDocument**

```
var stream = webResponse.GetResponseStream();
var xr = XmlReader.Create(stream);
var xdoc = XDocument.Load(xr);
var vms = from v in xdoc.Root.Elements("VM")
          select new { Name = v.Element("Name").Value};
foreach (var item in vms)
{
    Console.WriteLine(item.Name);
}
```

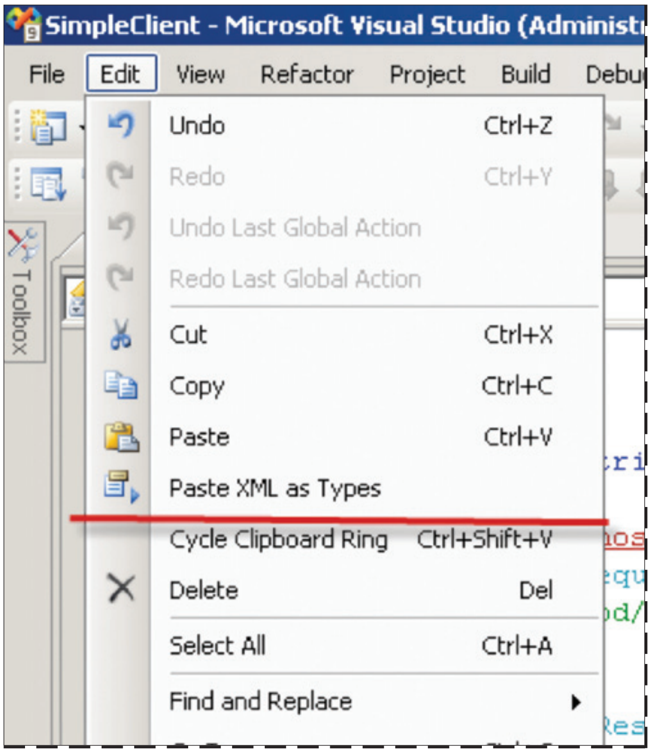


Figure 4 **Paste XML as Types Menu Item**

into the object manually. You can do this by using either the XmlSerializer or the WCF DataContract serializer. In most cases, the XmlSerializer is the preferable approach because it deals with more variations in XML documents (for example, attributes and nonqualified elements) than the DataContract serializer does.

The problem still seems to circle back to the fact that RESTful services generally don't expose metadata that can be used to autogenerate some or all of this code. Although many in the RESTful camp don't see this as a problem (again, anything that autogenerates against service definitions can be seen as the evil agent of tight coupling), there certainly are times when not having these tools is a hindrance to the adoption of REST.

One interesting approach taken by the WCF REST Starter Kit (asp.net/downloads/starter-kits/wcf-rest/), which is an enhancement from Microsoft of the REST programming model in .NET 3.5, is to provide a partial autogeneration feature. If you install the starter kit, you will have a new menu item in Visual Studio 2008 under the Edit menu, as you can see in **Figure 4**.

The usage model of this command is pretty simple. You copy the XML resource representation onto the Clipboard (either from some human-readable documentation the service exposes or by making a request with a tool like Fiddler). Once you do this, a set of XmlSerializable types is created, which you can then use to turn the Stream from the HttpWebResponse into an object. See **Figure 5** (the body of the generated types isn't shown for brevity's sake).

The REST Starter Kit not only simplifies the use of the XmlSerializer, but it also provides a very nice API on top of the

Figure 5 **Code Generated from Using Paste XML as Types**

```
var stream = webResponse.GetResponseStream();
//in a real app you'd want to cache this object
var xs = new XmlSerializer(typeof(VMs));
var vms = xs.Deserialize(stream) as VMs;
foreach (VMsVM vm in vms.VM)
{
    Console.WriteLine(vm.Name);
}
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "",
  IsNullable = false)]
public partial class VMs
{

    private VMsVM[] vmField;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("VM")]
    public VMsVM[] VM
    {
        get
        {
            return this.vmField;
        }
        set
        {
            this.vmField = value;
        }
    }
}
/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.Xml",
"2.0.50727.4918")]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType = true)]
public partial class VMsVM
{
//omitted
}
```

HttpWebRequest/WebResponse API. The following code shows the simple GET request from Figure 1 rewritten using the REST Starter Kit's HttpClient API:

```
string uri = "http://localhost/HyperVServices/VMs.svc/";
var client = new HttpClient();
var message = client.Get(uri);
ProcessOKResponse(message.Content.ReadAsStream());
```

The HttpClient class greatly simplifies (and makes explicit what verbs you are using) the use of the .NET HTTP API. Also, as you can see in the following code, it simplifies the use of the Paste XML As Types generated feature:

```
var vms = XmlSerializerContent.ReadAsXmlSerializable<VMs>(
  message.Content);
```

Although the REST Starter Kit isn't officially supported by Microsoft yet, it does illustrate one very important point: that REST client programming can be simplified and partially automated without a RESTful service having complete metadata à la a WSDL file.

Using HTTP

One way that a client and a service can take advantage of HTTP is to properly use status codes and headers. More information about status codes can be found at w3.org/Protocols/rfc2616/rfc2616-sec10.html.

As I've gone around the world speaking about REST, one of the advantages of RESTful services that I often point out is that GET requests can be cached. Make no mistake, the scalability and the success of the Web is largely because of HTTP caching.

One way to take advantage of HTTP caching is to use conditional GET. Conditional GET enables the client ("user agent" in HTTP lingo) to make a GET request for a resource that the user agent already has a copy of. If the resource hasn't changed, the server tells the user agent that the resource is exactly the same as the version already held by the user agent. The efficiency benefit of conditional GET is a reduction in bandwidth usage of the network between the server and the user agent, freeing up the bandwidth to be used by requests for newly created or modified resources. In addition, it saves the additional processing time required to serialize the resource, although not the processing time to generate or retrieve the resource (since

Figure 6 **Server-Side Implementation of Conditional GET**

```
[OperationContract]
[WebGet(UriTemplate = "/{name}")]
public VMData GetOne(string name)
{
  VMManager.Connect();
  var v = VMManager.GetVirtualMachine(name);
  var newVM = FromVM(v);
  string etag = GenerateETag(newVM);
  if (CheckETag(etag))
      return null;
  if (newVM == null)
  {
      OutgoingWebResponseContext ctx =
        WebOperationContext.Current.OutgoingResponse;
      ctx.SetStatusAsNotFound();
      ctx.SuppressEntityBody = true;
  }
  SetETag(etag);
  return newVM;
}
private bool CheckETag(string currentETag)
{
  IncomingWebRequestContext ctx =
      WebOperationContext.Current.IncomingRequest;
  string incomingEtag =
      ctx.Headers[HttpRequestHeader.IfNoneMatch];
  if (incomingEtag != null)
  {
      if (currentETag == incomingEtag)
      {
          SetNotModified();
          return true;
      }
  }
  return false;
}

string GenerateETag(VMData vm)
{
    byte[] bytes = Encoding.UTF8.GetBytes(vm.ID +
      vm.LastChanged.ToString());
    byte[] hash = MD5.Create().ComputeHash(bytes);
    string etag = Convert.ToBase64String(hash);
    return string.Format("\"{0}\"", etag);
}

void SetETag(string etag)
{
  OutgoingWebResponseContext ctx =
      WebOperationContext.Current.OutgoingResponse;
  ctx.ETag = etag;
}
```

you need a copy of the current resource to compare it with the information sent by the user agent with the conditional GET).

Like most of the other "advanced" HTTP concepts, WCF doesn't support conditional GET automatically because the implementation of conditional GET is highly variable among service implementations. However, as it does for other "advanced" HTTP concepts, WCF provides the tools to implement conditional GET. There are two approaches to accomplish this: using the time the resource was last modified, or using a unique identifier known as an ETag.

ETags have become the more popular way to implement condition GET. In **Figure 6**, you can see the code to implement an ETag-based conditional GET on the VM service. Please note that this is the server-side implementation; I'll get to the client in a moment.

