THE MICROSOFT JOURNAL FOR DEVELOPERS

OCTOBER 15, 2012 VOL 27 NO 10A

msdn magazine

Special Issue

Windows 8

COLUMNS

Microsoft

infragistics.com/**EXPERIENCE**

INFRAGISTICS™
DESIGN / DEVELOP / EXPERIENCE

Infragistics Sales US 800 231 8588 • Europe +44 (0) 800 298 9055 • India +91 80 4151 8042 • APAC (+61) 3 9982 4545

msdn magazine

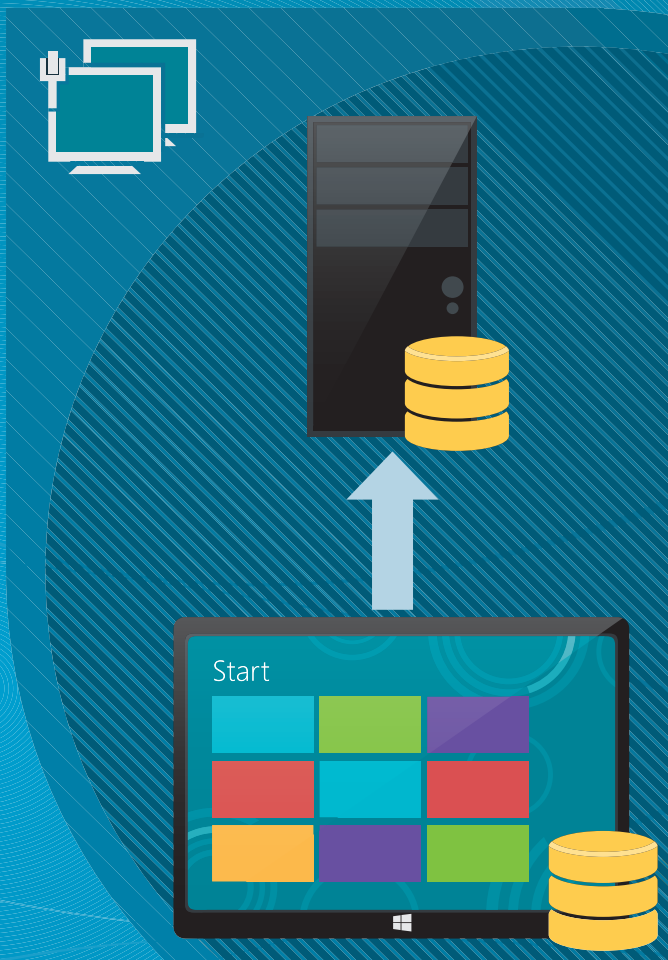
OCTOBER, 15 2012 VOLUME 27 NUMBER 10A
WINDOWS 8

MSDN Magazine (ISSN 1528-4859) is published 13 times a year, monthly with a special issue in October by 1105 Media, Inc., 9201 Oakdale Avenue, Ste. 101, Chatsworth, CA 91311. Periodicals postage paid at Chatsworth, CA 91311-9998, and at additional mailing offices. Annual subscription rates payable in US funds are: U.S. $35.00, International $60.00. Annual digital subscription rates payable in U.S. funds are: U.S. $25.00, International $25.00. Single copies/back issues: U.S. $10, all others $12. Send orders with payment to: *MSDN Magazine*, P.O. Box 3167, Carol Stream, IL 60132, email MSDNmag@1105service.com or call (847) 763-9560. **POSTMASTER:** Send address changes to *MSDN Magazine*, P.O. Box 2166, Skokie, IL 60076. Canada Publications Mail Agreement No: 40612608. Return Undeliverable Canadian Addresses to Circulation Dept. or XPO Returns: P.O. Box 201, Richmond Hill, ON L4B 4R5, Canada.

ADO.NET and LINQ Data Access for Metro-style Applications

LinqConnect for Metro is a fast and easy-to-use ORM solution for Metro style applications that allows connecting to **Oracle, MySQL,** and **PostgreSQL** database servers directly, and enables you to develop truly client-server data applications for Metro without the need to create a web service for accessing data. Our ORM also includes small and efficient, and at the same time powerful **SQLite database engine** for storing data in local databases.

Embedded database functionality together with ADO.NET and LINQ interfaces and direct database connectivity make LinqConnect a perfect choice to work with data in Metro-style applications.



Store Verification Passed

100% MANAGED

LinqConnect

LinqConnect is a fast, lightweight, and easy to use ORM solution, supporting SQL Server, Oracle, MySQL, PostgreSQL, and SQLite. It allows you to create efficient and powerful data access layer for your .NET Framework, Metro, Silverlight, or Windows Mobile 7 applications using Model-First, Database-First or mixed approaches.

devart

Editor's Note

MICHAEL DESMOND

All Eyes on Windows 8

Welcome to the Windows 8 special edition of *MSDN Magazine*. As many of you have no doubt noticed, we haven't had a lot to say about the newest Microsoft OS and the underlying Windows Runtime (WinRT). That's about to change—starting right now.

You're holding in your hands an unprecedented extra issue of *MSDN Magazine*, focused entirely on Windows 8, the Windows Runtime and the development of Windows Store apps. The Windows 8 special edition is packed with hands-on tutorials, from Jason Olson's informative, technical dive into the Windows Runtime and how it enables Windows Store app development in C++, C#/Visual Basic and JavaScript, to Christian Schormann's walk through using Expression Blend to create Windows Store apps built using HTML and XAML.

In between, you'll find great resources, such as David Tepper's exploration of memory management in the Windows Runtime, and Shawn Farkas' guide showing how managed .NET developers can call WinRT APIs from their Windows Store applications, and Diego Dagum's look at porting C++ MFC applications to the Windows Runtime.

There's more where that came from, including the debut of two new columns that will be appearing regularly in *MSDN Magazine* going forward. Rachel Appel's Modern Apps column works the expanding waterfront of Windows Store and rich client application development, kicking off with an informative look at the Windows Store app lifecycle and how it can be managed for optimal performance and efficiency. You might recognize Appel from her work as a columnist on the *MSDN Magazine* Web site, where she wrote the Web Dev Report.

Also coming to us from the *MSDN Magazine* Web site are Bruno Terkaly and Ricardo Villalobos, who team up to bring their Windows Azure Insider column to print. Fitting the theme of this special issue, their inaugural column describes how to build a simple cloud-hosted service to support asynchronous clients, then shows how easy it is to call into a Web service from a Windows Store application to retrieve data.

Obviously there's a huge amount of interest in the new Windows Store app UI and the underlying Windows Runtime, and our coverage here and going forward will reflect that interest. Our upcoming November issue, for instance, will include Windows Store app-focused explorations of C# and C++ memory-management techniques, security practices in JavaScript and design guidance for creating more-effective Windows Store app UIs.

One reason for all the excitement, says Jason Olson, senior program manager working on the Windows Runtime, is that Microsoft has pitched the proverbial big tent with the Windows Runtime. Right off the bat, developers working with C++, C#, Visual Basic and JavaScript can get to work creating Windows Store apps. That's a huge and diverse community of developers.

"I'm very excited—this is a big value-add for the Windows Runtime itself," Olson says of language support in the Windows Runtime. "Many different types of developers can continue leveraging their skills and assets when building Windows Store apps on Windows 8."

Olson, in his feature ("Reimagining App Development with the Windows Runtime," p. 20), also details the value of hybrid apps under the Windows Runtime. He writes:

"There are no boundaries. As a JavaScript developer, you aren't limited to JavaScript libraries. As a C#/Visual Basic developer, you aren't limited to .NET libraries. And as a C++ developer, you aren't limited to C/C++ libraries."

As Olson told me in an interview, "Developers can take full advantage of the 'language choice' ability in the Windows Runtime to take advantage of many different types of code assets they have access to, regardless of whether the asset is native code, managed code or JavaScript code."

And that, to be frank, is barely the tip of the iceberg. There's so much going on inside, around and on top of the Windows Runtime that we could never address even a fraction of it in this special edition. Which is why *MSDN Magazine* readers can look forward to in-depth explorations of Windows 8, the Windows Runtime and Windows Store app development in the months to come.

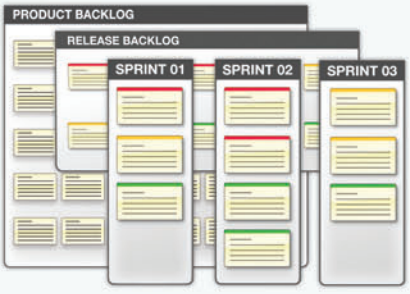
Is there are a particular issue, challenge or technology related to Windows 8 and the Windows Runtime you'd like to see covered in our pages? Let us know. Send me an e-mail at mmeditor@microsoft.com.

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

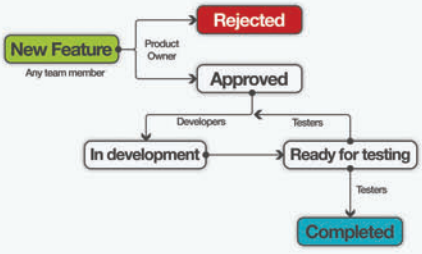
OnTime Scrum

Agile project management & bug tracking software

The **Scrum project management** tool your development team will **love to use.**



Easily manage product backlogs



Automate your workflow process

Project visibility with burndowns

Enterprise-quality software at prices **any team can afford.**

$10 per month for up to 10 users

special small-team pricing

$7 per user per month

for teams of 11+ users

Exclusive offer for MSDN readers: **3 months free** of **OnTime Scrum**

Visit **OnTimeNow.com/MSDN**

OnTime Scrum offers your dev team:

- flexible product backlog management
- powerful user story & bug tracking
- highly configurable user roles & workflows
- automated burndown charts
- **new** real-time visual project dashboard
- Visual Studio & Github integration
- all in a fast, versatile HTML5 interface

axosoft

800.653.0024 • www.ontimenow.com • www.axosoft.com • @axosoft

RACHEL APPEL

The Windows Store App Lifecycle

Windows 8 is changing how and when applications run, and you'll want to understand the nuances of the new application lifecycle so you can build apps that respond as they should at every point. Apps that conform to the Microsoft lifecycle management guidelines offer a better experience for the user, especially on small devices where memory and battery conservation are warranted.

Application Design

Two key design concepts underlie Windows Store apps: apps can run in full screen, snapped or filled mode; and apps must be highly responsive so the user can focus on the content at hand with fewer distractions. These two principles ensure that the currently running app gets all the available resources from the OS and the user, unlike Windows 8 desktop apps or apps on previous Windows versions, which had to share those resources.

> By responding suitably to app lifecycle events, you can ensure users have a consistent experience throughout the app's lifecycle.

All Windows Store apps have four states: not running, running, suspended and terminated. When you launch an app, it runs. Later, depending on user or system activity, the app might transition between running and suspended states. For example, if the user switches from app A to app B, Windows suspends app A after a short delay, moving it to the background. App A remains suspended (until the user switches back to it or Windows terminates it), while app B activates and moves into the running state (possibly from a suspended state if it was already in memory). If the user returns to app A, Windows simply wakes it up and, as far as both the OS and app A know, it's been running all along. Then, of course, it's app B's turn to be suspended.

When an app is in a suspended state, its code doesn't run and the app remains in memory, as is. Essentially, the app is cached and instantly ready to go when the user switches back. However, that's not the whole story—you can run background tasks if you follow the proper procedures, and Windows might terminate apps if memory pressure warrants it. **Figure 1** illustrates how apps move between execution states (see bit.ly/TuXN9F for more information).

As **Figure 1** suggests, an app can move between running and suspended states frequently. The app lifecycle state transitions highlight the difference between Windows Store apps and traditional desktop apps.

Visual Studio 2012 includes a rich set of debugging tools as well as a Windows Simulator (bit.ly/QzBy5I), which you can use to manage app lifecycle operations. That's important, because you need to react appropriately to app lifecycle events and handle state transitions properly in order to create a highly responsive experience for the end user—a core principle of app design on Windows 8. By responding suitably to app lifecycle events, you can ensure users have a consistent experience throughout the app's lifecycle. Specifically, this means saving and restoring the state of your app whenever necessary. This might entail returning the user back to where she was in your application (such as in a wizard), or repopulating the values of a form she was working on, or returning to the last article she was reading. Because application lifecycle is controlled by the user's movement through an app, an app needs to be prepared to checkpoint its state at a moment's notice—whenever the app gets the suspending event.

App Activation

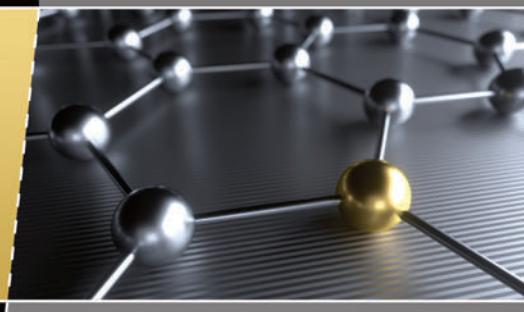
The WWAHost.exe process is an app host that executes Windows Store JavaScript apps. XAML apps written with any language—such as C#, Visual Basic or C++—run the app's corresponding executable. Either way, all apps go through activation, which has a number of catalysts:

- The user launches an app from a tile.
- The user switches to a suspended app.
- Windows launches the app through a Search or Share Target contract.
- Windows invokes a protocol (URI scheme) association (bit.ly/QyzX04) or launches an app due to a file association.
- Windows invokes an extension, such as a File Open Picker contract or Contact Picker.

How activation occurs determines what code needs to run. Launches from a tile, contract or protocol cause a Windows Library for JavaScript (WinJS) app to fire an activated event and a XAML app to fire an OnLaunched event. During these events, you check the app's previous state to take appropriate actions.

Powerful Tools for Developers

New! V4.6



Create & Edit PDFs in .Net - ActiveX - WinRT

- Edit, process and print PDF 1.7 documents programmatically.
- Fast and lightweight 32 and 64-bit components for .Net, COM and WinRT applications.
- Create, fill-out and annotate PDF forms.

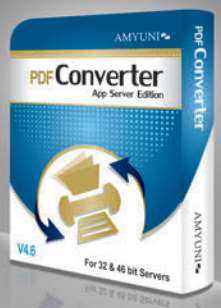
Complete Suite of Accurate PDF Components

- All your PDF processing, conversion and editing in a single package.
- Combines Amyuni PDF Converter and PDF Creator for easy licensing, integration and deployment.
- Includes our Microsoft certified PDF Converter printer driver.
- Export PDF documents into other formats such as JPeg, Word, Excel or Silverlight.

Advanced HTML to PDF & XAML

- Direct conversion of HTML files into PDF and XAML without the use of a web browser or a printer driver.
- Easy Integration and deployment within developer's applications.
- WebkitPDF is based on the Webkit Open Source library and Amyuni PDF Creator.

High Performance PDF Printer Driver



- Our high-performance printer driver optimized for Web, Application and Print Servers. Print to PDF in a fraction of the time needed with other tools. WHQL tested for Windows 32 and 64-bit including Windows Server 2008 and Windows 8.
- Standard PDF features included with a number of unique features. Interface with any .Net or ActiveX programming language.
- Easy licensing and deployment to fit system administrator's requirements.

Windows 7 | CERTIFIED FOR Windows Server 2008 | Ready for Windows 8

AMYUNI

All development tools available at

www.amyuni.com

USA and Canada
Toll Free: 1866 926 9864
Support: 514 868 9227
sales@amyuni.com

Europe
UK: 0800-015-4682
Germany: 0800-183-0923
France: 0800-911-248

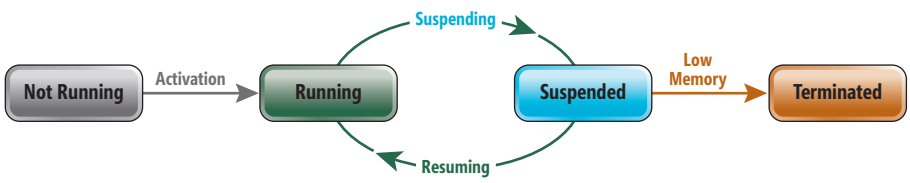


Figure 1 **How Windows Store Apps Move Between Execution States**

If the app is moving from a not-running state to a running state, you need to load fresh data, as this transition indicates the app launch came from a tile, charm, protocol or extension. If the app is returning from a suspended state, you generally need to do nothing; the user can just pick up where he left off. If the app is moving from a terminated state to a running state, you need to reload the data and navigate in the app to the last place the user was (unless it has been months since the last use). Code to handle activation is shown in JavaScript in **Figure 2** and in C# in **Figure 3**. As the code shows, you can test for the previous execution state as well as how Windows launched the app, because the Windows API has a handy set of enumerations for both cases. Once you have that information, you can repopulate the data as needed.

Both the JavaScript activated and XAML OnLaunched events contain an args argument you can query to determine the state the app was in prior to activation. **Figure 2** and **Figure 3** show examples of the activated event.

Code that runs during activation must run within 15 seconds or Windows will terminate the app. Though this may seem strict, apps that block the user from interacting with the UI for that long aren't responsive. According to UseIt.com research on users and response times (bit.ly/NWSumy), most users abandon Web sites that take more than 10 seconds to load. In fact, users become frustrated when pages take more than 1 or 2 seconds to load, and many leave far before the 10-second benchmark. Knowing this about users is especially important if you plan to sell your app in the Windows Store, because frustrated users post negative app reviews—not to mention that your app might not even pass store certification if it performs poorly. (See bit.ly/OxuEfu for more information on submitting an app.) It's important to carefully consider what data to load and when. Fortunately, the Windows Runtime (WinRT) libraries make it easy to load data quickly and progressively by offering a first-class asynchronous programming model (bit.ly/NCx6gE).