The basic flow is for the server to look for the ETag in the If-None-Match HTTP header sent by the client, and to try to match it against the current ETag generated for the resource. In this case, I'm using the unique ID of each VM plus the last-modified time stamp (converted

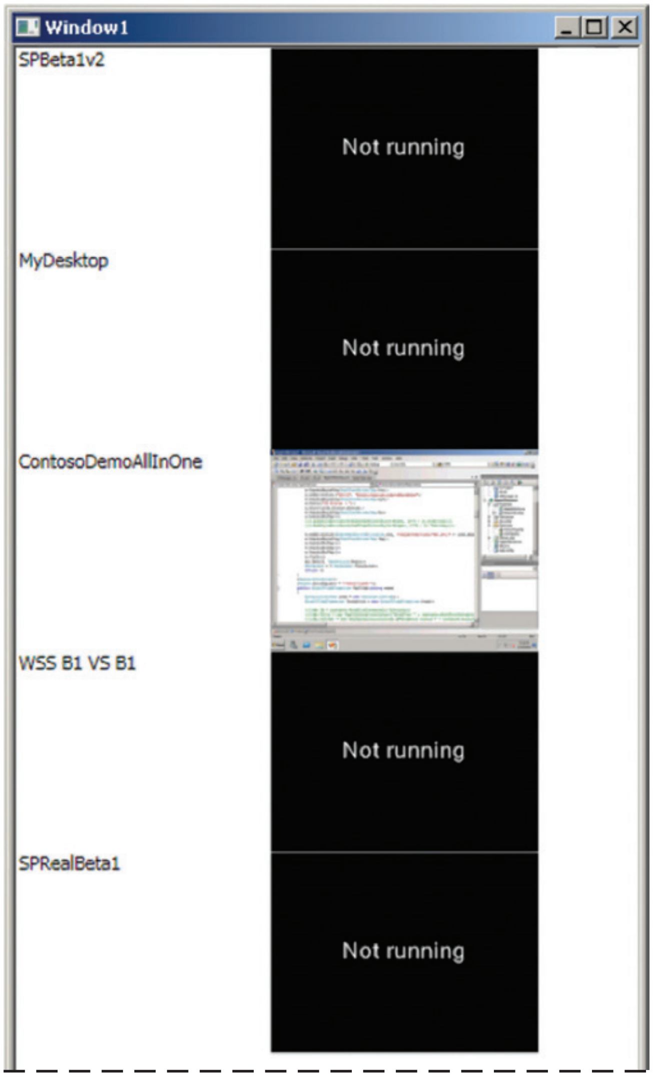


Figure 7 **A Simple WPF VM Client**

to bytes and then made into an MD5 hash, which is a fairly common implementation). If the two values match, the server sends back a 304 not modified HTTP header with an empty body, saving serialization time as well as bandwidth. The client gets a much quicker response and knows it can use the same resource it already has.

Imagine a client application like the one shown in **Figure 7**.This application shows the name of each VM, plus the image of the current status of the VM. Now imagine that you want to update this application to match the current state of each VM. To accomplish this, you would have to write some code in a timer and update each record if it was different from your local record. And it's likely that you'd update everything upon every iteration just to simplify the application, but this would be a waste of resources.

If you instead used conditional GET on the client, you could poll the service for changes by sending a conditional GET request, which would use an If-None-Match HTTP header to indicate the ETag of the resource. For the collection of VMs, the service can use the most recently changed VM to generate the ETag, and the client will update only if one or more of the VMs have changed their state. (If you had lots of VMs, you might want to do this for each VM. But since we are databinding to the collection in this application, I'm OK with updating the whole collection).

Now, implementing this logic isn't terribly difficult, but it is one of the features that the REST Starter Kit implements for you. Included in the code samples is a file named PollingAgent.cs, which has an automatic conditional GET client that polls a RESTful endpoint on an interval that you define. When the PollingAgent determines that the resource has changed (because the service is no longer returning a 302), it fires a callback.

So in my simple WPF application, I just take the result of that callback (which is the HttpResponseMessage object) and rebind my controls to the new data. **Figure 8** shows the code to implement this application using the PollingAgent.

Following Hypertext

Before leaving the subject of clients, I want to touch on another of the important constraints of REST: using hypermedia as the engine of application state (known as HATEOAS).

I find that HATEOAS is easier to understand in the context of the human Web. When using the human Web, sometimes we have bookmarks to sites we know we want to visit. Within those sites, however, we generally follow hyperlinks based on our needs of the day. I might have a bookmark to www.amazon.com, but when I want to buy something, I follow the links on each page (resource) to add an item to my cart. And then I follow the hyperlinks (and at times forms) on each subsequent page to process my order. At each stage of order processing, the links in the page represent the current state of the application (in other words, what can I, as the user agent, do?).

One important consideration when building RESTful clients (although part of this is incumbent on RESTful service implementers as well) is to keep "bookmarks" to a minimum. What this means practically is that clients should have as little knowledge as possible about the URI structure of your representations, but as much as possible they should

Figure 8 **Conditional GET Using the REST Starter Kit Polling Agent**

```
public Window1()
{
  InitializeComponent();
  string uri = "http://localhost/HyperVServices/VMs.svc/";
  var client = new HttpClient();
  var message = client.Get(uri);
  var vms = XmlSerializerContent.ReadAsXmlSerializable<VMs>(
    message.Content);
  _vmList.DataContext = vms.VM;
  var pa = new PollingAgent();
  pa.HttpClient = client;
  pa.ResourceChanged += new EventHandler<ConditionalGetEventArgs>(
    pa_ResourceChanged);
  pa.PollingInterval = new TimeSpan(0, 0, 5);
  pa.StartPolling(new Uri(uri), message.Headers.ETag, null);
}

void pa_ResourceChanged(object sender, ConditionalGetEventArgs e)
{
  var vms = XmlSerializerContent.ReadAsXmlSerializable<VMs>(
    e.Response.Content);
  _vmList.DataContext = vms.VM;
}
```

know how to "navigate" the "hyperlinks" inside those resources. I put "hyperlinks" in quotes because any piece of data in your resource can be a relative URI for building the next "link" for the client.

In my example, each VM's name is really a link, because a client can ask for a particular VM's data by asking for the name as part of the URI. So http://localhost/HyperVServices/VMs.svc/MyDesktop (where MyDesktop is the value of the Name element in the VM's element inside the collection) is the URI for the MyDesktop VM resource.

Imagine that this RESTful service allows for provisioning and starting VMs. It would make sense to embed inside of each VM resource hyperlinks to the various statuses that a VM could be put in at a particular time, rather than letting the client start a VM that hasn't been provisioned properly yet.

Using HATEOAS helps to verify the current state of the application, and it also encourages more loosely coupled clients (since the clients don't have to know as much about the URI structure of the service).

Easier Than You Think

There is no doubt that building RESTful clients is harder to start doing than building SOAP-based clients supported by WSDL metadata. As I wrote in my previous column, "REST is simple, but simple doesn't necessarily mean easy. SOAP is easy (because of WSDL), but easy doesn't always mean simple."With good coding practices and tools like the REST Starter Kit, building RESTful clients can be easier than you think. And in the end I think you'll find that the advantages you get from using the architectural style will more than make up for any temporary delay in building clients. ■

JON FLANDERS *is an independent consultant, speaker and trainer for Pluralsight. He specializes in BizTalk Server, Windows Workflow Foundation and Windows Communication Foundation. You can contact him at masteringbiztalk.com/blogs/jon.*

JUVAL LOWY

Routers in the Service Bus

In my previous two columns, I described the core problem the .NET service bus was designed to solve—that of Internet connectivity and related security issues. However, the service bus offers much more than mere connectivity and messaging relay. This column starts with the service registry and then describes aspects of routers. I'll dedicate my next column to queues in the cloud. In both columns, I will describe new powerful design patterns, how to combine routers and queues, and my helper classes and tools to streamline the overall experience.

The Services Registry

The .NET service bus offers an ATOM feed of listening services on the solution base address or on any one of its sub-URIs. You can view the feed by navigating to the solution (or the URI) with a browser. The feed serves as a registry and a directory for the services and aspects (such as routers and queues) in the solution.

By default, your service is not visible in the browser. You control service registry publishing by using the ServiceRegistrySettings endpoint behavior class, which is defined as follows:

```
public enum DiscoveryType
{
   Public,
   Private
}
public class ServiceRegistrySettings : IEndpointBehavior
{
   public ServiceRegistrySettings();
   public ServiceRegistrySettings(DiscoveryType discoveryType);

   public DiscoveryType DiscoveryMode
   {get;set;}

   public string DisplayName
   {get;set;}
}
```

The host needs to add that behavior programmatically to every endpoint you want to publish to the registry. (There is no matching configurable behavior.) For example, to publish all endpoints to the registry, you would use the code shown here:

```
IEndpointBehavior registeryBehavior =
          new ServiceRegistrySettings(DiscoveryType.Public);

ServiceHost host = new ServiceHost(typeof(MyService));

foreach(ServiceEndpoint endpoint in host.Description.Endpoints)
{
   endpoint.Behaviors.Add(registeryBehavior);
}
host.Open();
```

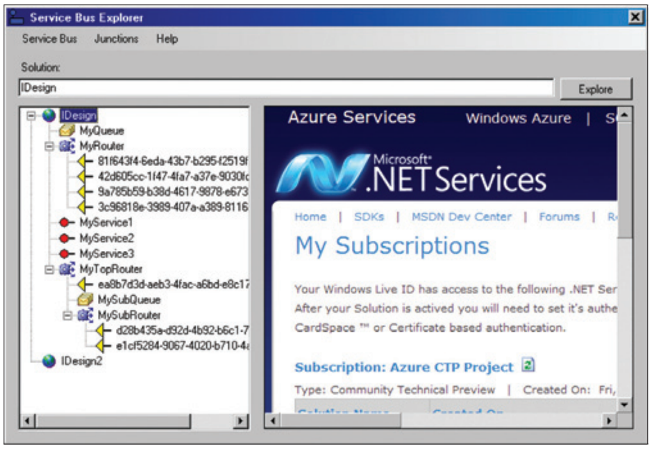


Figure 1 **The Services Bus Explorer**

To help visualize the service bus, I developed the Service Bus Explorer, shown in **Figure 1**. The tool accepts the name of the solution to explore, and after logging onto the feed, will visualize for you the running services. All the explorer does is parse the ATOM feed and place the items in the tree to the left. You can explore multiple solutions and see in the right pane your solution administration pages. The tool also shows the available routers and queues and is very handy in administering them.

Cloud as Interceptor

The service bus was indeed initially developed to address the acute connectivity issues of calls across the Web. However, it has the potential for much more. Compare the basic Windows Communication Foundation (WCF) architecture with the service bus architecture. In both cases, the client does not interact directly with the service, but instead the calls are intercepted by middleware. In the case of regular WCF, the middleware is the proxy and the interpretation chain leading to the service, as shown in **Figure 2**.

In the case of the relayed calls, the middleware consists of the regular WCF middleware and the service bus itself, as shown in **Figure 3**. From an architecture standpoint, this is the same

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

Code download available at code.msdn.microsoft.com/mag200910Foundations.

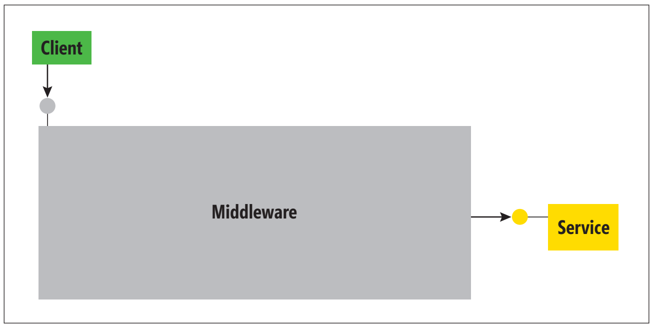


Figure 2 **Intercepting Regular WCF Calls**

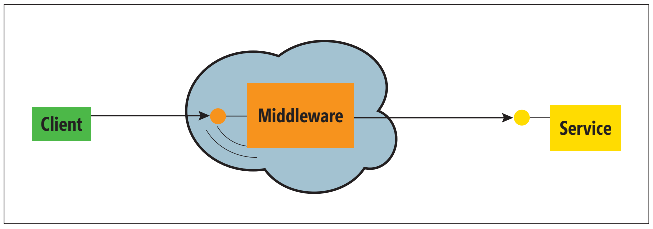


Figure 3 **The Cloud as Interceptor**

design—intercept the calls to provide additional value. In the current release of the service bus , that additional value is the ability to install routers and queues. In fact, I believe the service bus has great potential for powerful interceptors, and no doubt additional aspects will become available over time.

Junctions as Routers

In the service bus, every URI in the solution is actually an addressable messaging junction. The client can send a message to that junction, and the junction can relay it to the services. However, each junction can also function as a router. Multiple services can subscribe to that router, and the router can forward the same client message to all of them or just a single one of them, as shown in **Figure 4**.

The client is decoupled from the services behind the router, and as far as the client is concerned, there may not even be any services behind the router. All the client needs to know is where the router it needs to send messages to is located. It trusts the router to be configured with the appropriate distribution policy. Because of that indirectness, the messages are inherently one way, and, in fact, WCF insists that the binding used by both the client and the subscribing services is the one-way relay binding. The client has no way of receiving the results from the services or of being apprised of service-side errors. This would be the case even if the binding were not one way. If the client message is being multiplexed to multiple services, then by definition there is no meaning for results returned from the operation (from which service?) or for errors (of which service?). There is also no way for the services to call back to their client.

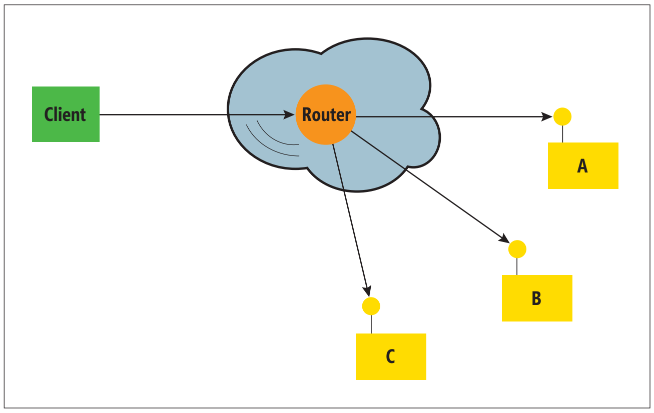


Figure 4 **The Services Bus as a Router**

Router Policy

The solution administrator manages the routers independently of services. Each router must have a policy governing its behavior and lifetime. Out of the box, the solution administrator must perform programmatic calls to create and manage routers. All messaging junction policies derive from the abstract class JunctionPolicy, shown in **Figure 5**.

The Discoverability junction property is an enum of the type DiscoverabilityPolicy. It controls whether the junction is included in the ATOM feed of the solution. Discoverability defaults to DiscoverabilityPolicy.Managers, meaning a private policy. Setting it to DiscoverabilityPolicy.Public publishes it to the feed. The ExpirationInstant property controls the lifetime of the junction. Once the policy has expired, the junction is removed. ExpirationInstant defaults to one day. Obviously, that may or may not be adequate for your routers (or queues), so you typically need to set it to the value you want and monitor it. As the junction is about to expire, you should renew it if the junction is still in use. I call that renewing the "lease time" of the junction, and I call the party that extends the lease the "sponsor." Finally, the TransportProtection property controls the transfer security of the message to the junction. When

Figure 5 **The JunctionPolicy Class**

```
public enum DiscoverabilityPolicy
{
   Managers,Public //More values
}
public enum TransportProtectionPolicy
{
   None,AllPaths
}

[DataContract]
public abstract class JunctionPolicy
{
   public JunctionPolicy();
   public JunctionPolicy(JunctionPolicy policyToCopy);

   public DateTime ExpirationInstant
   {get;set;}

   public DiscoverabilityPolicy Discoverability
   {get;set;}

   public TransportProtectionPolicy TransportProtection
   {get;set;}

   //More members
}
```

Figure 6 **The RouterPolicy Class**

```
public enum MessageDistributionPolicy
{
   AllSubscribers,
   OneSubscriber
}

[DataContract]
public class RouterPolicy : JunctionPolicy,...
{
   public RouterPolicy();
   public RouterPolicy(RouterPolicy otherRouterPolicy);

   public int MaxSubscribers
   {get;set;}

   public MessageDistributionPolicy MessageDistribution
   {get;set;}

   //More members
}
```

posting or retrieving raw WCF messages to or from the messaging junctions, you are restricted to using transport security with the value of TransportProtectionPolicy.AllPaths. Transport security is the default for all junction policies, and I recommend never setting it to any other value.

Each router policy is expressed with the class RouterPolicy, defined in **Figure 6**. The two main properties here are MaxSubscribers and MessageDistribution. As its name implies, MaxSubscribers is the maximum number of concurrent subscribers allowed to connect to the router. The default value is 1, and the maximum value is 50. Once the router is maxed-out, additional services trying to subscribe get an error. MessageDistribution is an enum of the type MessageDistributionPolicy and defaults to MessageDistributionPolicy.OneSubscriber—that is, only one of the subscribing services gets the message. Setting it to MessageDistributionPolicy.AllSubscribers delivers the message to all of the services.