Figure 2 **JavaScript App-Activation Code**

```
var app = WinJS.Application;
var activation = Windows.ApplicationModel.Activation;
var nav = WinJS.Navigation;
app.addEventListener("activated", function (args) {
  if (args.detail.kind === activation.ActivationKind.launch) {
    if (args.detail.previousExecutionState !==
      activation.ApplicationExecutionState.terminated) {
      // TODO: This application has been newly launched.
      // Initialize your application here.
    } else {
      // TODO: This application has been reactivated from suspension.
      // Restore application state here.
    }
    if (app.sessionState.history) {
        nav.history = app.sessionState.history;
    }
    args.setPromise(WinJS.UI.processAll().then(function () {
      if (nav.location) {
          nav.history.current.initialPlaceholder = true;
          return nav.navigate(nav.location, nav.state);
      } else {
        return nav.navigate(Application.navigator.home);
      }
    }));
  }
});
```

The Windows.ApplicationModel.Activation object is part of the Windows API accessible by all Windows Store apps, regardless of language. The Activation object contains an activationKind enumeration with the following values:

- **Launch:** The user launched the app or tapped a secondary tile.
- **Search:** The user wants to search with the app.
- **ShareTarget:** The app is activated as a target for share operations.

According to UseIt.com research on users and response times, most users abandon Web sites that take more than 10 seconds to load.

- **File:** An app launched a file whose file type this app is registered to handle.
- **Protocol:** An app launched a URL whose protocol this app is registered to handle.
- **FileOpenPicker:** The user wants to pick files or folders provided by the app.
- **FileSavePicker:** The user wants to save a file and selected the app as the location.
- **CachedFileUpdater:** The user wants to save a file for which the app provides content management.
- **ContactPicker:** The user wants to pick contacts.
- **Device:** The app handles AutoPlay.
- **PrintTaskSettings:** The app handles print tasks.
- **CameraSettings:** The app captures photos or video from an attached camera.

Because your app might support multiple features, such as both share and search, the activationKind enumeration allows you to see what the user's intent is for launching the app, and then to execute corresponding code.

Managing Suspension, Termination and Resume

When a user switches to a different app, or the device hibernates or goes to sleep, Windows stops the app code from running, yet

keeps the app in memory. The reason for this suspension is that it minimizes the battery and performance impact of apps the user is not actively getting value from. When Windows wakes up the app from suspension or a short termination, the user should feel as if the app had never stopped running. Proper handling of lifecycle events ensures a responsive design.

Windows starts the suspension process by raising either a WinJS oncheckpoint event or a XAML OnSuspending event where you can save data and user information before the app moves into suspended mode. Any code you run in these events must complete within 10 seconds or Windows stops the app completely. Just as with the activation process, this rule keeps the overall health of the OS stable and enables fast switching. Hooking into the oncheckpoint event requires a simple function definition:

```
app.oncheckpoint = function (args) {
  // Save app data in case of termination.
};
```

In XAML, you use an Application.Suspending event:

```
async void Suspending(Object sender,
  Windows.ApplicationModel.SuspendingEventArgs e) {
  // Save app data in case of termination.
}
```

During suspension you need to save the user's location or scroll position in the app, release file handles and connections, and date- and time-stamp the data. You can accomplish this using the built-in WinJS.Application.sessionState or XAML SuspensionManager objects. You should always save user information and app data in the suspending event because Windows doesn't notify apps before it terminates them. This is important because termination can occur under a variety of circumstances, such as when Windows needs to free memory or the device loses (battery) power.

So code defensively and assume the app will terminate. This means you should save app data to sessionState every time. If Windows does terminate the app, the data is saved and ready for repopulation.

Resume occurs when Windows wakes up an app from suspension. Most of the time, Windows will simply resume your app when the user switches back to it or re-launches it, and code-wise you need to do nothing. During suspension the app just sits in memory, untouched, so the app content remains unchanged and there's no need to take any action.

While most apps won't need to do any work upon resuming from a suspended state, apps that contain frequently changing data—such as RSS feeds, stocks or social networking—are good candidates for using the resuming event. Because the resuming event fits these few specific scenarios, it lives in the Windows.UI.WebUI.WebUIApplication object rather than in the more common WinJS.Application object, alongside the oncheckpoint, onactivated and other sibling lifecycle events:

```
Windows.UI.WebUI.WebUIApplication.addEventListener("resuming", function () {
  // Repopulate data from sessionState.
});
```

You can add the resume handler to the default.js file near the activation code for code-organization purposes. Keep your app responsive by quickly loading just the minimal amount of data while resuming.

Figure 3 **C# App-Activation Code**

```
async protected override void OnLaunched(LaunchActivatedEventArgs args)
  {
// Check whether the session data should be restored.
    if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      // Here we've created a SuspensionManager class that
      // handles restoring session data from a file and
      // then gives access to that data through a Dictionary.
      await SuspensionManager.RestoreAsync();

// Retrieve the data for restore.
data = SuspensionManager.SessionState["savedData"];
    }
    // If not, use the app's default values
    else
            {
data = "Welcome";
    }
    Window.Current.Activate();
}
```

Background Tasks and Real-Time Communications Apps

Although the UI code doesn't run while an app is in suspended mode, an app can perform background tasks (bit.ly/QyRvsU), transfer large files or check for e-mail. Some real-time communications apps, such as IM clients, need to be able to notify the user when a message arrives, regardless of the app execution state.

Background tasks are lightweight classes that run periodically, under certain restrictions, while the app is technically not running. They run in their own sandboxes or app containers while the app remains in a not-running state. JavaScript background tasks run in a new single-threaded apartment of WWAHost.exe, and non-JavaScript background tasks run in an in-process .dll loaded into its own threading apartment in the app. This separation allows background tasks to run independently of an app's UI, which remains suspended until the user returns to the app.

In addition to being able to execute code in the background, these apps also display information on the lock screen. The lock screen contains a background image with lightweight information superimposed on it, such as time, date, battery status and network status. In addition, apps—specifically apps that are running in the background—can display information on the lock screen. Because it's important to keep the lock screen easily digestible, the information that can be displayed is limited. Badges from up to seven apps can be displayed, and the text from a single app's tile can be shown on the lock screen. For more information, see the lock screen overview at bit.ly/RsE7pj.

Follow the Model

As a developer of Windows Store apps, you need to be concerned with issues such as battery life and memory pressure, running your app on a variety of devices and, most important, the UX. You'll find your goals easier to achieve if you follow the prescribed lifecycle management guidelines. Moreover, besides the technical reasons for app lifecycle management, if your app is highly responsive and performs well, it will get better reviews in the Windows Store. ■

Rachel Appel *is a developer evangelist at Microsoft New York City. Reach her via her Web site at rachelappel.com or by e-mail at rachel.appel@microsoft.com. You can also follow her latest updates on Twitter at twitter.com/rachelappel.*

Thanks to the following technical experts for reviewing this article:
Adam Barrus, Ben Betz, Arun Kishan, Michael Krause, Hari Pulapaka, John Sheehan and Ben Srour

File APIs

Modify Print Combine

Save Convert

100% Standalone - No Office Automation

.NET Java SSRS SharePoint JasperReports

Powerful file format components and controls

Aspose.Words

DOC
DOCX
RTF
HTML
PDF
XPS
& other doc formats

Aspose.Cells

XLS
XLSX
XLSM
XLTX
CSV
SpreadsheetML
& image formats

Aspose.Pdf

PDF
XML
XSL-FO
SVG
HTML
& other formats
... and more!

Aspose.BarCode Aspose.Tasks Aspose.Email
Aspose.Diagram Aspose.OCR Aspose.Imaging

ASPOSE

Your File Format Experts

Get your FREE evaluation copy at
http://www.aspose.com

EU Sales: +44 (0)141 416 1112
sales.europe@aspose.com
AU Sales: +61 2 8003 5926
sales.asiapacific@aspose.com

US Sales: 1.888.277.6734
sales@aspose.com

BRUNO TERKALY AND
RICARDO VILLALOBOS

Windows 8 + Windows Azure: Convergence in the Cloud

There's little question that today's software developer must embrace cloud technologies to create compelling Windows Store applications—the sheer number of users and devices make that a no-brainer. More than one-third of the Earth's population is connected to the Internet, and there are now more devices accessing online resources than there are people. Moreover, mobile data traffic grew 2.3-fold in 2011, more than doubling for the fourth year in a row. No matter how you look at it, you end up with a very simple conclusion: Modern applications require connectivity to the cloud.

The value proposition of cloud computing is compelling. Most observers point to the scalability on demand and to the fact that you pay only for what you use as driving forces to cloud adoption. However, the cloud actually provides some essential technology in the world of multiple connected devices. Windows Store application users, likely to be using many applications and multiple devices, expect their data to be centrally located. If they save data on a Windows Phone device, it should also be immediately available on their tablets or any of their other devices, including iOS and Android devices.

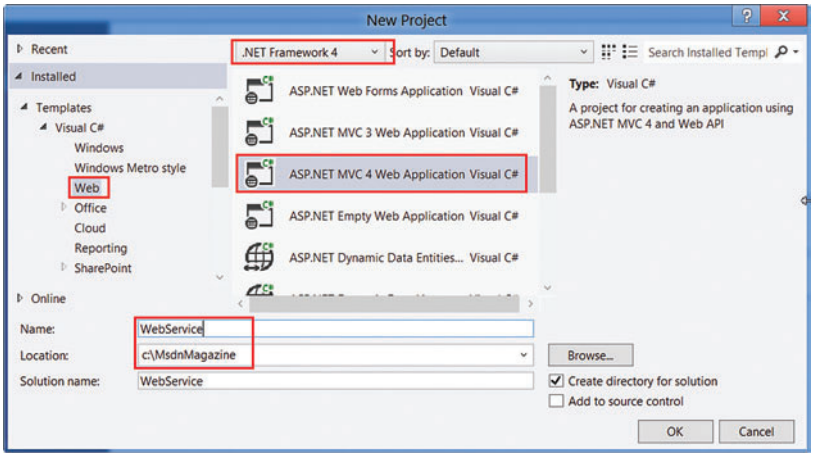


Figure 1 **New Project Dialog Box**

> There's little question that today's software developer must embrace cloud technologies to create compelling Windows Store applications.

Windows Azure is the Microsoft public cloud platform, offering the biggest global reach and the most comprehensive service back end. It supports the use of multiple OS, language, database and tool options, providing automatic OS and service patching. The underlying network infrastructure offers automatic load balancing and resiliency to hardware failure. Last, but not least, Windows Azure supports a deployment model that enables developers to upgrade applications without downtime.

The Web service application presented in this article can be hosted in one or more of Microsoft's global cloud datacenters in a matter of

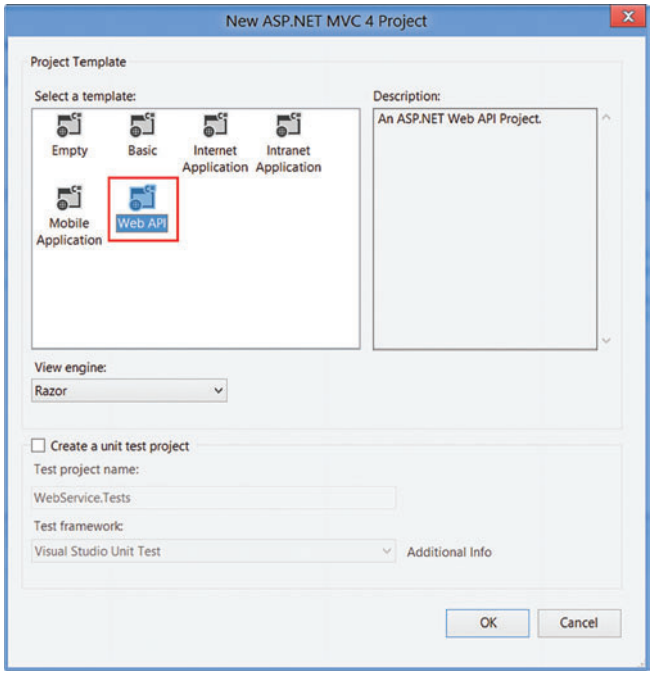


Figure 2 **Project Template Dialog Box**

Code download available at archive.msdn.microsoft.com/mag201210AzureInsider.

minutes. Whether you're building a to-do list application, a game or even a line-of-business accounting application, it's possible to leverage the techniques in this article to support scenarios that rely on either permanently or occasionally connected clients.

What You'll Learn

First, we'll describe how to build a simple cloud-hosted service on Windows Azure to support asynchronous clients, regardless of the type of device it's running on—phone, slate, tablet, laptop or desktop. Then we'll show you how easy it is to call into a Web service from a Windows Store application to retrieve data.

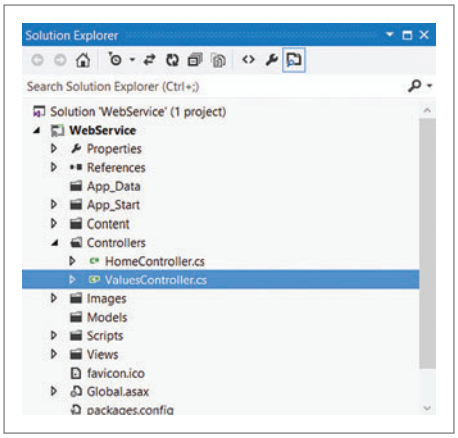


Figure 3 **Solution Explorer for WebService**

Generally speaking, there are two ways data can make its way into a Windows Store application. This article will focus on the "pull approach" for retrieving data, where the application needs to be running and data requests are issued via HTTP Web calls. The pull approach typically leverages open standards (HTTP, JSON, Representational State Transfer [REST]), and most—if not all—device types from different vendors can take advantage of it.

> The value proposition of cloud computing is compelling.

We won't be covering the "push approach" in this article. This approach relies on Windows Push Notification Services (WNS), which allows cloud-hosted services to send unsolicited data to a Windows Store application. Such applications don't need to be running in the foreground and there's no guarantee of message delivery. For information about using WNS, see bit.ly/RSXomc.

Two Projects

The full solution requires two main components: a server-side or Web services project (which can be deployed on-premises or in Windows Azure), and a client-side project, which consists of a Windows Store application based on the new Windows UI. Both projects can be created with Visual Studio 2012.

Basically, there are two options for building the Web services project: Windows Communication Foundation (WCF) or the ASP.NET Web API, which is included with ASP.NET MVC 4. Because exposing services via WCF is widely documented, for our scenario we'll use the more modern approach that the ASP.NET Web API brings to the table, truly embracing HTTP concepts (URIs and verbs). Also, this framework lets you create services that use more advanced HTTP features, such as request/response headers and hypermedia constructs.

Both projects can be tested on a single machine during development. You can download the entire solution at archive.msdn.microsoft.com/mag201210AzureInsider.

What You Need

The most obvious starting point is that Windows 8 is required, and it should come as no surprise that the projects should be created using the latest release of Visual Studio, which can be downloaded at bit.ly/win8devtools.

For the server-side project, you'll need the latest Windows Azure SDK, which includes the necessary assemblies and tooling for creating cloud projects from within Visual Studio. You can download this SDK and related tooling at bit.ly/NIB50B. You'll also need a Windows Azure account. A free trial can be downloaded at bit.ly/azuretestdrive.

Historically, SOAP has been used to architect Web services, but this article will focus on REST-style architectures. In short, REST is easier to use, carries a smaller payload and requires no special tooling.

Developers must also choose a data-exchange format when building Web services. That choice is generally between JSON and XML. JSON uses a compact data format based on the JavaScript language. It's often referred to as the "fat-free alternative" to XML because it has a much smaller grammar and maps directly to data structures used in client applications. We'll use JSON data in our samples.

With these decisions made, we're ready to create our Web service. Our goal is to build an HTTP-based service that reaches the broadest possible range of clients, including browsers and mobile devices.

Building the Web Service

Let's begin by starting Visual Studio 2012 as Administrator. Here are the steps to create the server-side Web service, using the ASP.NET MVC 4 Web API:

Figure 4 **ValuesController Class**

```
public class ValuesController : ApiController
{
  // GET api/values
  public IEnumerable<string> Get()
  {
    return new string[] { "value1", "value2" };
  }

  // GET api/values/5
  public string Get(int id)
  {
    return "value";
  }

  // POST api/values
  public void Post([FromBody]string value)
  {
  }

  // PUT api/values/5
  public void Put(int id, [FromBody]string value)
  {
  }

  // DELETE api/values/5
  public void Delete(int id)
  {
  }
}
```

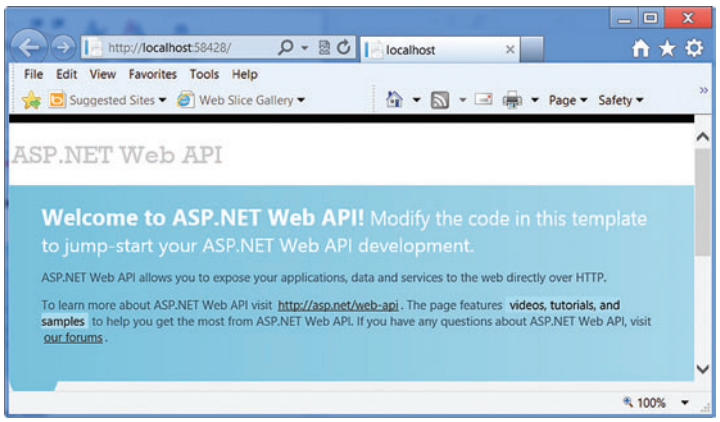


Figure 5 **Using Internet Explorer to Test the Web Service**

Figure 6 **Internet Explorer Dialog Box**

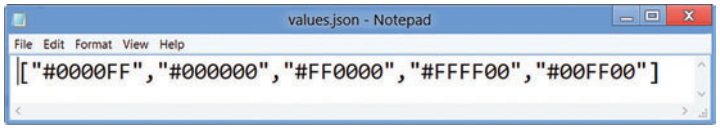


Figure 7 **The JSON Data Returned by the Web Service**

1. Click on the File menu and choose New | Project (see **Figure 1**).
2. Select Visual C# | Web for the template type.
3. Select .NET Framework 4 in the dropdown at the top.
4. Select ASP.NET MVC 4 Web Application.
5. Enter the Name WebService and a Location of your choice.
6. Click OK.
7. The wizard will ask you to select the Project Template. Choose Web API and be sure the View engine is Razor (the default), as shown in **Figure 2**.
8. A variety of files will be generated by Visual Studio. This can be overwhelming, but we only need to worry about a couple of files, as shown in **Figure 3**.