Managing the Router Policy

You can use the RouterManagementClient class, shown here, to administer your router policies:

```
public static class RouterManagementClient
{
    public static RouterClient CreateRouter(
        TransportClientEndpointBehavior credential,
        Uri routerUri,RouterPolicy policy);

    public static void DeleteRouter(
        TransportClientEndpointBehavior credential,
        Uri routerUri);

    public static RouterClient GetRouter(
        TransportClientEndpointBehavior credential,
        Uri routerUri);

    public static RouterPolicy GetRouterPolicy(
        TransportClientEndpointBehavior credential,
        Uri routerUri);

    public static DateTime RenewRouter(
        TransportClientEndpointBehavior credential,
        Uri routerUri,TimeSpan requestedExpiration);
}
```

RouterManagementClient is a static class, and all its methods require the credential object of the type TransportClientEndpointBehavior, discussed in my previous column (see msdn.microsoft.com/magazine/dd942847.aspx). The following code demonstrates creating a simple router and a policy:

```
Uri routerAddress =
      new Uri(@"sb://MySolution.servicebus.windows.net/MyRouter/");

TransportClientEndpointBehavior credential = ...;

RouterPolicy policy = new RouterPolicy();

policy.ExpirationInstant = DateTime.UtcNow + TimeSpan.FromMinutes(5);
policy.MaxSubscribers = 4;
policy.MessageDistribution = MessageDistributionPolicy.AllSubscribers;

RouterManagementClient.CreateRouter(credential,routerAddress,policy);
```

In the example, you instantiate a router policy object and set the policy to some values, such as distributing the message to all subscribing services and limiting the router to no more than four subscribers. The router is configured to have a short life of only five minutes. All it takes to install the router is to call the CreateRouter method of RouterManagementClient with the policy and some valid credentials.

As an alternative to programmatic calls, you can use my Services Bus Explorer to view and modify routers. You can create a new router by specifying its address and various policy properties. In much the same way, you can delete all routers in the solution.

You can also review and modify the policies of existing routers by selecting them in the solution tree and interact with their properties in the right pane, as shown in **Figure 7**.

All the client needs to do to post messages to a router is to create a proxy whose address is the router's address and call the proxy.

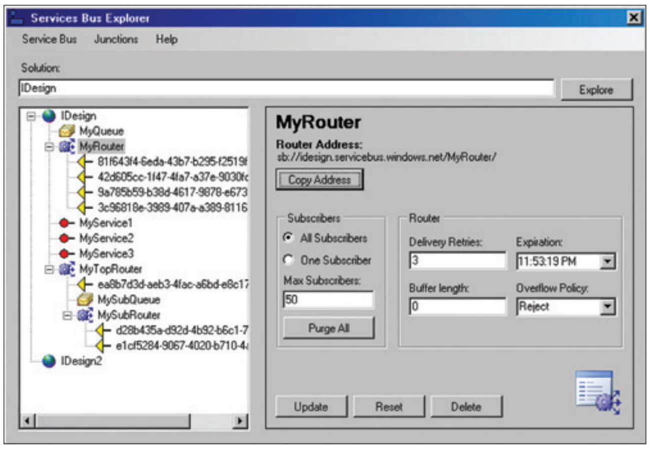
Subscribing to a Router

For a service to receive messages from a router, it must first subscribe to the router by adding to the host an endpoint whose address is the router's address. One way of doing this is by using the helper class RouterClient, defined as follows:

```
public sealed class RouterClient
{
    public ServiceEndpoint AddRouterServiceEndpoint<T>(
                               ServiceHost serviceHost);

    //More members
}
```

The methods of RouterManagementClient that create or get a router return an instance of RouterClient. You need to call the AddRouterServiceEndpoint<T> method with an instance of your service host. You can also simply add an endpoint to the host (either programmatically or in a config file) whose address is the router's address. Here's an example:

```
<service name = "MyService">
   <endpoint
      address  = "sb://MySolution.servicebus.windows.net/MyRouter/"
      binding  = "netOnewayRelayBinding"
      contract = "..."
   />
</service>
```



Solution: IDesign

```
IDesign
  MyQueue
  MyRouter
    81f643f4-6eda-43b7-b295-f2519f
    42d605cc-1f47-4fa7-a37e-9030fc
    9a785b59-b38d-4617-9878-e673
    3c96818e-3989-407a-a389-8116
  MyService1
  MyService2
  MyService3
  MyTopRouter
    ea8b7d3d-aeb3-4fac-a6bd-e8c17
    MySubQueue
    MySubRouter
      d28b435a-d92d-4b92-b6c1-7
      e1cf5284-9067-4020-b710-4
IDesign2
```

MyRouter

Router Address:
sb://idesign.servicebus.windows.net/MyRouter/

Subscribers: (•) All Subscribers ( ) One Subscriber
Max Subscribers: 50

Router
Delivery Retries: 3
Expiration: 11:53:19 PM
Buffer length: 0
Overflow Policy: Reject

Figure 7 **A Router in the Services Bus Explorer**

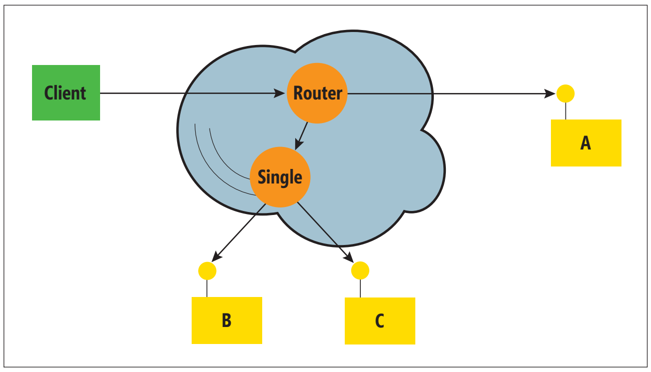


Figure 8 **Subscribing Router to Router**

Using Routers

Routers have three uses. The first is to broadcast the same message to multiple services. The classic case is, of course, publishing events to subscribing services, treating the service bus as an events-hub middleware. As mentioned in my column in the April issue (see msdn.microsoft.com/magazine/dd569756.aspx), there is more to event publishing than meets the eye, so I'll defer discussing events to the end of this column.

The second use for a router is as a load balancer among services. If you configure the router policy to distribute messages to only a single subscriber, the router in effect acts as a load balancer among the subscribing services. All load balancers follow some algorithm in deciding which service will handle the next message. Common algorithms include round robin, random, some fair scoreboard and queuing. My testing indicates that the service bus uses a pseudo–round robin—that is, it was a fair round robin some 95 percent of the time. To streamline creating a load-balancing router, my helper class ServiceBusHelper offers a number of overloaded CreateLoadBalancingRouter methods, shown here:

```
public static partial class ServiceBusHelper
{
   //Uses CardSpace for credential
   public static void CreateLoadBalancingRouter(string balancerAddress);


   /* Additional CreateLoadBalancingRouter(...)
      with different credentials */
}
```

CreateLoadBalancingRouter creates a router with a policy that distributes the messages to a single subscriber. It also detects whether the router exists before creating it and automatically renews the router's lease. The implementation of CreateLoadBalancing Router is included in the article's code download.

All the public CreateLoadBalancingRouter methods take different credentials types (the implementation I provide uses CardSpace), construct an instance of TransportClientEndpointBehavior, and call an internal overloaded CreateLoadBalancingRouter. That overloaded version creates a load-balancing policy that also publishes the router to the ATOM feed and calls CreateRouter with the specified policy and credentials. CreateRouter first uses the RouterExists method to check whether the router has already been created. Unfortunately, the only way to check is to see whether an error occurs when you try to obtain the router's policy.

If the router exists, CreateRouter returns its corresponding RouterClient. If the router does not exist, CreateRouter first creates a timer to monitor the expiration of the lease, using a Lambda expression as a sponsor. As the lease time approaches its end, the timer calls the Lambda expression to renew the lease. CreateRouter then creates the router.

Routers to Routers

The third use for routers is to route messages to other junctions, such as other routers (or queues). To subscribe one router to another, use the SubscribeToRouter method of RouterClient, shown here:

```
public sealed class RouterClient
{
    public RouterSubscriptionClient SubscribeToRouter(
      RouterClient routerClient,
      TimeSpan requestedTimeout);

    //More members
}

[Serializable]
public class RouterSubscriptionClient
{
   public DateTime Expires {get;}
   public DateTime Renew(TimeSpan requestedExpiration,
                         TransportClientEndpointBehavior credential);
   public void Unsubscribe(TransportClientEndpointBehavior credential);
   //More members
}
```

SubscribeToRouter accepts the router client of the router you want to subscribe to and returns an instance of RouterSubscriptionClient. The main uses of RouterSubscriptionClient is to unsubscribe and to renew the subscription lease.

For example, suppose you have service A, service B, and service C. You require that every client call always goes to service A and either service B or service c. (Imagine that you need to load balance service B and service C but also keep a record of all calls in the logbook service A). The router policies described so far, of either delivering to all subscribers or to a single subscriber, are inadequate. However, you can satisfy the requirements easily by using two routers, as shown in **Figure 8**.

The client delivers the messages to a top-level router with a policy of distribution to all services. Service A subscribes to the top-level router. A subrouter with a distribution policy of a single subscriber also subscribes to the top-level router. Both service B and service C subscribe to that subrouter. **Figure 9** shows the required routers configuration and how to subscribe the subrouter to the top router.

When it comes to a service subscribing to a subscribing router, there is one important detail regarding the subscription that differs from subscribing to a top-level router. The service endpoint must indicate that it is willing to receive messages from the subscribing router, even though the messages are addressed to the top-level router. This indication is made by setting the listening URI property of the endpoint to the subscribing service, while having the service endpoint itself address the top-level router. Using the addresses in **Figure 9**, both service B and service C would have to define their endpoints as follows:

```
<service name = "MyService">
   <endpoint listenUri =
     "sb://MySolution.servicebus.windows.net/MySubRouter/"
      address  = "sb://MySolution.servicebus.windows.net/MyTopRouter/"
      binding  = "netOnewayRelayBinding"
      contract = "..."
   />
</service>
```

You can automate this setting with the following service host extension:

```
public static partial class ServiceBusHelper
{
    public static void SubscribeToRouter(this ServiceHost host,
                                      string subRouterAddress)
    {
        for(int index = 0;
          index < host.Description.Endpoints.Count;index++)
        {
          host.Description.Endpoints[index].ListenUri =
                                  new Uri(subRouterAddress);
        }
    }

    //More members
}
```

Given that a router can subscribe to multiple routers, the services that subscribe to the subscribing router should be very mindful of which address—that is, which top-level router—they want to receive messages from.

Note that removing the top-level router or terminating the subscription cuts off the subrouters from client messages. This, in turn, is a feature of sorts for solution administrators, who can use this step to effectively shut off services without closing down their hosts.

Using Routers for Events

As I mentioned previously, the messaging junction in the service bus can act as a crude events hub and broadcast the messages to subscribing services. You need to install a router policy that maximizes the number of subscribers and notifies all subscribers. The shortcomings of doing so are similar to using the events binding (discussed in my column in April at msdn.microsoft.com/magazine/dd569756.aspx). There is no per-operation subscription, and consequently the subscribers (all of them) always receive all events, even events they do not care about (those that simply have a matching endpoint).

The solution is not to create a single router to handle all events but to create a router per event. Services that are interested in just that event will subscribe to that event's particular router. As with the events binding, having a router per event is accomplished by having each subscriber manage a host instance per operation, because a single host that monitors all routers will do nothing to filter the events. The subscriber needs to open and close the respective host to subscribe or unsubscribe to the event. This requires tedious and repetitive code.

The good news is that the helper class EventsRelayHost that I presented in my previous column is already doing just that, and with a little refactoring of a common base class called EventsHost, it can be adapted for use with routers rather than the events relay binding. (For routers, you need just the one-way relay binding.)

First, I defined the abstract base class EventsHost, shown here:

```
//Partial listing
public abstract class EventsHost
{
   public EventsHost(Type serviceType,string baseAddress);
   public EventsHost(Type serviceType,string[] baseAddresses);

   public abstract void SetBinding(NetOnewayRelayBinding binding);
   public abstract void SetBinding(string bindingConfigName)
   protected abstract NetOnewayRelayBinding GetBinding();

   public void Subscribe();
   public void Subscribe(Type contractType);
   public void Subscribe(Type contractType,string operation)

   public void Unsubscribe();
   public void Unsubscribe(Type contractType);
   public void Unsubscribe(Type contractType,string operation);
}
```

The two subclasses, EventsRelayHost and EventsRouterHost, are defined in **Figure 10**.The implementation of EventsHost and EventsRelayHost is presented in my previous column.

Figure 9 **Subscribing Router to Router**

```
Uri topRouterAddress =
    new Uri(@"sb://MySolution.servicebus.windows.net/MyTopRouter/");
Uri subRouterAddress =
    new Uri(@"sb://MySolution.servicebus.windows.net/MySubRouter/");

TransportClientEndpointBehavior credential = ...;

RouterPolicy topRouterPolicy = new RouterPolicy();
subRouterPolicy.MaxSubscribers = 20;
topRouterPolicy.MessageDistribution =
                             MessageDistributionPolicy.AllSubscribers;
RouterClient topRouterClient = RouterManagementClient.CreateRouter(
                       credential,topRouterAddress,topRouterPolicy);
```

```
RouterPolicy subRouterPolicy = new RouterPolicy();
subRouterPolicy.MaxSubscribers = 30;

subRouterPolicy.MessageDistribution =
                        MessageDistributionPolicy.OneSubscriber;
RouterClient subRouterClient = RouterManagementClient.CreateRouter(
                       credential,subRouterAddress,subRouterPolicy);
RouterSubscriptionClient subscription = subRouterClient.
                                SubscribeToRouter(topRouterClient,
                                TimeSpan.MaxValue);

//Sometime later
subscription.Unsubscribe(credential);
```

Figure 10 **Defining EventsRelayHost**

```
//For use with the events binding, see previous column
public class EventsRelayHost : EventsHost
{...}

public class EventsRouterHost : EventsHost
{
   public override void SetBinding(NetOnewayRelayBinding binding)
   {
      RelayBinding = binding;
   }
   public override void SetBinding(string bindingConfigName)
   {
      SetBinding(new NetOnewayRelayBinding(bindingConfigName));
   }
   protected override NetOnewayRelayBinding GetBinding()
   {
      return RelayBinding ?? new NetOnewayRelayBinding();
   }
}
```

EventsRouterHost is similar to EventsRelayHost—both manage a host per event, and it matters not if the address they monitor is the address of an events endpoint or that of a router. The only difference is that the EventsRouterHost must use the one-way relay binding to receive the events, as opposed to the events binding used by EventsRelayHost. Other than that, EventsRouterHost and EventsRelayHost should be identical. This is reflected in the fact that both derive from EventsHost, which does the heavy lifting of managing the hosts. Both EventsRouterHost and EventsRelayHost merely override the binding management methods.

Other than that, using EventsRouterHost is just like using EventsRelayHost, where Subscribe and Unsubscribe function instead of opening and closing the host:

```
EventsHost host = new EventsRouterHost(typeof(MyService),
                  "sb://MySolution.servicebus.windows.net/...");

host.Subscribe();
...
host.Unsubscribe(typeof(IMyEvents),"OnEvent2");
```

The previous code opens and closes a respective host internally that monitors just that event.

Creating Event Routers

EventsRouterHost requires the solution administrator to configure the routers beforehand, and it will not try to create the routers. I made that choice deliberately to separate configuring the routers and their policies from the subscribers. All the client sees is the junction's address, and to fire an event over a one-way relay binding, you can use the same EventRelayClientBase presented in my previous column to publish the events.

To streamline the effort of creating the event routers, you can use my ServiceBusHelper to create the routers:

```
public static partial class ServiceBusHelper
{
   //Uses CardSpace for credential
   public static RouterClient[] CreateEventRouters(string baseAddress,
                                                    Type contractType);

   /* Additional CreateEventRouters(...) with different credentials */
}
```

CreateEventRouters accepts the router base address. It appends to the base address the name of the contract and the operation name and opens an individual router under that address. For example, given the following base address:

```
sb://MySolution.servicebus.windows.net/
```

and this contract definition:

```
[ServiceContract]
interface IMyEvents
{
   [OperationContract(IsOneWay = true)]
   void OnEvent1();

   [OperationContract(IsOneWay = true)]
   void OnEvent2(int number);

   [OperationContract(IsOneWay = true)]
   void OnEvent3(int number,string text);
}
```

CreateEventRouters creates the following routers:

```
sb://MySolution.servicebus.windows.net/IMyEvents/OnEvent1/
sb://MySolution.servicebus.windows.net/IMyEvents/OnEvent2/
sb://MySolution.servicebus.windows.net/IMyEvents/OnEvent3/
```

This is exactly what EventsRouterHost expects. This article's code download provides a partial listing of the implementation of ServiceBusHelper.CreateEventRouters, without error handling and a few of the security overloaded methods.

The public CreateEventRouters that takes no credentials defaults to using CardSpace by creating a TransportClientEndpointBehavior object and passing it to the internal method. That method uses reflection to obtain the collection of operations on the service contract type. For each operation, it calls the CreateEventRouter method, passing it the address for the router and the policy. The policy for each router maxes out the number of subscribers, delivers the events to all of them and publishes the router to the ATOM feed.

Routers vs. the Events Binding

For most cases of cloud-based, publish-subscribe solutions, I would recommend the events binding. First, I find the limit on the maximum number of subscribers to be a serious handicap. It is possible that 50 will be more than adequate for your application. However, what if over time you need to support more subscribers? No such limit is imposed on the events binding. Other issues with the routers approach are the need to create the routers beforehand and to worry about lease sponsorship. The events binding, on the other hand, functions as a crude ad-hoc router without these liabilities.