ValuesController.cs is important because it contains the code that will execute when the Windows 8 client submits an HTTP request against the Web service. This is where we'll add code to return the JSON data required by the Windows Store application. The ValuesController class, shown in **Figure 4**, is generated by Visual Studio and it inherits from ApiController, which returns data that's serialized and sent to the client, automatically in JSON format.

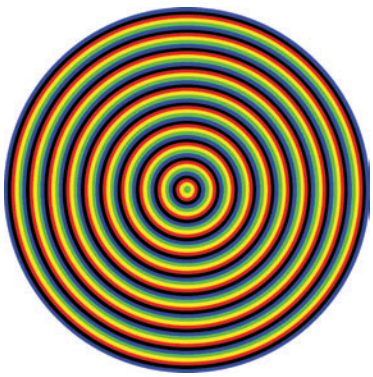


Figure 8 **The Running Windows Store Application**

Note that the methods in **Figure 4**—Get, Post, Put and Delete—map to specific create, read, update and delete (CRUD) operations and HTTP verbs executed by the Windows Store application. This is the beauty of the ASP.NET Web API framework: It automatically routes the HTTP verbs used by the client directly to the methods defined in the ValuesController class, minimizing potential programming mistakes.

Out of the box, the WebService project is ready to run. The methods in the ValuesController class are self-documenting:

```
// GET api/values
public IEnumerable<string> Get()
{
  return new string[] { "value1", "value2" };
}
```

Note that the Get method is called when the client issues the HTTP verb Get using the URL http://localhost:[port]/api/values. For the purpose of this demo, the code draws concentric circles based on colors returned from the Web service. Modify the preceding code to look like this:

```
// GET api/values
public IEnumerable<string> Get()
{
  return new string[] { "#0000FF", "#000000", "#FF0000",
    "#FFFF00", "#00FF00" };
}
```

As you can see, we now return an array of strings of various colors. The Windows Store application we create will render these strings as actual colors, specifically as colored concentric circles.

Testing with a Browser

It's always a good idea to test the Web service before creating the Windows Store application. Use any browser you like.

> Windows Store application users, likely to be using many applications and multiple devices, expect their data to be centrally located.

To test the Web service from a browser, perform the following steps:

1. In Visual Studio, click on the Debug menu and choose Start Debugging.
2. You should see the default start page, as shown in **Figure 5**.
3. Notice that Visual Studio chooses an arbitrary port, in this case 58428. The port your instance of Visual Studio uses will likely differ.
4. As you'll recall, we need to add "api/values" to the URL so the Get method gets called, which means the final URL (in our case) should be http://localhost:58428/api/values. This is a built-in feature when Visual Studio creates your project. You can change this mapping to suit your needs.

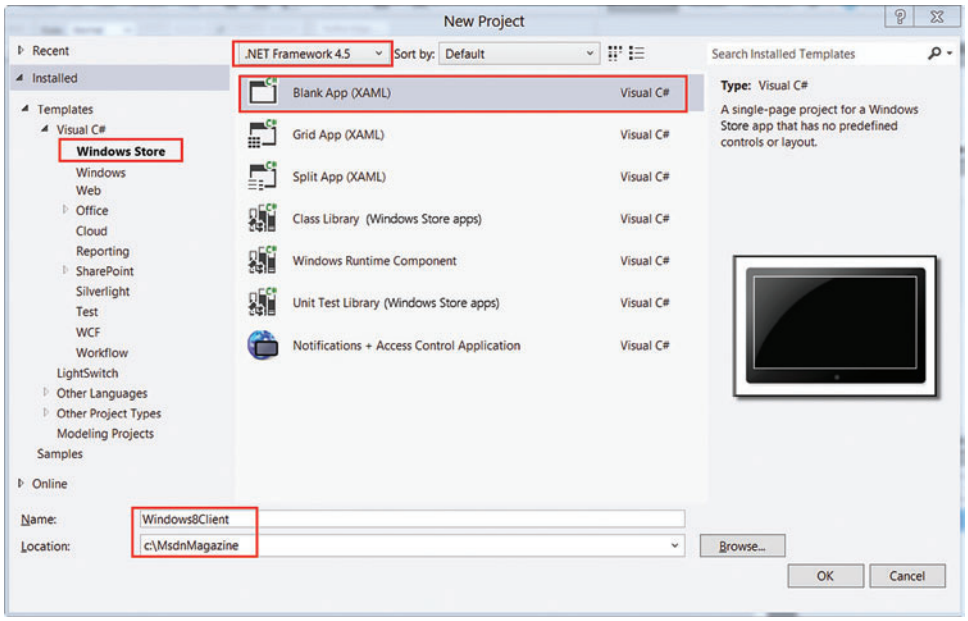


Figure 9 **New Project Dialog Box**

5. Go to the address bar of Visual Studio and type in http://localhost:58428/api/values. Remember to replace the port number with your own value.
6. A dialog box will appear like the one in **Figure 6**.
7. Click on the Save button, then open values.json in Notepad.
8. Notepad will display the values returned. Notice in **Figure 7** that the data returned is automatically in JSON format, courtesy of the ASP.NET Web API framework.

Windows Azure is the Microsoft public cloud platform, offering the biggest global reach and the most comprehensive service back end.

Enter Windows 8

Now that we've succeeded in returning JSON data from our Web service, let's see how to consume this data from a Windows Store application. The goal is to read the colors from the Web service and to draw the concentric circles based on the values returned by the Web service. Our final Windows Store application should produce an image that looks like **Figure 8**. The colors you see were defined by the Web service. If the Web service changes the colors returned, the Windows Store application would reflect those colors as the concentric circles you see in **Figure 8**.

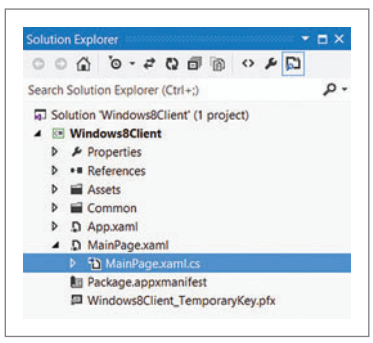


Figure 10 **Solution Explorer for Windows8Client**

You can either create the Windows Store application in the same solution as the Web service project, or you can start a new instance of Visual Studio specifically for the Windows 8 client. We'll start a new instance of Visual Studio as Administrator for the Windows Store app. To do this, follow these steps (see **Figure 9**):

1. Start a second instance of Visual Studio as Administrator.
2. Click on the File menu and choose New | Project.
3. Select Visual C# | Windows Store for the template type.
4. Select .NET Framework 4.5 in the dropdown at the top.
5. Select Blank App (XAML).
6. Enter the Name Windows8Client and a Location of your choice.
7. Click OK.

When you're finished with these steps, navigate to the View menu and choose Solution Explorer (see **Figure 10**), and open the file MainPage.xaml.cs.

Calling into the Web Service

We're ready to add the code that will call into the Web service from the Windows Store application. The Windows Store application will issue a Web request against the Web service.

To support a Web request, MainPage.xaml.cs will be modified as follows:

- A Loaded event will be added for the built-in Grid control.
- The Loaded event will submit an asynchronous Web request to call into the Web service. The HttpClient class will be used for this purpose.
- To convert the color string from the Web service into a real color, a method called CircleColorFromString will be added.
- To draw the concentric circles, we'll add a method called AddCircle.

Perform the following steps to issue a Web request against the Web service:

1. Right-click MainPage.xaml and choose View Designer.
2. Add the GridLoadedEvent and the Canvas control as follows (the canvas is where we'll draw the circles):

```
<Grid Name="maingrid"
      Background="{StaticResource
                   ApplicationPageBackgroundThemeBrush}"
      Loaded="GridLoadedEvent">
  <Canvas Name="myCanvas" Background="White"/>
</Grid>
```

3. Double-click MainPage.xaml.cs. The codebehind for MainPage.xaml.cs appears.

Figure 11 **MainPage.xaml.cs**

```
public sealed partial class MainPage : Page
  {
    // [ Generated code omitted for brevity ]
    private void AddCircle(Color color, int diameter, int t, int l)
    {
      //
      // Build a circle using the attributes provided as parameters.
      //
      Ellipse newColoredCircle = new Ellipse();
      newColoredCircle.Fill = new SolidColorBrush(color);
      newColoredCircle.StrokeThickness = 1;
      newColoredCircle.Stroke = new SolidColorBrush(color);
      newColoredCircle.Width = diameter;
      newColoredCircle.Height = diameter;
      //
      // Add the circle to the Canvas control.
      //
      myCanvas.Children.Add(newColoredCircle);
      Canvas.SetTop(newColoredCircle, t);
      Canvas.SetLeft(newColoredCircle, l);
    }
    Color CircleColorFromString(string rgb)
    {
      //
      // Convert the string-based hex values
      // into a Color object and return to caller.
      //
      Color ringColor = new Color();
      byte a = 255;
      byte r = (byte)(Convert.ToUInt32(rgb.Substring(1, 2), 16));
      byte g = (byte)(Convert.ToUInt32(rgb.Substring(3, 2), 16));
      byte b = (byte)(Convert.ToUInt32(rgb.Substring(5, 2), 16));
      ringColor = Color.FromArgb(a, r, g, b);
      return ringColor;
    }

    private async void GridLoadedEvent(object sender, RoutedEventArgs e)
    {
      //
      // Retrieve colors from Web service.
      //
      var client = new HttpClient();
      client.MaxResponseContentBufferSize = 1024 * 1024;

      //
      // Asynchronously call into the Web service.
      //
      var response = await client.GetAsync(
        new Uri("http://localhost:58428/api/values"));

      var result = await response.Content.ReadAsStringAsync();

      //
      // Parse the JSON data
      //
      var parsedResults = JsonArray.Parse(result);

      //
      // Build concentric circles, repeating colors as needed
      //

      const double startingPercentage = 1.0;
      const double desiredCircleCount = 50.0;
      for (int i = 0; i < desiredCircleCount; i++)
      {
        Color circleColor = CircleColorFromString(
          parsedResults[i % parsedResults.Count].GetString());
        int circleDiameter = Convert.ToInt32((startingPercentage -
          (i / desiredCircleCount)) * this.RenderSize.Width / 2.0);
        int top = Convert.ToInt32((this.RenderSize.Height -
          circleDiameter)/2.0);
        int left = Convert.ToInt32((this.RenderSize.Width -
          circleDiameter)/2.0);
        AddCircle(circleColor, circleDiameter, top, left);
      }
    }
  }
```

4. To support the different namespaces required by the code to be added, include the following references at the top of MainPage.xaml.cs:

```
using System.Net.Http;          // for HttpClient()
using Windows.Data.Json;        // for parsing JSON Array data
using Windows.UI.Xaml.Shapes;   // for the Ellipse object
using Windows.UI                // for the Color class
```

5. Add the supporting routines to draw the circles and parse the JSON string data sent by the Web service. Edit the code in MainPage.xaml.cs as shown in **Figure 11**.

Now test the application, which is quite simple:

1. Return to the WebService project in Visual Studio and select Start Debugging from the Debug menu.
2. Verify that the browser starts. You're now ready to start the Windows8Client application.
3. Return to the Windows8Client project in Visual Studio and select Start Debugging from the Debug menu.
4. Verify the Windows Store application appears, as shown in **Figure 8**.

Deploying the Web Service to Windows Azure

Once the Web service has been created and tested locally, the next logical step is to deploy it to the cloud for scalability purposes. This lets you vary the compute capacity assigned to the Web service—and its cost—as the number of requests from Windows 8 devices increases or decreases. Visual Studio offers different ways to deploy ASP.NET Web API projects to Windows Azure, and the only prerequisite is to add a cloud service project to the Web service solution. Follow these steps:

1. Right-click on the Solution name, and select Add | New project.
2. Select Visual C# | Cloud as the project template (**Figure 12**).
3. Select .NET Framework 4 in the dropdown at the top.
4. Select Windows Azure Cloud Service Visual C#.

> Visual Studio 2012 and the libraries included with the .NET Framework 4.5 make it very simple to issue asynchronous Web requests against a Web service.

5. Enter the Name WindowsAzureCloudService and a Location of your choice.
6. Click OK.
7. Now go back to the Solution Explorer. Find the new project in the solution and right-click on the Roles node. Select Add | Web Role Project.

componentart.com/windows8

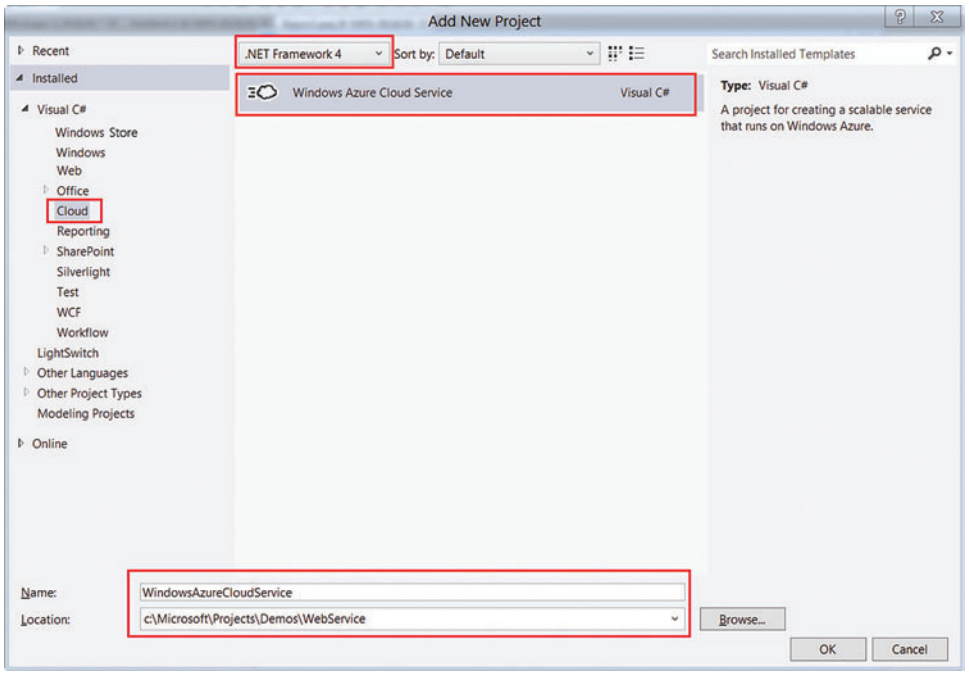


Figure 12 **Adding a Cloud Service Project to the Web Service Solution**

8. Find the previously created WebService project. Select it and click OK.
9. You should now see the WebService project listed under the Roles node as shown in **Figure 13**.
10. You can change various deployment parameters for the WebService role before deploying it to Windows Azure, such as the size of the virtual machine (VM) where it's going to be hosted and the number of instances supporting it. To review these parameters, right-click on the WebService Roles node and select Properties. A dialog like the one in **Figure 14** is displayed. Pay special attention to the Instance count and the VM size parameters, which will determine the consumption rate once the role has been deployed. For more information about Windows Azure pricing for compute instances, visit bit.ly/SnLj68.
11. Right-click on the WindowsAzureCloudService project, and select Package. Keep the default selections for the Service and Build configuration parameters, and click on the Package button. This action will generate two files: WindowsAzureCloudService.cspkg and ServiceConfiguration.Cloud.cscfg, which are required to create a cloud service on Windows Azure via the portal. Note that projects can be directly deployed from Visual Studio, but we won't cover this approach here.
12. Log in to your Windows Azure account by opening the URL manage.windowsazure.com, then follow the instructions at bit.ly/R15VNj for creating a new cloud service and uploading the two generated files.
13. Depending on the name you use for the cloud service, your new ASP.NET Web API will be available at http://{cloudservicename}.cloudapp.net/api/values.
14. Point the Windows Store application to this URL, and you're done.

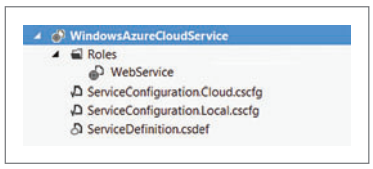


Figure 13 **Adding the WebService Project to a Cloud Service Role**

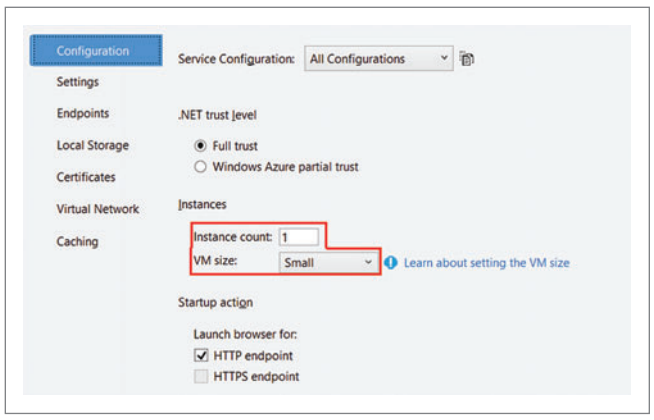


Figure 14 **Windows Azure Deployment Properties for the WebService Role**

In the Cloud

By using the ASP.NET Web API, it's extremely simple to support applications that asynchronously use REST commands for consuming and ingesting data via Web services. Visual Studio 2012 and the libraries included with the .NET Framework 4.5 make it very simple to issue asynchronous Web requests against a Web service. Last, but not least, the integration of the development environment with Windows Azure allows developers to rapidly deploy the services portion to the cloud, taking advantage of the scalability and worldwide infrastructure capabilities the cloud provides. ■

Bruno Terkaly *is a developer evangelist for Microsoft. His depth of knowledge comes from years of experience in the field, writing code using a multitude of platforms, languages, frameworks, SDKs, libraries and APIs. He spends time writing code, blogging and giving live presentations on building cloud-based applications, specifically using the Windows Azure platform.*

Ricardo Villalobos *is a seasoned software architect with more than 15 years of experience designing and creating applications for companies in the supply chain management industry. Holding several technical certifications as well as a master's degree in business administration from the University of Dallas, he works as a cloud architect in the Windows Azure CSV incubation group for Microsoft.*

Thanks to the following technical expert for reviewing this article: *Robert Green*

componentart.com/windows8

Reimagining App Development with the Windows Runtime

Jason Olson

Developing Windows applications and using new Windows features from your programming language of choice has not always been simple and straightforward. For customers to have the best experience on Windows 8, and to empower developers to create the best apps on Windows, investments needed to be made to improve the end-to-end development experience on Windows. At the heart of these investments is the Windows Runtime (WinRT).