There are two saving graces for router-based events management. The first is the option for the system administrators to use tools such as the Services Bus Explorer to manage the subscribers, even indirectly by managing the routers. The second, and more ubstantial, is the ability to combine routers with queues to create queued publishers and queues subscribers, the approach I will describe in my next column. ■

JUVAL LOWY *is a software architect with IDesign providing WCF training and architecture consulting. His recent book is "Programming WCF Services, 2nd Edition" (O'Reilly, 2008). He is also the Microsoft regional director for the Silicon Valley. Contact Juval at idesign.net.*

RICK MOLLOY

Four Ways to Use the Concurrency Runtime in Your C++ Projects

I'm often asked how to integrate the new parallel computing libraries in the Visual Studio 2010 Beta into existing C++ projects. In this column, I'll explain a few of the ways you can use the APIs and classes that are part of the Parallel Pattern Library (PPL), Asynchronous Agents Library, and Concurrency Runtime in your existing projects. I'll walk through four common scenarios developers face in multithreaded application development and describe how you can be productive right away, using the PPL and Agents Library to make your multithreaded program more efficient and more scalable.

One: Moving Work from a UI Thread to a Background Task

One of the first things you're told to avoid as a Windows developer is hanging the UI thread. The end-user's experience of an unresponsive window is incredibly unpleasant regardless of whether developers provide their customers with a waiting mouse pointer or Windows provides them with a hung UI in the form of a frosted-glass window. The guidance we're given is often rather terse: don't perform any blocking calls on the UI thread but instead move these calls to a background thread. My experience is that this guidance isn't sufficient and that the coding work involved is tedious, error-prone, and rather unpleasant.

In this scenario, I'll provide a simple serial example that shows how to move work by using the existing threading APIs. Then, I'll provide two approaches for moving work to a background thread using the PPL and Agents Library. I'll wrap up this scenario by tying the examples back to the specifics of a UI thread.

Move a Long-Running Serial Operation to a Background Thread

So what does it mean to move work to a background thread? If I have some function that is running a long or potentially blocking operation and I want to move that function to a background thread, a good deal of boilerplate code is involved in the mechanics of actually moving that work, even for something as simple as a single function call such as the one shown here:

```
void SomeFunction(int x, int y){
        LongRunningOperation(x, y);
}
```

First, you need to package up any state that's going to be used. Here I'm just packaging up a pair of integers, so I could use a built-in container like a std::vector, a std::pair, or a std::tuple, but more typically what I've seen folks do is package up the values in their own struct or class, like this:

```
struct LongRunningOperationParams{
        LongRunningOperationParams(int x_, int y_):x(x_),y(y_){}
        int x;
        int y;
}
```

Then you need to create a global or static function that matches the threadpool or CreateThread signature, unpackages that state (typically by dereferencing a void * pointer), executes the function, and then deletes the data if appropriate. Here's an example:

```
DWORD WINAPI LongRunningOperationThreadFunc(void* data){
         LongRunningOperationParams* pData =
            (LongRunningOperationParams*) data;
         LongRunningOperation(pData->x,pData->y);
         //delete the data if appropriate
         delete pData;
}
```

Now you can finally get around to actually scheduling the thread with the data, which looks like this:

```
void SomeFunction(int x, int y){
        //package up our thread state
        //and store it on the heap
        LongRunningOperationParams* threadData =
        new  LongRunningOperationParams(x,y);
        //now schedule the thread with the data
        CreateThread(NULL,NULL,&LongRunningOperationThreadFunc,
          (void*) pData,NULL);
}
```

This might not seem like that much more code. Technically, I've added only two lines of code to SomeFunction, four lines for our class and three lines for the thread function. But that's actually four times as much code. We went from three lines of code to 12 lines of code just to schedule a single function call with two parameters. The last time I had to do something like this, I believe I had to capture approximately eight variables, and capturing and setting all this state becomes quite tedious and prone to error. If I recall correctly, I found and fixed at least two bugs just in the process of capturing the state and building the constructor.

I also haven't touched on what it takes to wait for the thread to complete, which typically involves creating an event and a call to

This article is based on a prerelease version of Visual Studio 2010. All information is subject to change.

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

Code download available at code.msdn.microsoft.com/mag200910Concurrent.

WaitForSingleObject to track that handle and, of course, cleaning up the handle when you're done with it. That's at least three more lines of code, and that still leaves out handling exceptions and return codes.

An Alternative to CreateThread: The task_group Class

The first approach I'm going to describe is using the task_group class from the PPL. If you're not familiar with the task_group class, it provides methods for spawning tasks asynchronously via task_group::run and waiting for its tasks to complete via task_group::wait. It also provides cancellation of tasks that haven't been started yet and includes facilities for packaging an exception with std::exception_ptr and rethrowing it.

You'll see that significantly less code is involved here than with the CreateThread approach and that from a readability perspective, the code is much closer to the serial example. The first step is to create a task_group object. This object needs to be stored somewhere where its lifetime can be managed—for example, on the heap or as a member variable in a class. Next you use task_group::run to schedule a task (not a thread) to do the work. Task_group::run takes a functor as a parameter and manages the lifetime of that functor for you. By using a C++0x lambda to package up the state, this is effectively a two-line change to the program. Here's what the code looks like:

```
//a task_group member variable
task_group backgroundTasks;
void SomeFunction(int x, int y){
        backgroundTasks.run([x,y](){LongRunningOperation(x, y);});
}
```

Making Work Asynchronous with the Agents Library

Another alternative is to use the Agents Library, which involves an approach based on message passing. The amount of code change is about the same, but there's a key semantic difference worth pointing out with an agent-based approach. Rather than scheduling a task, you build a message-passing pipeline and asynchronously send a message containing just the data, relying on the pipeline itself to process the message. In the previous case, I'd send a message containing *x* and *y*. The work still happens on another thread, but subsequent calls to the same pipeline are queued, and the messages are processed in order (in contrast to a task_group, which doesn't provide ordering guarantees).

First, you need a structure to contain the message. You could, in fact, use the same structure as the earlier one, but I'll rename it as shown here:

```
struct LongRunningOperationMsg{
        LongRunningOperationMsg (int x, int y):m_x(x),m_y(y){}
        int m_x;
        int m_y;
}
```

The next step is to declare a place to send the message to. In the Agents Library, a message can be sent to any message interface that is a "target," but in this particular case the most suitable is call<T>. A call<T> takes a message and is constructed with a functor that takes the message as a parameter. The declaration and construction of the call might look like this (using lambdas):

```
call<LongRunningOperationMsg>* LongRunningOperationCall = new
   call<LongRunningOperationMsg>([]( LongRunningOperationMsg msg)
{
LongRunningOperation(msg.x, msg.y);
})
```

The modification to SomeFunction is now slight. The goal is to construct a message and send it to the call object asynchronously. The call will be invoked on a separate thread when the message is received:

```
void SomeFunction(int x, int y){
        asend(LongRunningOperationCall, LongRunningOperationMsg(x,y));
}
```

Getting Work Back onto the UI Thread

Getting work off the UI thread is only half the problem. Presumably at the end of LongRunningOperation, you're going to get some meaningful result, and the next step is often getting work back onto the UI thread. The approach to take varies based on your application, but the easiest way to achieve this in the libraries offered in Visual Studio 2010 is to use another pair of APIs and message blocks from the Agents Library: try_receive and unbounded_buffer<T>.

An unbounded_buffer<T> can be used to store a message containing the data and potentially the code that needs to be run on the UI thread. Try_receive is a nonblocking API call that can be used to query whether there is data to display.

For example, if you were rendering images on your UI thread, you could use code like the following to get data back onto the UI thread after making a call to InvalidateRect:

```
unbounded_buffer<ImageClass>* ImageBuffer;
LONG APIENTRY MainWndProc(HWND hwnd, UINT uMsg,
  WPARAM wParam, LPARAM lParam)
{
    RECT rcClient;
    int i;
    ...
   ImageClass image;
   //check the buffer for images and if there is one there, display it.
   if (try_receive(ImageBuffer,image))
      DisplayImage(image);
   ...
}
```

Some details, like the implementation of the message loop, have been omitted here, but I hope this section was instructive enough to demonstrate the technique. I encourage you to check the sam-

Figure 1 **A Non-Thread-Safe Class**

```
class Widget{
    size_t m_width;
    size_t m_height;
public:
    Widget(size_t w, size_t h):m_width(w),m_height(h){};
    size_t GetWidth(){
        return m_width;
    }
    size_t GetHeight(){
        return m_height;
    }
    void SetWidth(size_t width){
        m_width = width;
    }
    void SetHeight(size_t height){
        m_height = height;
    }
};
```

ple code for the article, which has a full working example of each of these approaches.

Two: Managing Shared State with Message Blocks and Agents

Another common situation in multithreaded application development is managing shared state. Specifically, as soon as you try to communicate or share data between threads, managing shared state quickly becomes a problem you need to deal with. The approach I've often seen is to simply add a critical section to an object to protect its data members and public interfaces, but this soon becomes a maintenance problem and sometimes can become a performance problem as well. In this scenario, I'll walk through a serial and naïve example using locks, and then I'll show an alternative using the message blocks from the Agents Library.

Locking a Simple Widget Class

Figure 1 shows a non-thread-safe Widget class with width and height data members and simple methods that mutate its state.

The naïve approach to making the Widget class thread safe is to protect its methods with a critical section or reader-writer lock. The PPL contains a reader_writer_lock, and **Figure 2** offers a first look at the obvious solution to the naïve approach: using the reader_writer_lock in the PPL.

What I've done here is add a read_writer_lock as a member variable and then decorate all appropriate methods with either the reader or the writer version of the lock. I'm also using scoped_lock objects to ensure that the lock isn't left held in the midst of an exception. All the Get methods now acquire the reader lock, and the Set methods acquire the write lock. Technically, this approach looks like it is correct, but the design is actually incorrect and is fragile overall because its interfaces, when combined, are not thread safe. Specifically, if I have the following code, I'm likely to have corrupt state:

```
Thread1{
        SharedWidget.GetWidth();
        SharedWidget.GetHeight();
}
Thread2{
        SharedWidget.SetWidth();
        SharedWidget.SetHeight();
}
```

Because the calls on Thread1 and Thread2 can be interleaved, Thread1 can acquire the read lock for GetWidth, and then before GetHeight is called, SetWidth and SetHeight could both execute. So, in addition to protecting the data, you have to ensure that the interfaces to that data are also correct; this is one of the most insidious kinds of race conditions because the code looks correct and the errors are very difficult to track down. Naïve solutions I've seen for this situation often involve introducing a lock method on the object itself—or worse, a lock stored somewhere else that developers need to remember to acquire when accessing that widget. Sometimes both approaches are used.

An easier approach is to ensure that interfaces can be interleaved safely without exposing this ability to tear the state of the object between interleaved calls. You might decide to evolve your interface as shown in **Figure 3** to provide GetDimensions and

Figure 2 **Using the reader_writer_lock from the Parallel Pattern Library**

```
class LockedWidget{
    size_t m_width;
    size_t m_height;
    reader_writer_lock lock;
public:
    LockedWidget (size_t w, size_t h):m_width(w),m_height(h){};
    size_t GetWidth(){
        auto lockGuard = reader_writer::scoped_lock_read(lock);
        return m_width;
    }
    size_t GetHeight(){
        auto lockGuard = reader_writer::scoped_lock_read(lock);
        return m_height;
    }
    void SetWidth(size_t width){
        auto lockGuard = reader_writer::scoped_lock(lock);
        m_width = width;
    }
    void SetHeight(size_t height){
        auto lockGuard = reader_writer::scoped_lock(lock)
        m_height = height;
    }
};
```

Figure 3 **A Version of the Interface with GetDimensions and UpdateDimensions Methods**

```
struct WidgetDimensions
{
    size_t width;
           size_t height;
};
class LockedWidgetEx{
    WidgetDimensions m_dimensions;
    reader_writer_lock lock;
public:
    LockedWidgetEx(size_t w, size_t h):
      m_dimensions.width(w),m_dimensions.height(h){};
    WidgetDimensions GetDimensions(){
        auto lockGuard = reader_writer::scoped_lock_read(lock);
        return m_dimensions;
    }
    void UpdateDimensions(size_t width, size_t height){
        auto lockGuard = reader_writer::scoped_lock(lock);
        m_dimensions.width = width;
        m_dimensions.height = height;
    }
};
```

UpdateDimensions methods. This interface is now less likely to cause surprising behavior because the methods don't allow exposing unsafe interleavings.

Managing Shared State with Message Blocks

Now let's take a look at how the Agents Library can help make managing shared state easier and the code a little more robust. The key classes from the Agents Library that are useful for managing shared variables are overwrite_buffer<T>, which stores a single updatable value and returns a copy of the latest value when receive is called; single_assignment<T>, which stores and returns a copy of a single value when receive is called but, like a constant, can be assigned only once; and unbounded_buffer<T>, which stores an unlimited number of items (memory permitting) and, like a FIFO queue, dequeues the oldest item when receive is called.

I'll start by using an overwrite_buffer<T>. In the Widget class, I'll first replace the m_dimensions member variable with overwrite_ buffer<WidgetDimensions>, and then I'll remove the explicit locks from the methods and replace them with the appropriate send and receive calls. I still need to worry about our interface being safe, but I no longer have to remember to lock the data. Here's how this looks in code. It's actually slightly fewer lines of code than the locked version and the same number of lines as the serial version:

```
class AgentsWidget{
    overwrite_buffer<WidgetDimensions> m_dimensionBuf;
public:
    AgentsWidget(size_t w, size_t h){
        send(&m_dimensionBuf,WidgetDimensions(w,h));
    };
    WidgetDimensions GetDimensions(){
        return receive(&m_dimensionBuf);
    }
    void UpdateDimensions(size_t width, size_t height){
        send(&m_dimensionBuf,WidgetDimensions(w,h));
    }
};
```

There's a subtle semantic difference here from the reader_writer lock implementation. The overwrite_buffer allows a call to UpdateDimensions to occur during a call to Dimensions. This allows practically no blocking during these calls, but a call to GetDimensions may be slightly out of date. It's worth pointing out that the problem existed in the locked version as well, because as soon as you get the dimensions, they have the potential to be out of date. All I've done here is remove the blocking call.

An unbounded_buffer can also be useful for the Widget class. Imagine that the subtle semantic difference I just described was incredibly important. For example, if you have an instance of an object that you want to ensure is accessed by only one thread at a time, you can use unbounded_buffer as an object holder that manages access to that object. To apply this to the Widget class, you can remove m_dimensions and replace it with unbounded_buffer<WidgetDimension> and use this buffer via the calls to GetDimensions and UpdateDimensions. The challenge here is to ensure that no one can get a value from our widget while it is being updated. This is achieved by emptying the unbounded_buffer so that calls to GetDimension will block waiting for the update to occur. You can see this in **Figure 4**. Both GetDimensions and UpdateDimensions block, waiting for exclusive access to the dimensions variable.

Figure 4 **Emptying the Unbounded_Buffer**

```
class AgentsWidget2{
    unbounded_buffer<WidgetDimensions> m_dimensionBuf;
public:
    AgentsWidget2(size_t w, size_t h){
        send(&m_dimensionBuf,WidgetDimensions(w,h));
    };
    WidgetDimensions GetDimensions(){
        //get the current value
        WidgetDimensions value = receive(&m_dimensionBuf);

        //and put a copy of it right back in the unbounded_buffer
        send(&m_dimensionBuf,value);

        //return a copy of the value
        return WidgetDimensions(value);
    }
    void  UpdateDimensions (size_t width, size_t height){

        WidgetDimensions oldValue = receive(&m_dimensionBuf);

        send(&m_dimensionBuf,WidgetDimensions(width,height));
    }
};
```

It's Really About Coordinating Access to the Data

I want to stress one more thing about our Widget class: ensuring that methods and data that can be accessed concurrently work "safely" together is critical. Often, this can be achieved by coordinating access to state rather than by locking methods or objects. From a pure "lines of code" perspective, you won't see a big win over the locked example, and, in particular, the second example might even have a little more code. What is gained, however, is a safer design, and with a little thought, you can often modify serial interfaces so that the internal state of the object isn't "torn." In the Widget example, I did this by using message blocks, and I was able to protect that state in such a way that it is safer. Adding methods or functionality to the Widget class in the future is less likely to destroy the internal synchronization we've set up. With a member lock, it's pretty easy to simply forget to lock the lock when a method is added on a class. But moving operations to a message-passing model and using message blocks such as the overwrite buffer in their natural way can often keep data and classes synchronized.

Three: Using Combinable for Thread Local Accumulations and Initialization

The second scenario, in which we protected access to an object with locks or message blocks, works very well for heavier weight objects that are accessed infrequently. If while reading that example you thought that there might be a performance problem if the synchronized widget were used in a tight (and parallel) loop, you're probably right. That's because protecting shared state can be problematic, and for completely general purpose algorithms and objects that truly share state, there unfortunately aren't a lot of options other than to coordinate access or introduce a lock. But you can almost always find a way to refactor code or an algorithm to relax the dependency on shared state, and once you've done this, a few specific but common patterns in which an object calls combinable<T> in the Parallel Pattern Library can really help out.

Combinable<T> is a concurrent container that offers support for three broad use cases: holding a thread-local variable or performing thread-local initialization, performing associative binary operations (like sum, min, and mix) on the thread-local variables and combining them, and visiting each thread-local copy with an operation (like splicing lists together). In this section, I'll explain each of these cases and provide examples of how to use them.