The Windows Runtime is part of a reimagining of the developer experience for Windows. It's the modern Windows API surface used to create new Windows Store apps on Windows 8. The Windows Runtime is designed from the ground up to equally support several major programming languages (C#/Visual Basic, JavaScript and C++), to allow developers to leverage their existing skills and assets, to provide a thoughtful and consistently designed API surface, and to be deeply integrated into the developer tool chain.

This article discusses:

- The variety of development language choices
- The familiar development experience with the Windows Runtime
- WinRT functionality
- Speed and fluidity
- Building hybrid apps

Technologies discussed:

Windows 8, Windows Runtime, C#, Visual Basic, C++, JavaScript

The Windows Runtime Enables Language Choice

There are lots of app developers out there. They range from developers using JavaScript/HTML/CSS to create Web apps, to developers using C#/Visual Basic to create apps with the Microsoft .NET Framework, to native developers creating apps with C++. The Windows Runtime enables Windows Store apps to be written in any of these programming languages, empowering all of these developers to develop great Windows apps, as shown in **Figure 1**.

This is beneficial to both developers and consumers. There's a huge market opportunity for developers to make money by developing Windows Store apps. And with the support for all major Microsoft languages, a huge number of developers are available to create the great Windows apps consumers want to buy and use.

As a developer, you have existing skills and experience in your particular programming language of choice. You also have a lot of existing assets you use when developing apps (such as existing code, build infrastructure and so on). You shouldn't be required to learn an entirely new programming language and toolset simply to develop Windows Store apps on

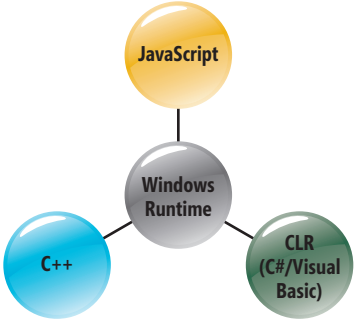


Figure 1 **The Windows Runtime Enables New Windows Store Apps to Be Written in Many Languages**

Figure 2 **The Previous Purely Native Platform Presented Real Usability Problems**

```
[DllImport("avicap32.dll", EntryPoint = "capCreateCaptureWindow")]
static extern int capCreateCaptureWindow(
  string lpszWindowName, int dwStyle,
  int X, int Y, int nWidth, int nHeight,
  System.IntPtr hwndParent, int nID);

[DllImport("avicap32.dll")]
static extern bool capGetDriverDescription(
  int wDriverIndex,
  [MarshalAs(UnmanagedType.LPTStr)] ref string lpszName,
  int cbName,
  [MarshalAs(UnmanagedType.LPTStr)] ref string lpszVer,
  int cbVer);
```

Windows 8 when there's already a rich set of programming languages and development tools provided by Microsoft.

The Windows Runtime Is Natural and Familiar

The previous Windows development experience was designed when C was the dominant programming language, when exception-based code was uncommon and when being bound to a single language was acceptable for a Windows OS. Today's modern developer world is quite different. Programming features such as namespaces, collections, events, asynchrony and so on aren't just common today, they're expected.

Let's take a look at using one of the more common devices on Windows today: the webcam. This is how part of the webcam API was previously declared in Windows:

```
HWND VFWAPI capCreateCaptureWindow(
  LPCTSTR lpszWindowName,
  DWORD dwStyle,
  int x,
  int y,
  nit nWidth,
  int nHeight,
  HWND hWnd,
  int nID
);
```

Previous Windows APIs were primarily declared in a native header file (*.h) that shipped as part of the Windows SDK. The SDK also included a .lib file against which code using the API was linked. Windows APIs were implemented in C/C++ as native flat-C functions or as native COM components and packaged in a .dll file shipped as part of Windows. The programming model was designed natively with performance in mind.

There were no namespaces, modern collections or true exceptions. Even for classic native developers the development experience wasn't ideal, as there were problems with IntelliSense, code browsing, lack of namespaces and more.

This purely native platform presented real usability problems as well for .NET developers trying to use Windows APIs, as shown in **Figure 2**.

Somebody needed to write a bunch of code to manually fill that gap between your code and traditional Windows APIs. If nobody had, you'd be left trying to write this complicated code yourself. Many resources for developers are aimed specifically at addressing this gap, including pinvoke.net, the Vista Bridge sample library (archive.msdn.microsoft.com/VistaBridge) and the Windows API Code Pack for Microsoft .NET Framework (archive.msdn.microsoft.com/WindowsAPICodePack).

Now, with Windows 8, you can use Windows APIs out of the box the day Windows ships, the exact same way you use other APIs in your apps today (such as in the .NET Framework for C#/Visual Basic developers). Code no longer needs to be written to fill the gap between your code and traditional Windows APIs. For example, using the webcam is much more straightforward today:

```
using Windows.Media.Capture;

MediaCapture captureMgr = new MediaCapture();
await captureMgr.InitializeAsync();

// Start capture preview; capturePreview
// is a CaptureElement defined in XAML
capturePreview.Source = captureMgr;
await captureMgr.StartPreviewAsync();
```

So how is using the Windows Runtime natural and familiar for developers? Because many apps retrieve data from the Internet, let's examine this question in the context of a common networking scenario: retrieving an RSS feed. Here's C# code to retrieve such a feed asynchronously:

```
var feedUri = new Uri("http://www.devhawk.com/rss.xml");
var client = new Windows.Web.Syndication.SyndicationClient();
var feed = await client.RetrieveFeedAsync(feedUri);
var title = feed.Title.Text;
```

Here's the equivalent Visual Basic code:

```
Dim feedUri = New Uri("http://www.devhawk.com/rss.xml");
Dim client = New Windows.Web.Syndication.SyndicationClient();
Dim feed = Await client.RetrieveFeedAsync(feedUri);
Dim title = feed.Title.Text;
```

Except for the root namespace (Windows), there's not much different in this code from code you already write with the .NET Framework. There are namespaces. There are classes and constructors. There are methods and properties. Identifiers are Pascal case because they're in the .NET Framework, and we also use the await keyword for asynchronous programming—everything a developer expects when writing code using C# and Visual Basic today.

Similar to the C#/Visual Basic development experience, C++ uses namespaces, classes, constructors, methods and properties. You can also use built-in C++ types such as strings (as opposed to previous Win32 data types). Standard operators such as "::" and "->" and others are also used as expected. And for asynchronous programming, we provide a Concurrency Runtime-like experience. Here's C++ code to retrieve an RSS feed asynchronously:

```
auto feedUri = ref new Windows::Foundation::Uri("http://www.devhawk.com/rss.xml");
auto client = ref new Windows::Web::Syndication::SyndicationClient();

task<SyndicationFeed^> retrieveTask(client->RetrieveFeedAsync(feedUri));
retrieveTask.then([this] (SyndicationFeed^ feed) {
  this->title = feed->Title->Text;
});
```

Finally, there's JavaScript, which shows some interesting differences in equivalent code:

```
var title;
var feedUri = new Windows.Foundation.Uri("http://www.devhawk.com/rss.xml");
var client = new Windows.Web.Syndication.SyndicationClient();

client.retrieveFeedAsync(feedUri).done(function (feed) {
  title = feed.title.text;
});
```

You'll notice in JavaScript that all methods and properties are camel case because it's natural and familiar to JavaScript developers. We're also using a promises framework with anonymous functions when writing asynchronous code. We continue to leverage the built-in types of the language here as well.

The developer experience is also familiar, with tooling features that developers expect, such as IntelliSense and functioning standard tools such as ILDASM and .NET Reflector (see **Figure 3**).

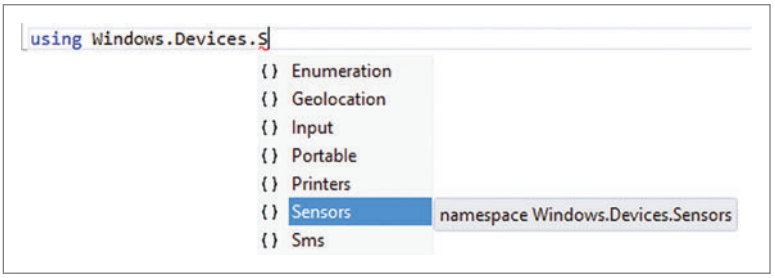


Figure 3 **Visual Studio IntelliSense for WinRT APIs**

The Windows Runtime enables this natural and familiar experience by extending and combining foundational concepts from .NET and COM to add modern semantics to WinRT APIs, such as classes, constructors, statics and events. These semantics can be exposed in different ways from different languages, all depending on what's natural and familiar to developers in that language.

This is possible because all WinRT APIs include metadata that programming languages use to know how that API is exposed via the Windows Runtime. WinRT metadata files use an updated version of the .NET metadata format. This metadata is available to every developer because it's installed on every single Windows 8 machine.

By reimagining the Windows API development surface, it doesn't feel like learning or using a new platform. It doesn't feel "alien." It looks and feels like code you've been writing. You're leveraging the existing skills and assets you have when creating new Windows Store apps.

The Windows Runtime Is Rich in Functionality

The Windows Runtime has fewer APIs than Win32, so it's easy to learn, yet it's rich in functionality. The richness of the Windows Runtime is too much to cover fully in an article of this size, but both the standard Windows features and the new Windows 8 features you expect to be there are there, as shown in **Figure 4**.

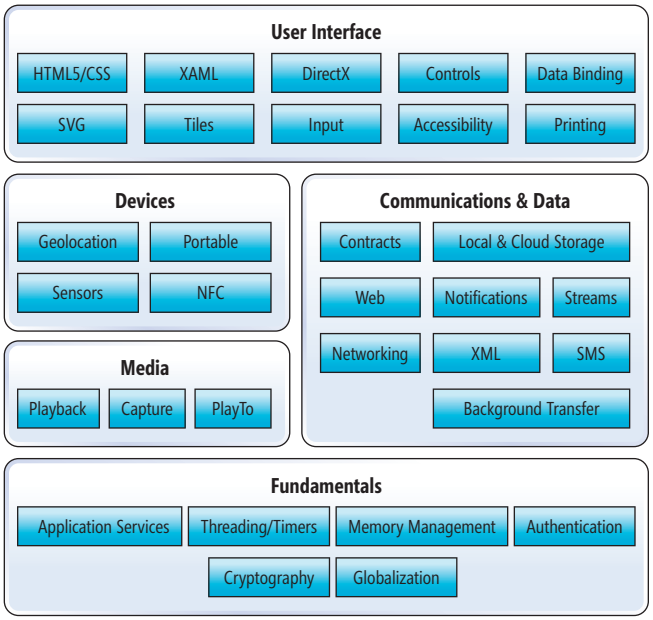


Figure 4 **The Windows Runtime Includes Standard Windows Features and New Windows 8 Features**

And the Windows Runtime is consistently designed through the use of a single unified set of API design guidelines. You learn the concepts and design once with a single API, and then you can apply that knowledge when using different APIs across the entire Windows API surface.

On the Windows Dev Center (dev.windows.com), you'll find many samples that cover most of these areas. Following are some examples.

Contracts You can let users search within your app when they select the Search charm and display suggestions in the Search pane. Your app can also share content with another app via the Share charm. Here are some examples:

- Search contract sample: bit.ly/QxsLQk
- Sharing content source app sample: bit.ly/H10Nd0
- Sharing content target app sample: bit.ly/HeuMOL

Social Using classes from the Windows.ApplicationModel.Contacts namespace, you can launch the Contact Picker and acquire contacts. For apps that need to supply contacts to other apps, you can use the ContactPickerUI class to create a collection of contacts:

- Contact Picker app sample: bit.ly/OFsebV

Media From simple media playback and media capture, to transcoding of media and playing media to external devices via the Play To contract, you'll find Windows APIs in the Windows.Media namespace:

- XAML media playback sample: bit.ly/RPIk4q
- HTML media playback sample: bit.ly/NwMjPL
- Media Play To sample: bit.ly/OYlZTg
- Transcoding media sample: bit.ly/NevTxK
- Media capture using capture device sample: bit.ly/HHb76Z

Security You can retrieve credentials, which can then be passed to APIs that might require credentials (for example, to support single sign-on). You can also use password-based strong encryption to securely store private information on a local computer to protect credit-card accounts, bank accounts and software products. You can even perform many different forms of encryption using the Windows.Security.Cryptography namespace:

- Credential picker sample: bit.ly/OUr8LC
- CryptoWinRT sample: bit.ly/OUrdif
- Secret saver encryption sample: bit.ly/Ms4ZGe

Networking/Web Whether you're connecting to the network via TCP using the StreamSocket class, connecting via User Datagram Protocol (UDP) using the DatagramSocket, or simply querying for networking information status to figure out whether you can save some application data to a Web service or need to save it locally, you can find what you need in the Windows.Networking namespace:

- StreamSocket sample: bit.ly/qL4c9D
- HttpClient sample: bit.ly/LIqcGH
- Network information sample: bit.ly/QxxDoK

This is not all the functionality in the Windows Runtime, of course. There's much more functionality available to you as a Windows developer, and you can investigate further at the Windows Dev Center at bit.ly/yyJI2J.

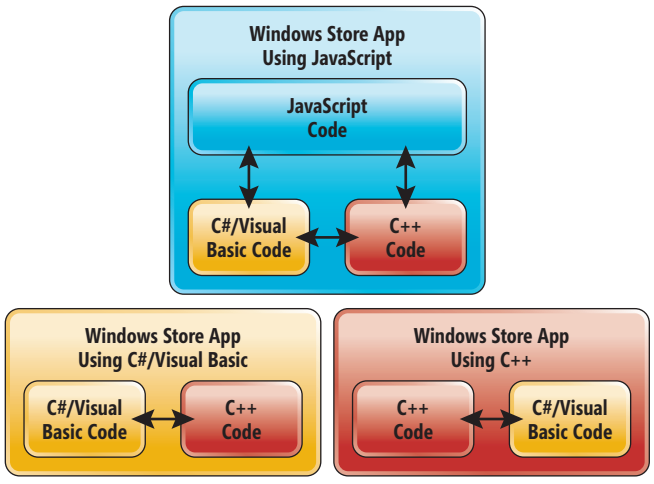


Figure 5 **The Windows Runtime Enables Hybrid Apps**

The Windows Runtime Is Fast and Fluid (Native and Async)

The Windows Runtime is implemented at the lowest level as native objects with a binary contract, but it's exposed to the world through metadata, a common type system and shared idioms. WinRT APIs continue to provide the fast performance capabilities of previous Windows APIs while also providing a modern development experience, as I demonstrated earlier.

For apps to remain fluid and responsive is a different challenge. We human beings instinctively multitask, which directly impacts how we expect apps to respond to us. We expect apps to be responsive to all interactions. When we use our favorite news-reading app, we want to add news feeds, read news articles, save news articles and so on. And we should be able to do all of these things even when the app is retrieving the latest articles from the Internet.

Figure 6 **XAudioWrapper.h**

```
#pragma once

#include "mmreg.h"
#include <vector>
#include <memory>

namespace XAudioWrapper
{
  public ref class XAudio2SoundPlayer sealed
  {
    public:
      XAudio2SoundPlayer(uint32 sampleRate);
      virtual ~XAudio2SoundPlayer();

      void Initialize();

      bool   PlaySound(size_t index);
      bool   StopSound(size_t index);
      bool   IsSoundPlaying(size_t index);
      size_t GetSoundCount();

      void Suspend();
      void Resume();

    private:
      interface IXAudio2*               m_audioEngine;
      interface IXAudio2MasteringVoice* m_masteringVoice;
      std::vector<std::shared_ptr<ImplData>>  m_soundList;
    };
}
```

This becomes especially important when we're interacting with apps using touch. We notice when the app doesn't "stick" to our finger. Even minor performance problems can degrade our experience and break the fast and fluid feeling.

Many modern apps connect to social Web sites, store data in the cloud, work with files on the hard disk, communicate with other gadgets and devices, and so on. Some of these sources have unpredictable latencies, which makes creating fast and fluid apps challenging. Unless built correctly, apps spend more time waiting for the outside environment and less time responding to the user's needs.

Addressing this connected world is a core principle of the Windows Runtime. You, the developer, should fall into "The Pit of Success" (see bit.ly/NjYMXM) when creating apps that are connected to the world. To achieve this, potentially I/O-bound or long-running APIs are asynchronous in the Windows Runtime. These are the most likely candidates to visibly degrade performance if written synchronously (for example, they likely could take longer than 50 ms to execute). This asynchronous approach to APIs sets you up to write code that's fast and fluid by default and promotes the importance of app responsiveness when developing Windows Store apps.

To understand more about the asynchronous nature of the Windows Runtime, you can read the blog post, "Keeping apps fast and fluid with asynchrony in the Windows Runtime," at bit.ly/GBLQLr.

The Windows Runtime Enables Hybrid Apps

A powerful feature of the Windows Runtime is that you aren't restricted by the programming language with which you choose to create your app. You aren't limited to just the libraries and code available to that programming language environment. You can use the language, library or component that's best suited for the job. It might be an open source library.

Figure 7 **XAudioWrapper.cpp**

```
XAudio2SoundPlayer::XAudio2SoundPlayer(uint32 sampleRate) :
  m_soundList()
{
  ...

  XAudio2Create(&m_audioEngine, flags);

  // Create the mastering voice
  m_audioEngine->CreateMasteringVoice(
    &m_masteringVoice,
    XAUDIO2_DEFAULT_CHANNELS,
    sampleRate
  );
}

bool XAudio2SoundPlayer::PlaySound(size_t index)
{
  //
  // Setup buffer
  //
  XAUDIO2_BUFFER playBuffer = { 0 };
  std::shared_ptr<ImplData> soundData = m_soundList[index];
  playBuffer.AudioBytes = soundData->playData->Length;
  playBuffer.pAudioData = soundData->playData->Data;
  playBuffer.Flags = XAUDIO2_END_OF_STREAM;

  HRESULT hr = soundData->sourceVoice->Stop();
  if (SUCCEEDED(hr))
  {
    ...
  }

  ...
}
```

Figure 8 **MainPage.cs**

```
using XAudioWrapper;

namespace BasicSoundApp
{
  public sealed partial class MainPage : Page
  {
    XAudio2SoundPlayer _audioPlayer = new XAudio2SoundPlayer(48000);
    public MainPage()
    {
      this.InitializeComponent();
    }

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
      _audioPlayer.Initialize();
    }

    private void Button_Click_1(object sender, RoutedEventArgs e)
    {
      _audioPlayer.PlaySound(0);
    }
  }
}
```

What if you're writing a game with JavaScript/HTML/CSS and you want a very fast physics library? You can use Box2D in C++. Writing a Windows Store app with JavaScript and want to work on some zipped files? You can use the zip functionality in the .NET Framework in a straightforward way. Creating a new Windows Store music app in C# and want to do some lower-level audio programming? No problem, just use Xaudio or WASAPI in C++. These capabilities are illustrated in **Figure 5**.

Let's look at an example software synthesizer app using XAML written in C#. I want to add some cool filter support to it, so I'm going to leverage Xaudio to have direct control of audio buffers. I use C++/CX to build the APIs that will be exposed to the C# project.

First, I create a simple WinRT wrapper around the Xaudio functionality I want to expose (see **Figure 6**).

First, note the usage of the "public," "ref" and "sealed" keywords in the class declaration. This is simply how you declare a public WinRT class in C++/CX. This will allow you to use this class from other languages such as JavaScript, C# or Visual Basic. It's important to note that this is not managed C++ or C++/Common Language Infrastructure (CLI). This is not compiling down to Microsoft intermediate language; this is a native component through and through.

You'll also notice that the public functionality (methods, properties and so on) of the class is limited to C++ built-in types or WinRT types. Those are the only types allowed for crossing the language boundary in WinRT components. However, you can use existing C++ libraries (for example, the Standard Template Library) as much as you wish in the implementation of your WinRT components.

Resources

For more information, check out the following resources:

- Windows Store app samples: **dev.windows.com**
- "Keeping apps fast and fluid with asynchrony in the Windows Runtime": **bit.ly/GBLQLr**
- "Creating Windows Runtime Components in C# and Visual Basic": **bit.ly/OWDe2A**
- "Creating Windows Runtime Components in C++": **bit.ly/TbgWz7**
- API reference for Windows Store apps: **bit.ly/zdFRK2**

Frequently Asked Questions

Q. Is the Windows Runtime a replacement for the CLR or the Microsoft .NET Framework?

A. No. The Windows Runtime does not replace the .NET Framework or any other frameworks. When you build a Windows Store app in C#, your code is still executing using the CLR. Not only that, but a subset of the .NET Framework (as well as Win32 and COM) is available for you to use when building your Windows Store apps.

Q. So when writing Windows Store apps in C++, I'm using C++/CLI?

A. No. Code using the Windows Runtime for Windows Store apps is written using C++/CX. Though it may look like C++/CLI at first, it is truly native. You won't introduce garbage collection or other C++/CLI features into your native app.

For more details on creating WinRT components in C++, you can read the Windows Dev Center topic, "Creating Windows Runtime Components in C++," at **bit.ly/TbgWz7**.

Now that I've defined the basic interface for my class, let's take a quick look at some of the implemented methods (the details aren't important) as shown in **Figure 7**.

As you'll notice from the code snippet in **Figure 7**, I'm using the Xaudio2 COM APIs available in the Windows SDK for Windows Store apps to wire up the audio engine. Additionally, I'm using C++ constructs and types beyond just the WinRT types to implement the necessary functionality.

Once I've defined and implemented my basic class, I can simply add a reference to my C++ project from my C# project. As a result, the class that's exposed from my C++ project becomes available to my C# project (see **Figure 8**).

You can see that, though the XAudioWrapper was written in native C++, it can be used as if it were a regular .NET class. I reference its namespace, instantiate the component and start invoking the various methods it exposes—all without requiring any DllImports to call into the native code!

There are no boundaries. As a JavaScript developer, you aren't limited to JavaScript libraries. As a C#/Visual Basic developer, you aren't limited to .NET libraries. And as a C++ developer, you aren't limited to C/C++ libraries. Will you need to create your own components often? Perhaps not. But the option is available to you.

To sum up, the Windows Runtime is at the heart of creating Windows Store apps on Windows 8. It provides a powerful platform on which to create Windows Store apps. The Windows Runtime provides a development surface that has a consistent and thoughtful design and is rich in functionality. And whether you're a JavaScript developer, C#/Visual Basic developer or a C++ developer, you can now be a Windows developer creating new Windows Store apps for Windows 8. ■

JASON OLSON *is a senior program manager working on the Windows Runtime at Microsoft. When he's not working on Windows, he can be found playing piano around the Seattle area and spending time with his wife and two boys.*

THANKS to the following technical experts for reviewing this article: *Noel Cross, Anantha Kancherla, Ines Khelifi, John Lam, Martyn Lovell, Harry Pierson, Mahesh Prakriya and Steve Rowe*

LEADTOOLS®

THE WORLD LEADER IN IMAGING SDKs

LEADTOOLS provides developers easy access to decades of expertise in developing color, grayscale, document, medical, vector and multimedia imaging technology. Develop with LEADTOOLS to eliminate months of research and development and add essential functionality to your application without sacrificing quality or performance.

OCR	BARCODE	PDF & PDF/A	FORMS RECOGNITION & PROCESSING	150 + FILE FORMATS
DOCUMENT PREPROCESSING		ANNOTATIONS		SCANNING
DICOM & PACS	MEDICAL WORKSTATION		IMAGE PROCESSING	VIRTUAL PRINTER
MPEG-2 TRANSPORT		MULTIMEDIA PLAYBACK & CAPTURE	DVD & DVR	CODECS

ZERO FOOTPRINT AND RICH CLIENT CONTROLS FOR DOCUMENT & MEDICAL

C++ .NET WinRT HTML5
WEB SERVICES, WPF, ASP.NET & CLOUD

DOWNLOAD OUR 60 DAY EVALUATION
WWW.LEADTOOLS.COM

LEAD TECHNOLOGIES INCORPORATED

SALES@LEADTOOLS.COM
800.637.1840

VSP² VISUAL STUDIO PARTNER PROFILE

GREG LUTZ
PRODUCT MANAGER, DEV TOOLS
COMPONENTONE, A DIVISION OF GRAPECITY

Expanding Your Visual Studio 2012 Toolbox

Developer Tools for WinRT XAML and JavaScript

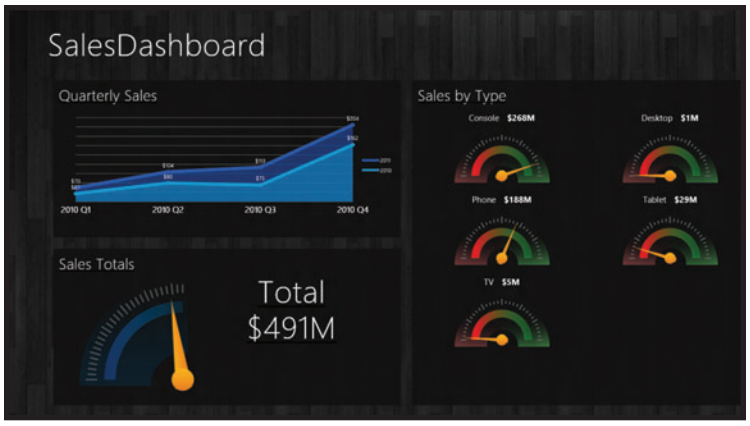
Windows 8 opens up a whole new market for developers who want to create applications for commercial and mobile users. Opportunities to innovate, fill in gaps, and add features that make development easier and more powerful arise each time Microsoft introduces a new application platform or technology. Rather than developers having to write their own controls, like charts, calendars, and datagrids, they may access reusable ones from Microsoft Visual Studio Partners like ComponentOne. Developers have come to expect quality controls that boost productivity from ComponentOne; the company plans to meet their expectations by creating some of the first WinRT XAML and JavaScript controls for Modern UI-style applications. This article provides an overview of these latest Windows 8 development tools.

Filling the Gap

Most Windows 8 developers will need to display or gather DateTime values somewhere in their apps. The standard control libraries do not include any calendars or DateTime editors. This is where ComponentOne Studio for WinRT XAML becomes absolutely necessary, as it gives you seven globalized DateTime and calendar controls that work out of the box. This makes developers lives a lot easier by not having to reinvent the wheel, so to speak.

Making Your Apps More Powerful

The tablet nature of Windows 8 will inspire a lot of "touch-first" dashboard apps, and most agree that great dashboards need some form of data visualization. Charts and gauges can be very complex controls to write from scratch, so ComponentOne is taking advantage of their existing XAML and JavaScript libraries to bring developers full-featured and powerful controls right out of the gate. It's not typical to see such advanced controls so early in a platforms life, but the similarities between existing platforms and WinRT are so great



that it makes sense to share code and deliver similar controls. It means developers get familiar control APIs across multiple platforms, as well as a clearer migration path when the next UI platform of tomorrow arrives.

Windows 8 developers need access to charts and gauges to build powerful dashboards in record time. Using out-of-the-box controls is as easy as binding to a data set and setting a few properties to customize the appearance. The ComponentOne chart controls support many chart types like area, line, scatter, bar, pie, and financial. They have been modified for a touch environment with new gesture-based manipulation modes. Users can slide, pinch, and tap a plot for added interactivity. The gauge controls include various modern looking linear and radial designs. And like all of the other desktop and web controls, the WinRT ones also support a common theming technology that will make styling controls a quick and painless process.

Innovating

Enabling innovative, cutting-edge experiences are key factors to being successful. ComponentOne plans to introduce several new live tile-inspired controls that make it easier to create a Modern UI style around typical business objects. There are also plans for ComponentOne to support many of their innovative controls, such as the popular PdfViewer and RichTextBox. These controls enable users to display PDF and edit HTML content entirely in XAML.

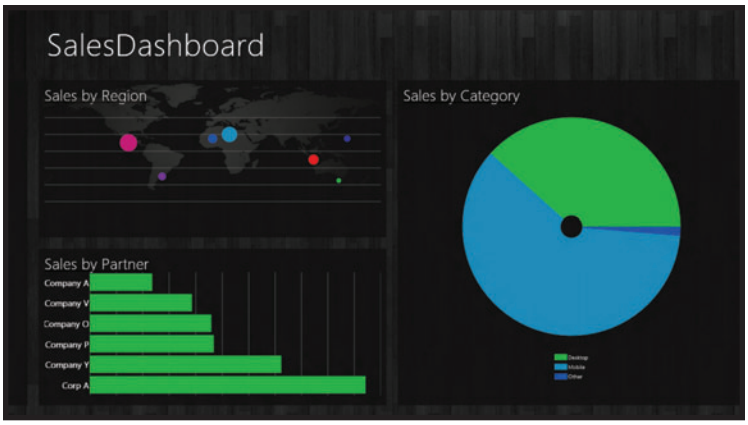
Best known for their grids on the desktop and web platforms, ComponentOne plans to bring that success to Windows 8. While standard in most platforms, ComponentOne will be re-imagining their data grids for a Modern UI with their GridView for JavaScript developers and FlexGrid for XAML developers. Grids such as these enable powerful, tabular data display and analysis that developers won't get out of the box with the standard controls.

One of the big issues in moving from the desktop to mobile environments, like Windows Phone and Windows 8, is performance. Performance is a top consideration for ComponentOne, so many of the controls root level architecture has been revamped for a flawless, fast, and fluid experience. New paradigms for user interaction, like drag and drop, have also been re-invented for the environment.

Strategizing for the Future

The excitement level is high with Windows 8 and with this may come anxiety as every developer determines how their development path will adapt or change. Companies like ComponentOne offer developers a sense of silent confidence as they navigate this new era of development. The strategy from ComponentOne is to have code-compatible libraries from Silverlight, WPF, or Windows Phone to WinRT and from ASP.NET Web Forms or MVC to WinRT. This will position the company to be the only component vendor that developers need for all of their development efforts today and tomorrow.

To learn more about ComponentOne and to gain access to free trial downloads and samples with source code, visit http://c1.ms/winrt



Under the Hood with .NET and the Windows Runtime

Shawn Farkas

The Windows Runtime (WinRT) provides a large set of new APIs to Windows Experience developers. The CLR 4.5, which ships as part of the Microsoft .NET Framework 4.5 in Windows 8, enables developers writing managed code to use the APIs in a natural way, just as though they were another class library. You can add a reference to the Windows Metadata (WinMD) file that defines the APIs you want to call and then call into them just as with a standard managed API. Visual Studio automatically adds a reference to the built-in WinRT APIs to new Windows UI projects, so your app can simply begin using this new API set.

Under the hood, the CLR provides the infrastructure for managed code to consume WinMD files and transition between managed code and the Windows Runtime. In this article, I'll show some of these details. You'll come away with a better understanding of what occurs behind the scenes when your managed program calls a WinRT API.

This article discusses:

- Windows Runtime metadata
- CLR metadata adapter
- Runtime callable wrappers and COM callable wrappers
- CLR marshaling stubs
- .NET extension methods for Windows Runtime

Technologies discussed:

XAML, Windows Runtime, CLR, Microsoft .NET Framework

Consuming WinMD Files from Managed Code

WinRT APIs are defined in WinMD files, which are encoded using the file format described in ECMA-335 (bit.ly/sLlLl). Although WinMD files and .NET Framework assemblies share a common encoding, they're not the same. One of the main differences in the metadata stems from the fact that the WinRT type system is independent of the .NET type system.

Programs such as the C# compiler and Visual Studio use the CLR metadata APIs (such as IMetaDataImport) to read .NET Framework assembly metadata and can now read metadata from WinMD files as well. Because the metadata is not exactly the same as a .NET assembly, the CLR metadata reader inserts an adapter between the metadata APIs and the WinMD file being read. This enables WinMD files to be read as though they were .NET assemblies (see **Figure 1**).

Running ILDasm helps you understand the modifications that the CLR metadata adapter performs on a WinMD file. By default, ILDasm shows the contents of a WinMD file in its raw form, as it's encoded on disk without the CLR metadata adapter. However, if you pass ILDasm the /project command-line parameter, it enables the metadata adapter, and you can see the metadata as the CLR and managed tools will read it.

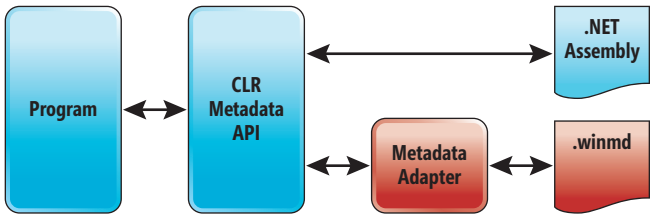


Figure 1 **The CLR Inserts a Metadata Adapter Between WinMD Files and the Public Metadata Interface**

By running copies of ILDasm side by side—one with the /project parameter and one without—you can easily explore the changes that the CLR metadata adapter makes to a WinMD file.

The Windows Runtime and .NET Type Systems

One of the major operations the metadata adapter performs is to merge the WinRT and .NET type systems. At a high level, five different categories of WinRT types can appear in a WinMD file and need to be considered by the CLR. These are listed in **Figure 2**. Let's look at each category in more detail.

Standard WinRT Types While the CLR has special support for many categories of types exposed by the Windows Runtime, the vast majority of WinRT types are not treated specially by the CLR at all. Instead, these types appear to .NET developers unmodified, and they can be used as a large class library to enable writing Windows Store applications.

Primitive Types This set of primitive types is encoded into a WinMD file using the same ELEMENT_TYPE enumeration that .NET assemblies use. The CLR automatically interprets these primitive types as though they were the .NET equivalents.

For the most part, treating WinRT primitive types as .NET primitive types just works. For instance, a 32-bit integer has the same bit pattern in the Windows Runtime as it does in .NET, so the CLR can treat a WinRT DWORD as a .NET System.Int32 without any trouble. But two notable exceptions are strings and objects.

In the Windows Runtime, strings are represented with the HSTRING data type, which is not the same as a .NET System.String. Similarly, ELEMENT_TYPE_OBJECT means System.Object to .NET, while it means IInspectable* to the Windows Runtime. For both strings and objects, the CLR needs to marshal objects at run time to convert between the WinRT and .NET representations of the types. You'll see how this marshaling works later in this article.

Projected Types There are some existing fundamental .NET types that have equivalents in the WinRT type system. For example, the Windows Runtime defines a TimeSpan structure and a Uri class, both of which have corresponding types in the .NET Framework.

To avoid forcing .NET developers to convert back and forth between these fundamental data types, the CLR projects the WinRT version to its .NET equivalent. These projections are effectively merge points that the CLR inserts between the .NET and WinRT type systems.

For example, the Windows Runtime SyndicationClient.RetrieveFeedAsync API takes a WinRT Uri as its parameter. Instead of requiring .NET developers to manually create a new Windows.Foundation.Uri instance to pass to this API, the CLR projects the type as a System.Uri, which lets .NET developers use the type they're more familiar with.