Holding a Thread-Local Variable or Performing Thread-Local Initialization

The first use case I mentioned for combinable<T> was for holding a thread-local variable. It is relatively common to store a thread-local copy of global state. For example, in the colorized ray tracers applications like the one in our sample pack (code.msdn.microsoft.com/concrtextras) or in the samples for parallel development with .NET

4.0 (code.msdn.microsoft.com/ParExtSamples) there is an option to colorize each row by thread to visualize the parallelism. In the native version of the demo, this is done by using a combinable object that holds the thread-local color.

You can hold a thread-local variable, of course, by using thread-local storage (TLS), but there are some disadvantages—most notably lifetime management and visibility, and these go hand in hand. To use TLS, you first need to allocate an index with TlsAlloc, allocate your object, and then store a pointer to your object in the index with TlsSetValue. Then, when your thread is exiting, you need to ensure that your object is deallocated. (TlsFree is called automatically.) Doing this once or twice per thread and ensuring that there aren't any leaks because of early exits or exceptions isn't that challenging, but if your application needs dozens or hundreds of these items, a different approach is likely better.

Combinable<T> can be used to hold a thread-local value as well, but the lifetimes of the individual objects are tied to the lifetime of the combinable<T> item, and much of the initialization is automated. You access the thread-local value simply by calling the combinable::local method, which returns a reference to the local object. Here's an example using task_group, but this can be done with Win32 threads as well:

```
combinable<int> values;

auto task = [&](){
        values.local() = GetCurrentThreadId();
        printf("hello from thread: %d\n",values.local());
};

task_group tasks;

tasks.run(task);

//run a copy of the task on the main thread
task();

tasks.wait();
```

I mentioned that thread-local initialization can also be achieved with combinable. If, for example, you need to initialize a library call on each thread on which it is used, you can create a class that performs the initialization in its constructor. Then, on the first use per thread, the library call will be made, but it will be skipped on subsequent uses. Here's an example:

```
class ThreadInitializationClass
{
public:
      ThreadInitializationClass(){
            ThreadInitializationRoutine();
      };
};

...
//a combinable object will initialize these
combinable<ThreadInitializationClass> libraryInitializationToken;

...
//initialize the library if it hasn't been already on this thread
ThreadInitializationClass& threadInit = libraryInitalizationToken.local();
```

Performing Reductions in a Parallel Loop

Another major scenario for the combinable object is to perform thread-local reductions, or thread-local accumulations. Specifically, you can avoid a particular type of race condition when parallelizing loops or in recursive parallel traversals with combinable. Here's an incredibly naïve example that's not intended to show speed-ups. The following code shows a simple loop that looks like it can be parallelized with parallel_for_each, except for access to the sum variable:

```
int sum = 0;
for (vector<int>::iterator it = myVec.begin(); it != myVec.end(); ++it)
{
    int element = *it;
    SomeFineGrainComputation(element);
    sum += element;
}
```

Now, rather than placing a lock in our parallel_for_each, which destroys any chance we had of speed-ups, we can use a combinable object to calculate the thread-local sums:

```
combinable<int> localSums;

parallel_for_each(myVec.begin(),  myVec.end(), [&localSums] (int element) {
   SomeFineGrainComputation(element);
   localSums.local() += element;
});
```

We've now successfully avoided the race condition, but we have a collection of thread-local sums stored in the localSums object, and we still need to extract the final value. We can do this with the combine method, which takes a binary functor like the following:

```
int sum = localSums.combine(std::plus<int>);
```

The third use case for combinable<T>, which involves using the combine_each method, is when you need to visit each of the thread-local copies and perform some operation on them (like cleanup or error checking). Another, more interesting example is when your combinable object is a combinable<list<T>>, and in your threads you are building up std::lists or std::sets. In the case of std::lists, they can easily be spliced together with list::splice; with std::sets, they can be inserted with set::insert.

Four: Converting an Existing Background Thread to an Agent or a Task

Suppose you already have a background or worker thread in your application. There are some very good reasons why you might want to convert that background thread to a task from the PPL or to an agent, and doing so is relatively straightforward. Some of the major advantages of doing this include the following:

Composability and performance. If your worker threads are compute intensive and you are considering using additional threads in the PPL or Agents Library, converting your background thread to a worker task allows it to cooperate with the other tasks in the runtime and avoid oversubscription on the system.

Cancellation and exception handling. If you want to be able to easily cancel work on a thread or have a well-described mechanism for handling exceptions, a task_group has these capabilities built in.

Control flow and state management. If you need to manage the state of your thread (started or completed, for example) or have an object whose state is effectively inseparable from the worker thread, implementing an agent might be useful.

Task_group Offers Cancellation and Exception Handling

In the first scenario, we explored what it takes to schedule work with a task_group: essentially packaging your work into a func-

Figure 5 **Implementation of MyAgentClass**

```
class MyAgentClass : public agent{

public:
      MyAgentClass (){
      }
      AgentsWidget widget;
      void run(){
              //run is started asynchronously when agent::start is called

              //...

              //set status to complete
              agent::done();
      }
};
```

tor (using a lambda, an std::bind, or a custom function object) and scheduling it with the task_group::run method. What I didn't describe was the cancellation and exception-handling semantics, which are, in fact, related.

First, I'll explain the straightforward semantics. If your code makes a call to task_group::cancel or a task throws an uncaught exception, cancellation is in effect for that task_group. When cancellation is in effect, tasks that haven't been started on that task_group won't be started, which allows scheduled work to easily and quickly be canceled on a task_group. Cancellation doesn't interrupt tasks that are running or blocked, so a running task can query the cancellation status with the task_group::is_canceling method or by the helper function is_current_task_group_canceling. Here's a brief example:

```
task_group tasks;
tasks.run([](){
    ...
    if(is_current_task_group_canceling())
    {
        //cleanup then return
        ...
        return;
    }
});
tasks.cancel();
tasks.wait();
```

Exception handling impacts cancellation because an uncaught exception in a task_group triggers cancellation on that task_group. If there is an uncaught exception, the task_group will actually use std::exception_ptr to package up the exception on the thread it was thrown on. Later, when task_group::wait is called, the exception is rethrown on the thread that called wait.

Implementing an Asynchronous Agent

The Agents Library offers an alternative to using a task_group: replacing a thread with the agent base class. If your thread has a lot of thread-specific state and objects, an agent might be a better fit for the scenario. The abstract agent class is an implementation of the actor pattern; the intended usage is to implement your own class derived from agent and then encapsulate any state that your actor (or thread) may have into that agent. If there are fields that are intended to be publicly accessible, the guidance is to expose them as message blocks or sources and targets and use message passing to communicate with the agent.

Implementing an agent requires deriving a class from the agent base class and then overriding the virtual method run. The agent can then be started by calling agent::start, which spawns the run method as a task, much like a thread. The advantage is that thread-local state can now be stored in the class. This allows for easier synchronization of state between threads, particularly if the state is stored in a message block. **Figure 5** shows an example of an implementation that has a publicly exposed member variable of type AgentsWidget.

Note that I've set the agent's status to done as the run method is exiting. This allows the agent to not only be started but also be waited on. Furthermore, the agent's current status can be queried by a call to agent::status. Starting and waiting on our agent class is straightforward, as the following code shows:

```
MyAgentClass MyAgent;

//start the agent
MyAgent.start();

//do something else
...

//wait for the agent to finish
MyAgent.wait(&MyAgent);
```

Bonus Item: Sorting in Parallel with parallel_sort

Finally, I'd like to suggest another potentially easy point of parallelization, this time not from the PPL or the Agents Library but from our sample pack available at code.msdn.microsoft.com/concrtextras. Parallel quicksort is one of the examples we use for explaining how to parallelize recursive divide-and-conquer algorithms with tasks, and the sample pack contains an implementation of parallel quicksort. Parallel sort can show speed-ups if you're sorting a large number of items where the comparison operation is somewhat expensive, as with strings. It probably won't show speed-ups for small numbers of items or when sorting built-in types like integers and doubles. Here's an example of how it can be used:

```
//from the sample pack
#include "parallel_algorithms.h"
int main()

using namespace concurrency_extras;
{
        vector<string> strings;

        //populate the strings
        ...
        parallel_sort(strings.begin(),strings.end());
}
```

Wrapping Up

I hope this column helps expand the horizons of how the parallel libraries in Visual Studio 2010 will apply to your projects, beyond simply using parallel_for or tasks to speed up compute-intensive loops. You'll find many other instructive examples in our documentation on MSDN (msdn.microsoft.com/library/dd504870(VS.100).aspx) and in our sample pack (code.msdn.microsoft.com/concrtextras) that help illustrate the parallel libraries and how they can be used. I encourage you to check them out. ■

RICK MOLLOY *is a program manager on the Parallel Computing Platform team at Microsoft.*