Another example of a projection is the Windows.Foundation.HResult structure, which is projected by the CLR to the System.Exception type. In .NET, developers are used to seeing error information conveyed as an exception rather than as a failure HRESULT, so having a WinRT API such as IAsycnInfo.ErrorCode express error information as an HResult structure won't feel natural. Instead, the CLR projects HResult to Exception, which makes a WinRT API such as IAsyncInfo.ErrorCode more usable for .NET developers. Here's an example of the IAsyncInfo ErrorCode property encoded in Windows.winmd:

```
.class interface public windowsruntime IAsyncInfo
{
  .method public abstract virtual
    instance valuetype Windows.Foundation.HResult
    get_ErrorCode()
}
```

And here's the IAsyncInfo ErrorCode property after the CLR projection:

```
.class interface public windowsruntime IAsyncInfo
{
 .method public abstract virtual
   instance class [System.Runtime]System.Exception
   get_ErrorCode()
}
```

Projected Interfaces The Windows Runtime also provides a set of fundamental interfaces that have .NET equivalents. The CLR performs type projections on these interfaces as well, again merging the type systems at these fundamental points.

The most common examples of projected interfaces are the WinRT collection interfaces, such as IVector<T>, IIterable<T> and IMap<K,V>. Developers who use .NET are familiar with collection interfaces such as IList<T>, IEnumerable<T> and IDictionary<K,V>. The CLR projects the WinRT collection interfaces to their .NET equivalents and also hides the WinRT interfaces so that developers don't have to deal with two functionally equivalent sets of types that do the same thing.

In addition to projecting these types when they appear as parameters and return types of methods, the CLR also must project these interfaces when they appear in the interface implementation list of a type. For example, the WinRT PropertySet type implements the WinRT IMap<string, object> interface. The CLR, however, will project PropertySet as a type that implements IDictionary<string, object>. When performing this projection, the members of PropertySet that are used to implement IMap<string, object> are hidden.

Figure 2 **WinRT Types for a WinMD File**

Category	Examples
Standard WinRT types	Windows.Foundation.Collections.PropertySet, Windows.Networking.Sockets.DatagramSocket
Primitive types	Byte, Int32, String, Object
Projected types	Windows.Foundation.Uri, Windows.Foundation.DateTime
Projected interfaces	Windows.Foundation.Collections.IVector<T>, Windows.Foundation.IClosable
Types with .NET helpers	Windows.Storage.Streams.IInputStream, Windows.Foundation.IasyncInfo

Instead, .NET developers access PropertySet through corresponding IDictionary<string, object> methods. Here's a partial view of PropertySet as encoded in Windows.winmd:

```
.class public windowsruntime PropertySet
  implements IPropertySet,
    class IMap`2<string,object>,
    class IIterable`1<class
IKeyValuePair`2<string,object> >
{
  .method public instance uint32 get_Size()
  {
    .override  method instance uint32 class
IMap`2<string,object>::get_Size()
  }
}
```

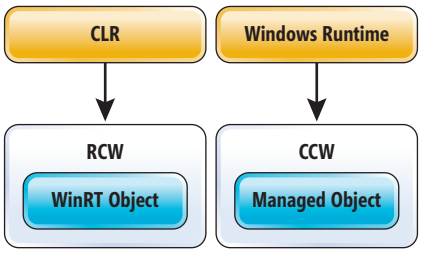


Figure 3 **Using Wrappers for Windows Runtime and Managed Objects**

And here's a partial view of PropertySet after the CLR projection:

```
.class public windowsruntime PropertySet
  implements IPropertySet,
    class IDictionary`2<string,object>,
    class IEnumerable`1<class KeyValuePair`2<string,object> >
{
  .method private instance uint32 get_Size()
  {
  }
}
```

Notice that three type projections occur: IMap<string,object> to IDictionary<string,object>, IKeyValuePair<string,object> to KeyValuePair<string, object>, and IIterable<IKeyValuePair> to IEnumerable<KeyValuePair>. Also notice that the get_Size method from IMap is hidden.

Types with .NET Framework Helpers The Windows Runtime has several types that don't have a full merge point into the .NET type system but are important enough to most applications that the .NET Framework provides helper methods to work with them. Two of the best examples are the WinRT stream and async interfaces.

Although the CLR does not project Windows.Storage.Streams.IRandomAccessStream to System.Stream, it does provide a set of extension methods for IRandomAccessStream that allows your code to treat these WinRT streams as though they were .NET streams. For example, you can easily read a WinRT stream with the .NET StreamReader by calling the OpenStreamForReadAsync extension method.

The Windows Runtime provides a set of interfaces representing asynchronous operations, such as the IAsyncInfo interface. In the .NET Framework 4.5, there's built-in support for awaiting asynchronous operations, which developers want to use with WinRT APIs in the same way they do for .NET APIs.

To enable this, the .NET Framework ships with a set of GetAwaiter extension methods for the WinRT async interfaces. These methods are used by the C# and Visual Basic compilers to enable awaiting WinRT asynchronous operations. Here's an example:

```
private async Task<string> ReadFilesync(StorageFolder parentFolder,
string fileName)
{
  using (Stream stream = await parentFolder.OpenStreamForReadAsync(fileName))
  using (StreamReader reader = new StreamReader(stream))
  {
    return await reader.ReadToEndAsync();
  }
}
```

Transitioning Between the .NET Framework and the Windows Runtime The CLR provides a mechanism for managed code to seamlessly call WinRT APIs, and for the Windows Runtime to call back into managed code.

At its lowest level, the Windows Runtime is built on top of COM concepts, so it's no surprise that the CLR support for calling WinRT APIs is built on top of existing COM interop infrastructure.

One important difference between WinRT interop and COM interop is how much less configuration you have to deal with in the Windows Runtime. WinMD files have rich metadata describing all of the APIs they're exposing with a well-defined mapping to the .NET type system, so there's no need to use any MarshalAs attributes in managed code. Similarly, because Windows 8 ships with WinMD files for its WinRT APIs, you don't need to have a primary interop assembly bundled with your application. Instead, the CLR uses the in-box WinMD files to figure out everything it needs to know about how to call WinRT APIs.

These WinMD files provide the managed type definitions that are used at run time to allow managed developers access to the Windows Runtime. Although the APIs that the CLR reads out of a WinMD file contain a method definition that's formatted to be easily used from managed code, the underlying WinRT API uses a different API signature (sometimes referred to as the application binary interface, or ABI, signature). One example of a difference between the API and ABI signatures is that, like standard COM, WinRT APIs return

Figure 4 **Example of a Marshaling Stub for Making a Call from the CLR to the Windows Runtime**

```
public string Join(IEnumerable<string> list, string separator)
{
  // Convert the managed parameters to WinRT types
  CCW ccwList = GetCCW(list);
  IIterable<HSTRING>* pCcwIterable = ccwList.QueryInterface(IID_IIterable_HSTRING);
  HSTRING hstringSeparator = StringToHString(separator);

  // The object that managed code calls is actually an RCW
  RCW rcw = this;

  // You need to find the WinRT interface pointer for IConcatenation
  // implemented by the RCW in order to call its Join method
  IConcatination* pConcat = null;
  HRESULT hrQI = rcw.QueryInterface(IID_ IConcatenation, out pConcat);
  if (FAILED(hrQI))
    {
      // Most frequently this is an InvalidCastException due to the WinRT
      // object returning E_NOINTERFACE for the interface that contains
      // the method you're trying to call
      Exception qiError = GetExceptionForHR(hrQI);
      throw qiError;
    }

    // Call the real Join method
    HSTRING returnValue;
    HRESULT hrCall = pConcat->Join(pCcwIterable, hstringSeparator, &returnValue);

    // If the WinRT method fails, convert that failure to an exception
    if (FAILED(hrCall))
    {
      Exception callError = GetExceptionForHR(hrCall);
      throw callError;
    }

    // Convert the output parameters from WinRT types to .NET types
    return HStringToString(returnValue);
}
```

Figure 5 **Example of a Marshaling Stub for Making a Call from the Windows Runtime to the CLR**

```
HRESULT Join(__in IIterable<HSTRING>* list, HSTRING separator, __out HSTRING* retval)
{
  *retval = null;

  // The object that native code is calling is actually a CCW
  CCW ccw = GetCCWFromComPointer(this);

  // Convert the WinRT parameters to managed types
  RCW rcwList = GetRCW(list);
  IEnumerable<string> managedList = (IEnumerable<string>)rcwList;
  string managedSeparator = HStringToString(separator);

  string result;
  try
  {
    // Call the managed Join implementation
    result = ccw.Join(managedList, managedSeparator);
  }
  catch (Exception e)
  {
    // The managed implementation threw an exception -
    // return that as a failure HRESULT
    return GetHRForException(e);
  }

  // Convert the output value from a managed type to a WinRT type
  *retval = StringToHSTring(result);
  return S_OK;
}
```

HRESULTS, and the return value of a WinRT API is actually an output parameter in the ABI signature. I'll show an example of how a managed method signature is transformed into a WinRT ABI signature when I look at how the CLR calls a WinRT API later in this article.

Runtime Callable Wrappers and COM Callable Wrappers

When a WinRT object enters the CLR, it needs to be callable as if it were a .NET object. To make this happen, the CLR wraps each WinRT object in a runtime callable wrapper (RCW). The RCW is what managed code interacts with, and is the interface between your code and the WinRT object that your code is using.

Conversely, when managed objects are used from the Windows Runtime, they need to be called as if they were WinRT objects. In this case, managed objects are wrapped in a COM callable wrapper (CCW) when they're sent to the Windows Runtime. Because the Windows Runtime uses the same infrastructure as COM, it can interact with CCWs to access functionality on the managed object (see **Figure 3**).

Figure 6 **Conceptual Implementation of IDictionary in Terms of IMap**

```
bool TryGetValue(K key, out V value)
{
  // "this" is the IMap RCW
  IMap<K,V> rcw = this;

  // IMap provides a HasKey and Lookup function, so you can
  // implement TryGetValue in terms of those functions
  if (!rcw.HasKey(key))
    return false;

  value = rcw.Lookup(key);
  return true;
}
```

Marshaling Stubs

When managed code transitions across any interop boundary, including WinRT boundaries, several things must occur:

1. Convert managed input parameters into WinRT equivalents, including building CCWs for managed objects.
2. Find the interface that implements the WinRT method being called from the RCW that the method is being called on.
3. Call into the WinRT method.
4. Convert WinRT output parameters (including return values) into managed equivalents.
5. Convert any failure HRESULTS from the WinRT API into a managed exception.

These operations occur in a marshaling stub, which the CLR generates on your program's behalf. The marshaling stubs on an RCW are what managed code actually calls before transitioning into a WinRT API. Similarly, the Windows Runtime calls into CLR-generated marshaling stubs on a CCW when it transitions into managed code.

Marshaling stubs provide the bridge that spans the gap between the Windows Runtime and the .NET Framework. Understanding how they work will help you gain a deeper understanding of what happens when your program calls into the Windows Runtime.

An Example Call

Imagine a WinRT API that takes a list of strings and concatenates them, with a separator string between each element. This API might have a managed signature such as:

```
public string Join(IEnumerable<string> list, string separator)
```

The CLR has to call the method as it's defined in the ABI, so it needs to figure out the ABI signature of the method. Thankfully, a set of deterministic transformations can be applied to unambiguously get an ABI signature given an API signature. The first transformation is to replace the projected data types with their WinRT equivalents, which returns the API to the form in which it's defined in the WinMD file before the metadata adapter loaded it. In this case, IEnumerable<T> is actually a projection of IIterable<T>, so the WinMD view of this function is actually:

```
public string Join(IIterable<string> list, string separator)
```

WinRT strings are stored in an HSTRING data type, so to the Windows Runtime, this function actually looks like:

```
public HSTRING Join(IIterable<HSTRING> list, HSTRING separator)
```

At the ABI layer, where the call actually occurs, WinRT APIs have HRESULT return values, and the return value from their signature is an output parameter. Additionally, objects are pointers, so the ABI signature for this method would be:

```
HRESULT Join(__in IIterable<HSTRING>* list, HSTRING separator, __out HSTRING* retval)
```

All WinRT methods must be part of an interface that an object implements. Our Join method, for instance, might be part of an IConcatenation interface supported by a StringUtilities class. Before making a method call into Join, the CLR must get a hold of the IConcatenation interface pointer to make the call on.

The job of a marshaling stub is to convert from the original managed call on an RCW to the final WinRT call on a WinRT interface. In this case, the pseudo-code for the marshaling stub might look like **Figure 4** (with cleanup calls omitted for clarity).

In this example, the first step is to convert the managed parameters from their managed representation to their WinRT representation. In

this case, the code creates a CCW for the list parameter and converts the System.String parameter to an HSTRING.

The next step is to find the WinRT interface that supplies the implementation of Join. This occurs by issuing a QueryInterface call to the WinRT object that's wrapped by the RCW that the managed code called Join on. The most common reason an InvalidCastException gets thrown from a WinRT method call is if this QueryInterface call fails. One reason this can happen is that the WinRT object doesn't implement all the interfaces that the caller expected it to.

Now the real action occurs—the interop stub makes the actual call to the WinRT Join method, providing a location for it to store the logical return value HSTRING. If the WinRT method fails, it indicates this with a failure HRESULT, which the interop stub converts into an Exception and throws. This means that if your managed code sees an exception being thrown from a WinRT method call, it's likely that the WinRT method being called returned a failure HRESULT and the CLR threw an exception to indicate that failure state to your code.

The final step is to convert the output parameters from their WinRT representation to their .NET form. In this example, the logical return value is an output parameter of the Join call and needs to be converted from an HSTRING to a .NET String. This value can then be returned as the result of the stub.

Calling from the Windows Runtime into Managed Code

Calls that originate from the Windows Runtime and target managed code work in a similar manner. The CLR responds to the QueryInterface calls that the Windows Runtime Component makes against it with an interface that has a virtual function table that's filled out with interop stub methods. These stubs perform the same function as the one I showed previously, but in reverse direction.

Let's consider the case of the Join API again, except this time assume it's implemented in managed code and is being called into from a Windows Runtime Component. Pseudo-code for a stub that allows this transition to occur might look like **Figure 5**.

First, this code converts the input parameters from their WinRT data types into managed types. Assuming the input list is a WinRT object, the stub must get an RCW to represent that object to allow managed code to use it. The string value is simply converted from an HSTRING to a System.String.

Next, the call is made into the managed implementation of the Join method on the CCW. If this method throws an exception, the interop stub catches it and converts it to a failure HRESULT that's returned to the WinRT caller. This explains why some exceptions thrown from managed code called by Windows Runtime Components don't crash the process. If the Windows Runtime Component handles the failure HRESULT, that's effectively the same as catching and handling the thrown exception.

The final step is to convert the output parameter from its .NET data type to the equivalent WinRT data type, in this case converting the System.String to an HSTRING. The return value is then placed in the output parameter and a success HRESULT is returned.

Figure 7 **Call Stack for TryGetValue Call**

Stack	Description
PropertySet::HasKey	WinRT PropertySet implementation
HasKey_Stub	Marshaling stub converting the dictionary stub's HasKey call into a WinRT call
TryGetValue_Stub	Stub implementing IDictionary in terms of IMap
Application	Managed application code calling PropertySet.TryGetValue

Projected Interfaces

Earlier, I mentioned that the CLR will project some WinRT interfaces into equivalent .NET interfaces. For instance, IMap<K,V> is projected to IDictionary<K,V>. This means any WinRT map is accessible as a .NET dictionary, and vice versa. To enable this projection to work, another set of stubs is needed to implement the WinRT interface in terms of the .NET interface it's projected to, and vice versa. For example, IDictionary<K,V> has a TryGetValue method, but IMap<K,V> doesn't contain this method. To allow managed callers to use TryGetValue, the CLR provides a stub that implements this method in terms of methods that IMap does have. This might look similar to **Figure 6**.

> Any WinRT map is accessible as a .NET dictionary, and vice versa.

Notice that to do its work, this conversion stub makes several calls to the underlying IMap implementation. For instance, let's say you wrote the following bit of managed code to see whether a Windows.Foundation.Collections.PropertySet object contains the key "NYJ":

```
object value;
if (propertySet.TryGetValue("NYJ", out value))
{
  // ...
}
```

As the TryGetValue call determines whether the property set contains the key, the call stack might look like **Figure 7**.

Wrapping Up

The CLR support for the Windows Runtime allows managed developers to call WinRT APIs defined in WinMD files as easily as they can call managed APIs defined in a standard .NET assembly. Under the hood, the CLR uses a metadata adapter to perform projections that help to merge the WinRT type system with the .NET type system. It also uses a set of interop stubs to allow .NET code to call WinRT methods, and vice versa. Taken together, these techniques make it easy for managed developers to call WinRT APIs from their Windows Store applications. ■

SHAWN FARKAS *has worked on the CLR for 10 years and is currently the development lead in charge of CLR Windows Runtime projection and .NET interop. Prior to the Microsoft .NET Framework 4.5, he worked on the CLR security model. His blog can be found at blogs.msdn.com/shawnfa.*

THANKS to the following technical experts for reviewing this article: *Ryan Byington, Layla Driscoll and Yi Zhang*

Be a Mobile Superhero

TouchDraw for
iOS and Android

"I've built my mobile strategy on Xamarin and it was the best decision I made."

Jon Lipsky, Creator of TouchDraw

Build fully native iOS & Android apps in C#

Leverage existing skills, teams, code and tools. Xamarin is turning Visual Studio developers into native iOS and Android developers virtually overnight.

Native UI. Native performance. Complete access to the iOS and Android APIs in C#. Xamarin apps are compiled as native binaries, not interpreted, for brilliant app performance.

Broad device reach. Quickly. Accelerate time-to-market by sharing up to 90% code across iOS, Android and Windows apps.

Powerful IDE support. Use Visual Studio 2012 to design, develop, debug and deploy apps. Or use our own cross-platform IDE.

Join over 185,000 developers who are using Xamarin to build consumer, corporate and gaming apps.
Visit **xamarin.com/MSDN** to try it for free.

Xamarin

Visual Studio
Microsoft
Partner

Windows Runtime Components in a .NET World

Jeremy Likness

The new type of program known as a Windows Store app—optimized to run on Windows 8 devices—has a default view with a full-screen window and no distracting chrome, so the content is the focal point. Windows Store apps support fluid layouts that adapt and scale to a variety of screen sizes and resolutions. They provide a touch-first experience while providing full support for the traditional keyboard and mouse.

Windows Store apps run on a new set of APIs called the Windows Runtime (WinRT). The Windows Runtime exposes components that are built as part of the Windows 8 OS along with third-party components you can develop yourself. Although some core Windows Runtime Components are accessible from desktop apps, third-party Windows Runtime Components are only available from within the Windows 8 environment. WinRT types are described using WinRT metadata files that have the .winmd extension. These files are encoded using the same standard the Microsoft .NET Framework uses for providing metadata definitions and semantics for classes, ECMA-335 (see bit.ly/sLILl).

You can quickly navigate to the type definitions on a Windows 8 machine by changing to the directory that contains system files for Windows (usually c:\windows\system32). A folder within that directory called WinMetadata contains all of the type definitions. You can use the ILDasm.exe utility to explore the types. Open a Visual Studio 2012 command prompt, navigate to the c:\windows\system32\WinMetadata folder and type the following in the command line:

```
Ildasm.exe windows.web.winmd
```

You should see a result similar to **Figure 1**. You can use the ILDasm.exe utility to inspect all of the namespaces and types defined for that particular Windows Runtime Component.

What's interesting to note is that there's no code contained within the file; only metadata information is available. The component is part of the underlying OS. It's most likely written using native code. A unique feature called language projection allows Windows Runtime Components (both native and managed) to be accessed from any language that supports Windows Store app development.

This article discusses a prerelease version of Visual Studio 2012. All related information is subject to change.

This article discusses:

- Projection and mapping
- Requirements for creating WinRT types
- The sample app
- Consuming a component from C#, JavaScript and C++
- Alternatives to managed projects

Technologies discussed:

Windows Runtime, Windows 8

Code download available at:

archive.msdn.microsoft.com/mag201210WinRTComp

Projection and Mapping

Many languages, including C#, Visual Basic, C++ and JavaScript, have been updated with Windows 8 to support language projection. This allows Windows Runtime Components to be accessed in a natural way using multiple languages. Projection handles exposing a WinRT type as an object or class that's native to the language being used to develop the Windows Store app. The following code accesses a native Windows Runtime Component directly from a Windows Store app built using C#:

```
var ui = new CameraCaptureUI();
```

The CameraCaptureUI is a Windows Runtime Component. The component isn't a managed C# type, but it can be easily accessed and referenced from within C# code as if it were. This is because the CLR automatically generates a Runtime Callable Wrapper (RCW) for the Windows Runtime Component using its metadata and causes it to appear as a native CLR type to managed code. For more on this, see the MSDN Library article, "Runtime Callable Wrapper," at bit.ly/PTiAly. The RCW makes it easy and straightforward to interact with these components. The reverse is also true. Projection enables a Windows Runtime Component created with managed code to be referenced like a C++ type from native code and as a JavaScript object from within HTML/JavaScript projects.

Fundamental types appear automatically as C# types. The Windows Runtime has an ELEMENT_TYPE_STRING type that appears in .NET code as a String object. The ELEMENT_TYPE_I4 scalar appears as an Int32. The CLR will also take certain WinRT types and map them to appear in code as their .NET equivalents. For example, the WinRT type for a fixed-sized collection is IVector<T>, but this type will automatically appear as an IList<T> in .NET code. A WinRT HRESULT appears in the .NET Framework as an Exception type. The CLR will automatically marshal these types between the managed and native representations. Some types, such as streams, can be converted explicitly using a set of extension methods provided by the CLR. For a full list of types that are mapped in this fashion, refer to the MSDN Dev Center topic, ".NET Framework Mappings of WinRT Types," at bit.ly/PECJ1W.

These built-in features enable developers to create their own Windows Runtime Components using managed code with C# and Visual Basic. Visual Studio 2012 provides a template for creating Windows Runtime Components from Visual Basic, C# and C++. These components can be consumed and called from any other language that supports the Windows Runtime, including JavaScript. For this reason, you must follow some specific rules to create a Windows Runtime Component in C#.

Many languages, including C#, Visual Basic, C++ and JavaScript, have been updated with Windows 8 to support language projection.

Playing by the Rules

In general, the rules for creating WinRT types in C# relate to any publicly visible types and members your component provides. The restrictions exist because the Windows Runtime Component must be bound by the WinRT type system. The full set of rules is listed in the MSDN Dev Center topic, "Creating Windows Runtime Components in C# and Visual Basic," at bit.ly/OWDe2A. The fields, parameters and return values that you expose must all be WinRT types (it's fine to expose .NET types that are automatically mapped to WinRT types). You can create your own WinRT types to expose provided those types, in turn, follow the same set of rules.

Another restriction is that any public classes or interfaces you expose can't be generic or implement any non-WinRT interface. They must not derive from non-WinRT types. The root namespace for Windows Runtime Components *must* match the assembly name, which in turn can't start with "Windows." Public structures are also restricted to only have public fields that are value types. Polymorphism isn't available to WinRT types, and the closest you can come is implementing WinRT interfaces; you must declare as sealed any classes that are publicly exposed by your Windows Runtime Component.

These restrictions might be reason to consider an alternative approach to integrating components within your apps, especially if you're dealing with legacy code that would require significant refactoring. I'll discuss possible alternative approaches

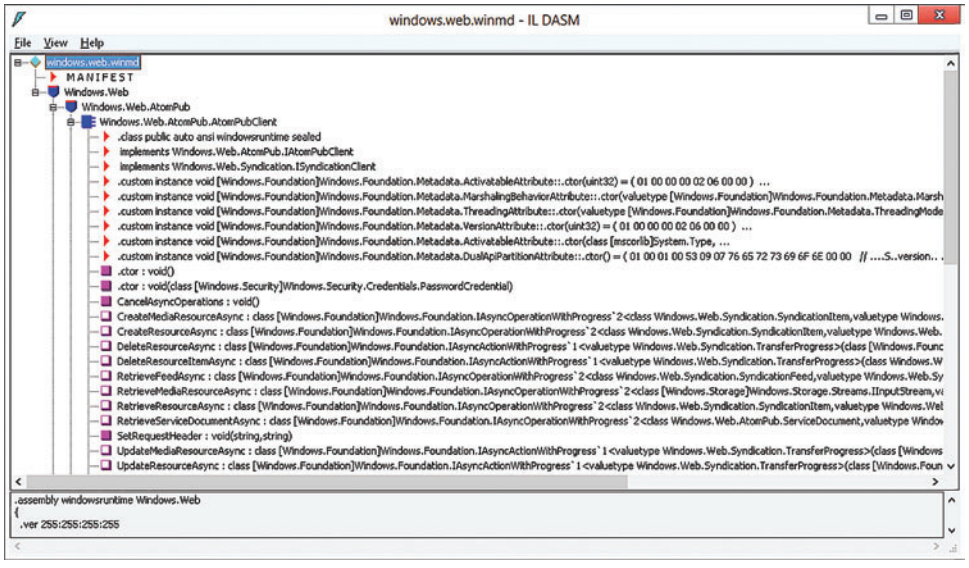


Figure 1 **Inspecting WinRT Types**

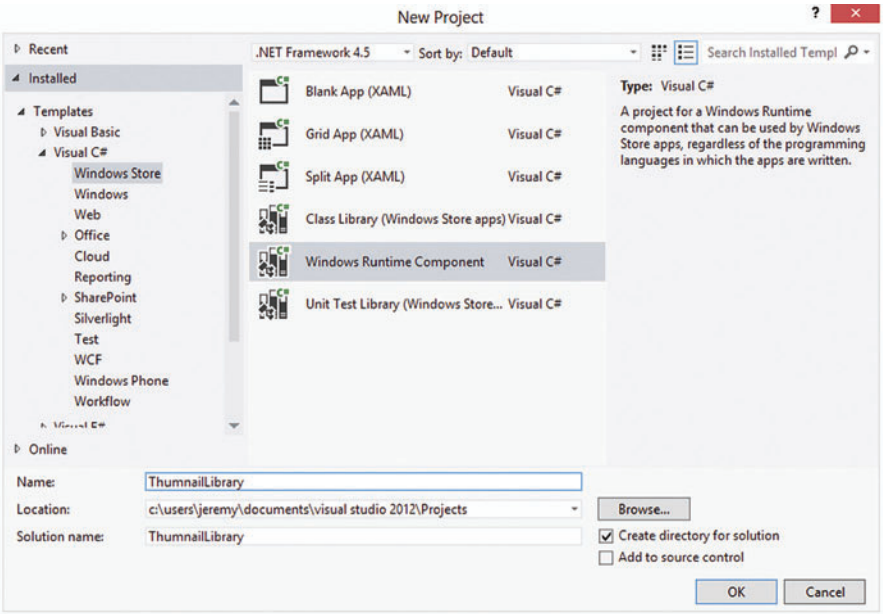


Figure 2 **Creating the Windows Runtime Component Project**

later. The restrictions are important to ensure the Windows Runtime Components can function appropriately within the WinRT environment and can be referenced and called from all language environments, including C++ and JavaScript.

The Thumbnail Generator

A simple app will demonstrate how to create a managed Windows Runtime Component with C# and consume it from a Windows Store app built with C#, JavaScript or C++. The component accepts a reference to an image file passed by the WinRT IStorageFile interface. It then creates a 100x100 pixel thumbnail of the image and saves it to the Windows Store app's local storage. It finally returns a URI that points to the thumbnail. The steps involved include:

1. Create the solution in Visual Studio 2012.
2. Build the Windows Runtime Component.

Figure 3 **XAML for Windows Store App Built with C#**

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <TextBlock
    Text="Tap the button below to choose an image to generate a thumbnail."
    Style="{StaticResource BodyTextStyle}"
    Margin="12"/>
  <Button Content="Pick Image" Grid.Row="1" Margin="12"
    Click="Button_Click_1"/>
  <TextBlock Text="Thumbnail:" Style="{StaticResource BodyTextStyle}"
    Grid.Row="2" Margin="12"/>
  <Image x:Name="ThumbnailImage" HorizontalAlignment="Left" Stretch="None"
    Grid.Row="3" Margin="12"/>
  <TextBlock Text="Source Image:" Style="{StaticResource BodyTextStyle}"
    Grid.Row="4" Margin="12"/>
  <Image x:Name="SourceImage" HorizontalAlignment="Left" Stretch="None"
    Grid.Row="5" Margin="12"/>
</Grid>
```

3. Create the language-specific project in C#, JavaScript or C++.
4. Reference the Windows Runtime Component.
5. Set up the UI for each project to enable the user to pick an image.
6. Display the image.
7. Call the Windows Runtime Component.
8. Display the thumbnail.

Create the Project and Solution

From within Visual Studio 2012, you begin by specifying your language of choice (in this case, C#) and choosing the Windows Store app templates. A template exists specifically for generating Windows Runtime Components. I selected this template and created a component called ThumbnailLibrary with a solution of the same name, as shown in **Figure 2**.

For this example, I created a single class called ThumbnailMaker. A private method returns a Task to asynchronously generate the thumbnail:

```
private static async Task<Uri> GenerateThumbnail(IStorageFile file)
{
}
```

The first step within the method is to open the file from storage and use the WinRT BitmapDecoder to decode the image stream:

```
using (var fileStream = await file.OpenReadAsync())
{
  var decoder = await BitmapDecoder.CreateAsync(fileStream);
}
```

Next, a file is created in the local storage for the app to hold the thumbnail. It will be named "thumbnail," with the same extension as the source file. The option to generate a unique name will ensure that multiple thumbnails can be generated without overwriting previous operations:

```
var thumbFile = await ApplicationData.Current.LocalFolder.CreateFileAsync(
  string.Format("thumbnail{0}", file.FileType),
  CreationCollisionOption.GenerateUniqueName);
```

An encoder is created from the decoded stream. It simply scales the bitmap to 100x100 pixels and then writes it to the file system:

```
using (var outputStream =
  await thumbFile.OpenAsync(FileAccessMode.ReadWrite))
{
  var encoder = await BitmapEncoder.CreateForTranscodingAsync(
    outputStream,
    decoder);
  encoder.BitmapTransform.ScaledHeight = 100;
  encoder.BitmapTransform.ScaledWidth = 100;
  await encoder.FlushAsync();
}
```

The last step is to build a URL that points to the file. The special ms-appdata prefix is used to reference the files in local storage. To learn more about how to reference content using URIs, read the MSDN Dev Center topic, "How to Reference Content," at bit.ly/SS711o. Although the topic is for HTML and JavaScript, the convention used to access resources is the same regardless of what language option you're using:

```
const string URI_LOCAL = "ms-appdata:///Local/{0}";
var storageUrl = string.Format(URI_LOCAL, thumbFile.Name);
return new Uri(storageUrl, UriKind.Absolute);
```

Windows Runtime Components written in C# can use any .NET functionality that's allowed for the Windows Store app profile. As mentioned earlier, however, public types and interfaces must only expose WinRT types. Because Task isn't a valid WinRT type, the public method for the component must expose the WinRT IAsyncOperation<T> type instead. Fortunately, an extension method exists to easily convert the .NET Task type to the WinRT IAsyncOperation type, as shown here:

```
public IAsyncOperation<Uri> GenerateThumbnailAsync(IStorageFile file)
{
  return GenerateThumbnail(file).AsAsyncOperation();
}
```

With the component now complete, you can compile it to make it ready for consumption from Windows Store apps.

Windows Runtime Components written in C# can use any .NET functionality that's allowed for the Windows Store app profile.

Under the Hood: Metadata

Build the Windows Runtime Component and then navigate to the output directory by right-clicking on the project name in the Solution Explorer and choosing "Open folder in Windows Explorer." When you navigate to the bin/Debug subdirectory, you'll find a metadata file has been generated for the component named ThumbnailLibary.winmd. If you open the file with ILDasm.exe, you'll see an interface has been generated for the component with a return type of:

```
Windows.Foundation.IAsyncOperation<Windows.Foundation.Uri>
```

Those are the WinRT types that were mapped for the component. It's possible to also inspect the metadata and see how the CLR projects WinRT types. Open the same metadata file with the special /project extension like this:

```
Ildasm.exe /project ThumbnailLibrary.winmd
```

The return type now appears as:

```
Windows.Foundation.IAsyncOperation<System.Uri>
```

Notice that the WinRT version of the URI is projected to the .NET equivalent. The method signature exposes all valid WinRT types for Windows Store apps to consume, but from managed code the types will appear as .NET classes. You can use the /project extension to inspect how projection will affect the signature of managed and unmanaged Windows Runtime Components.

Consume from C#

Consuming the component from C# should seem familiar because it's no different than referencing a class library. Note that there's no reason to build a Windows Runtime Component if your only target is other managed code. You simply reference the WinRT project and then consume the classes as you would from an ordinary C# class library. In the sample code, the CSharpThumbnails project has a reference to the ThumbnailLibrary. The XAML for the main page defines a button for the user to pick a photograph and contains two images to host the original image and the thumbnail version. **Figure 3** shows the basic XAML markup.

The codebehind creates an instance of the WinRT FileOpenPicker component and configures it to browse images:

```
var openPicker = new FileOpenPicker
{
  ViewMode = PickerViewMode.Thumbnail,
    SuggestedStartLocation = PickerLocationId.PicturesLibrary
};
openPicker.FileTypeFilter.Add(".jpg");
openPicker.FileTypeFilter.Add(".jpeg");
openPicker.FileTypeFilter.Add(".png");
```

The picker is called and a simple dialog is displayed if no valid files are found:

```
var file = await openPicker.PickSingleFileAsync();
if (file == null)
{
  var dialog = new MessageDialog("No image was selected.");
  await dialog.ShowAsync();
  return;
}
```

The source image is then wired for display. The file is passed to the Windows Runtime Component to generate a thumbnail, and the URI passed back is used to set the thumbnail image for display:

```
using (var fileStream = await file.OpenReadAsync())
{
  SourceImage.Source = LoadBitmap(fileStream);
  var maker = new ThumbnailMaker();
  var stream = RandomAccessStreamReference
    .CreateFromUri(await maker.GenerateThumbnailAsync(file));
  var bitmapImage = new BitmapImage();
  bitmapImage.SetSource(await stream.OpenReadAsync());
  ThumbnailImage.Source = bitmapImage;
}
```

Figure 4 shows the result of running this against a photo I took of a pumpkin I carved.

Consume from JavaScript

Unlike regular C# class libraries, Windows Runtime Components can be called from any language that's supported for creating Windows Store apps (core Windows Runtime Components that are part of the OS can be called from desktop apps as well). To see this in action, you can create the thumbnail app using HTML and JavaScript. This project is called JavaScriptThumbnails in the accompanying sample code download. The first step is to create an empty app using the blank Windows Store template for apps built using JavaScript. Once the template has been created, you can use simple HTML markup to define the page using the existing default.html file:

```
<p>Tap the button below to choose an image to generate a thumbnail.</p>
<button id="pick">Pick Image</button>
<br/><br/>
<p>Thumbnail:</p>
<img id="thumbnail" src="images/logo.png" alt="Thumbnail"/>
<p>Source Image:</p>
<img id="sourceImage" src="images/logo.png" alt="Image"/>
```

Next, add a reference to the WinRT project (ThumbnailLibrary) just as you would to a regular C# project. Build the project so that you can use IntelliSense for the newly referenced component. You can reference the source code for the project to see the JavaScript equivalent for opening the file picker and selecting the image. To create an

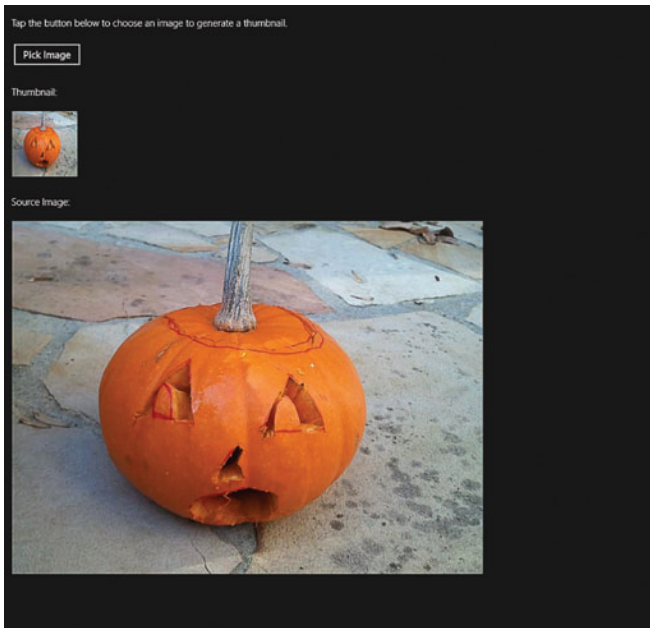


Figure 4 **Windows Store Thumbnail App Built Using C#**

instance of the managed Windows Runtime Component, generate the thumbnail and display it to the user, use the following JavaScript:

```
var thumb = new ThumbnailLibrary.ThumbnailMaker();
thumb.generateThumbnailAsync(file).then(function (fileUrl) {
  var thumbImage = document.getElementById("thumbnail");
  thumbImage.src = fileUrl.rawUri;
  thumbImage.alt = thumbImage.src;
});
```

As you can see, the API call is almost identical to the one used in the C# project. Projection automatically changed the method signature from Pascal case to camel case (the call to generate the thumbnail begins with a lowercase character, as is the common convention in JavaScript code), and a special library called "promises" is used to handle the asynchronous nature of the code using a then or done statement. You can learn more about promises by reading the MSDN Dev Center topic, "Quickstart: Using Promises," at bit.ly/OeWQCQ. The image tag supports URLs out of the box, so the URL passed back from the Windows Runtime Component is simply set directly to the src attribute on the image.

> One important caveat for using managed components in JavaScript code is that you can't debug JavaScript and managed code at the same time.

One important caveat for using managed components in JavaScript code is that you can't debug JavaScript and managed code at the same time. If you need to debug your component, you must right-click the project and choose the Debugging tab, and then select a debugging option that includes managed code. This is shown in **Figure 5**.

Consume from C++

You can also consume managed Windows Runtime Components from native projects. C++ shares the same rendering engine as C#, so the CPlusPlusThumbnails project has the same XAML as the CSharpThumbnails project. The codebehind is different because the project uses the native C++ language option. C++ uses a special concurrency library to handle asynchronous operations. You can learn more about this library by reading the MSDN Dev Center topic, "Asynchronous Programming in C++," at bit.ly/MUEqnR. The resulting code looks similar to the promises you saw in the JavaScript versions:

```
ThumbnailMaker^ maker = ref new ThumbnailMaker();
create_task(maker->GenerateThumbnailAsync(file)).then([this](Uri^ uri)
{
  RandomAccessStreamReference^ thumbnailStream =
    RandomAccessStreamReference::CreateFromUri(uri);
  create_task(thumbnailStream->OpenReadAsync()).then([this](
    IRandomAccessStream^ imageStream) {
    auto image = ref new BitmapImage();
    image->SetSource((IRandomAccessStream^)imageStream);
    ThumbnailImage->Source = image;
  });
});
```

When you run the app, you'll find it looks and behaves identically to the C# version.

Understand the Cost

Creating Windows Runtime Components using managed languages is a powerful feature. This feature does come at a cost, however, and it's important to understand the cost when you're using it in projects. Windows Store apps built using native code don't require the CLR to run. These apps may run directly in the Windows 8 environment. Similarly, apps developed using JavaScript also don't require a dependency on the CLR. They rely on the Trident rendering engine and Chakra JavaScript engine (the same engines that drive Internet Explorer 10) to render HTML and CSS and interpret JavaScript code. Windows Store apps built with JavaScript may call native Windows Runtime Components directly, but when they call managed Windows Runtime Components, they take on a dependency to the CLR.

The code written for the managed Windows Runtime Component will be compiled just-in-time (JIT) when it's first accessed by the CLR's JIT compiler. This might cause some delay the first time it's accessed. A precompilation service called NGen handles compiling modules installed on the device, but it can take up to a full day to eventually compile all of the modules in a package once it has been installed. The CLR also manages memory by performing garbage collection. The garbage collector (GC) divides the heap into three generations and collects only portions of the heap using an algorithm designed to optimize performance.

The GC might pause your app while it's performing work. This often only introduces a slight delay that isn't recognizable to the end user, and more intense garbage collection operations can often run in the background. If you have a large enough heap (when the managed portion of your code references hundreds of megabytes or more in memory objects), garbage collection might pause the app long enough for the user to perceive the lack of responsiveness.

Most of these considerations are already in place when you're building a managed Windows Store app. Managed code does add new concerns when you're adding it to a Windows Store app built

with C++ or JavaScript. It's important to recognize that your app will consume additional CPU and memory when you introduce managed components. It might also take a recognizable performance hit, depending on the component (although many apps take on managed references without any noticeable impact). The benefit is that you don't have to worry about managing memory yourself and, of course, you can leverage legacy code and skills.

When your component includes more than just code, you might consider creating an Extension SDK.

Alternatives for Managed Projects

If you're building Windows Store apps using managed code (C# or Visual Basic), you have several alternatives to create Windows Runtime Components that don't have the same restrictions. You can easily create reusable components using a simple C# class library. If the class library is built for Windows Store apps, you can reference the project from your own Windows Store app. The creation of a class library also removes the restrictions of having to expose only WinRT types and not being able to use features that aren't part of the WinRT type system, such as generics.

Another alternative to consider is the Portable Class Library (PCL). This is a special type of class library that can be referenced from a variety of platforms without recompiling. Use this option if you have code you wish to share between other platforms—such as Windows Presentation Foundation, Silverlight and Windows Phone—and your Windows Store app. You can learn more about the PCL by reading my three-part blog series, "Understanding the Portable Library by Chasing ICommand," at bit.ly/pclasslib.

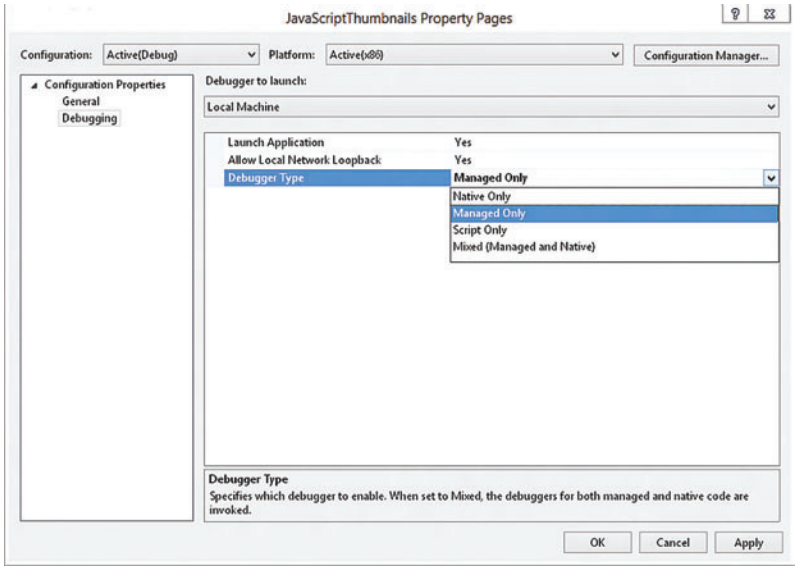


Figure 5 **Setting the Debug Options for a JavaScript Project**

When your component includes more than just code, you might consider creating an Extension SDK. This is a special form of SDK that Visual Studio 2012 can treat as a single item. The package might include source code, assets, files and even binaries, including Windows Runtime Components. You can also create design-time extensions to make it easier to consume and use your component from within Visual Studio 2012. Extension SDKs can't be posted to the Windows Store because they're not self-contained apps. You can learn more about Extension SDKs by reading the MSDN Library article, "How to: Create a Software Development Kit," at bit.ly/L90gnt.

When to Create Managed Windows Runtime Components

With so many possible alternatives, does it ever make sense to create Windows Runtime Components using managed code? Yes—but consider the following questions. The first question to ask is whether you need your components to be referenced from Windows Store apps written using JavaScript or native code using C++. If this isn't the case, there's no reason to use a Windows Runtime Component when class libraries and other options will work instead. If this *is* the case, you must create a Windows Runtime Component to be consumable from all of the available language options.

The next question is whether you should create your component in C++ or use managed code. There are a number of reasons to use managed code. One reason might be that your team is more experienced in C# or Visual Basic than in C++ and can leverage existing skills to build the components. Another reason might be that you have existing algorithms written in a managed language that will be easier to port if you keep the same language selection. There are some tasks that might be easier to write and maintain using managed languages and class libraries instead of using native code, and teams that are used to developing in managed languages will be far more productive.

Wrapping up, in this article you've learned you can create reusable Windows Runtime Components using managed C# and Visual Basic code. These components can be easily referenced and consumed from Windows Store apps written in any language, including JavaScript and C++. While it's important to understand the rules for creating Windows Runtime Components and the impact of choosing to use managed code, this option provides a unique opportunity to use the language of your choice and leverage existing code to create components that can be consumed by all Windows Store apps. ■

JEREMY LIKNESS *is a principal consultant for Wintellect LLC in Atlanta. He's a three-year Microsoft Silverlight MVP and the author of several books, including the upcoming "Building Windows 8 Applications with C# and XAML" (Addison-Wesley Professional, 2012). Learn more online at bit.ly/win8design and follow his blog at csharperimage.jeremylikness.com.*

THANKS to the following technical experts for reviewing this article: *Layla Driscoll, Shawn Farkas, John Garland, Jeff Prosise and Jeffrey Richter*

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

YOUR MAP TO THE .NET DEVELOPMENT PLATFORM

BIRDS OF A FEATHER CODE TOGETHER

Orlando, FL | **December 10-14**

Royal Pacific Resort at Universal Orlando | **vslive.com/orlando**

Don't miss your chance for great .NET education in 2012 at Visual Studio Live! Orlando.

- **Mobile**
- **Visual Studio / .NET**
- **Web / HTML5**
- **Windows 8 / WinRT**
- **WPF / Silverlight**

Save Up To $400! Register Before November 7

Use Promo Code VSL2OCT

vslive.com/orlando

ROYAL PACIFIC RESORT LIVE! DECEMBER 10 - 14 LIVE! ORLANDO

Visit **vslive.com/orlando** or scan the QR code to register and for more event details.

GOLD SPONSOR: axosoft

SUPPORTED BY: Microsoft, Microsoft Visual Studio, msdn, Visual Studio MAGAZINE

PRODUCED BY: 1105 MEDIA

MEDIA SPONSOR: THE CODE PROJECT WWW.CODEPROJECT.COM

VISUAL STUDIO LIVE! ORLANDO AGENDA AT-A-GLANCE

Web / HTML 5	Mobile	WPF / Silverlight	Windows 8 / WinRT	Visual Studio / .NET

Start Time	End Time	Visual Studio Live! Pre-Conference Workshops: Monday, December 10, 2012 *(Separate entry fee required)*			
8:00 AM	12:00 PM	VSM1 - Workshop: XAML User Experience Design *Billy Hollis*	VSM2 - Workshop: Services - Using WCF and ASP.NET Web API *Miguel Castro*		VSM3 - Workshop: Build a Windows 8 Application in a Day *Rockford Lhotka*
1:00 PM	5:00 PM	VSM1 - Workshop: XAML User Experience Design *Billy Hollis*	VSM2 - Workshop: Services - Using WCF and ASP.NET Web API *Miguel Castro*		VSM3 - Workshop: Build a Windows 8 Application in a Day *Rockford Lhotka*
5:00 PM	7:00 PM	EXPO Preview			
7:00 PM	8:00 PM	Live! 360 Keynote			

Start Time	End Time	Visual Studio Live! Day 1: Tuesday, December 11, 2012			
8:00 AM	9:00 AM	Visual Studio Live! Keynote			
9:15 AM	10:30 AM	Live! 360 Keynote			
11:00 AM	12:15 PM	VST1 **Controlling ASP.NET MVC 4** *Philip Japikse*	VST2 **Leveraging XAML for a Great User Experience** *Billy Hollis*	VST3 **Intro to Metro** *Miguel Castro*	VST4 **What's New in Visual Studio 2012** *Rob Daigneau*
2:00 PM	3:15 PM	VST5 **Azure Web Hosting (Antares)** *Vishwas Lele*	VST6 **Implementing the MVVM Pattern in WPF** *Miguel Castro*	VST7 **Smackdown: Metro Style Apps vs. Websites** *Ben Dewey*	VST8 **What's New in Windows Phone 8 for Developers** *Nick Landry*
4:15 PM	5:30 PM	VST9 **I'm Not Dead Yet! AKA the Resurgence of WebForms** *Philip Japikse*	VST10 **Building High Performance, Human-Centered Interactive Data Visualizations Dashboards** *Nick Landry*	VST11 **Going from Silverlight or WPF to Metro** *Greg Levenhagen*	VST12 **What's New in the .NET 4.5 BCL** *Jason Bock*
5:30 PM	7:30 PM	Exhibitor Reception			

Start Time	End Time	Visual Studio Live! Day 2: Wednesday, December 12, 2012			
8:00 AM	9:00 AM	Visual Studio Live! Keynote			
9:15 AM	10:30 AM	VSW1 **JavaScript and jQuery for .NET Developers** *John Papa*	VSW2 **Navigation and UI Shells for XAML Applications** *Billy Hollis*	VSW3 **Filling Up Your Charm Bracelet** *Ben Dewey*	VSW4 **Coming Soon!**
11:00 AM	12:15 PM	VSW5 **Applying JavaScript Patterns** *John Papa*	VSW6 **Static Analysis in .NET** *Jason Bock*	VSW7 **WinRT Services** *Rockford Lhotka*	VSW8 **Reach the Mobile Masses with ASP.NET MVC 4 and jQuery Mobile** *Keith Burnell*
1:45 PM	3:00 PM	VSW9 **HTML5 for Business: Forms and Input Validation** *Todd Anglin*	VSW10 **What's New in Azure for Devs** *Vishwas Lele*	VSW11 **Using Azure with Windows Phone and Windows 8!** *Greg Levenhagen*	VSW12 **iOS Development Survival Guide for the .NET Guy** *Nick Landry*
4:00 PM	5:15 PM	VSW13 **Making HTML5 Work Everywhere** *Todd Anglin*	VSW14 **In Depth Azure PaaS** *Vishwas Lele*	VSW15 **From a New Win8 Project to the Store** *Elad Shaham*	VSW16 **Creating Restful Web Services with the Web API** *Rob Daigneau*

Start Time	End Time	Visual Studio Live! Day 3: Thursday, December 13, 2012			
8:00 AM	9:15 AM	VSTH1 **Identify & Fix Performance Problems with Visual Studio Ultimate** *Benjamin Day*	VSTH2 **Patterns for Parallel Programming** *Tiberiu Covaci*	VSTH3 **Windows 8 Metro Style Apps for the Enterprise** *Ben Hoelting*	VSTH4 **How to Take WCF Data Services to the Next Level** *Rob Daigneau*
9:30 AM	10:45 AM	VSTH5 **Building Single Page Web Applications with HTML5, ASP.NET MVC4, Upshot.js and Web API** *Marcel de Vries*	VSTH6 **ALM Features in VS12** *Brian Randell*	VSTH7 **Build Windows 8 Apps using Windows Online Services** *Rockford Lhotka*	VSTH8 **Coming Soon!**
11:00 AM	12:15 PM	VSTH9 **Busy .NET Developer's Guide to Node.js** *Ted Neward*	VSTH10 **Intellitrace, What is it and How Can I Use it to My Benefit?** *Marcel de Vries*	VSTH11 **Expression Blend 5 For Developers : Design Your XAML or HTML5\CSS3 UI Faster** *Ben Hoelting*	VSTH12 **Coming Soon!**
1:30 PM	2:45 PM	VSTH13 **Cloudant NoSQL on Azure** *Sam Bisbee*	VSTH14 **Team Foundation Server 2012 Builds: Understand, Configure, and Customize** *Benjamin Day*	VSTH15 **Making your Windows 8 App Come to Life, Even When It's Not Running** *Elad Shaham*	VSTH16 **EF Code First Magic Unicorn Edition and Beyond** *Keith Burnell*
3:00 PM	4:15 PM	VSTH17 **Introduction to OAuth** *Ted Neward*	VSTH18 **Azure Hosted TFS** *Brian Randell*	VSTH19 **No User Controls Please: Customizing Metro Style Controls using XAML** *Elad Shaham*	VSTH20 **The LINQ Programming Model** *Marcel de Vries*
4:30 PM	5:45 PM	Live! 360 Conference Wrap-up			

Start Time	End Time	Visual Studio Live! Post-Conference Workshops: Friday, December 14, 2012 *(Separate entry fee required)*	
8:00 AM	12:00 PM	VSF1 **Workshop: Mastering ASP.NET MVC4 in Just One Day** *Tiberiu Covaci*	VSF2 **Workshop: TFS/ALM** *Brian Randell*
1:00 PM	5:00 PM	VSF1 **Workshop: Mastering ASP.NET MVC4 in Just One Day** *Tiberiu Covaci*	VSF2 **Workshop: TFS/ALM** *Brian Randell*

For the complete session schedule and full session descriptions, please check the Visual Studio Live! Orlando web site at ***vslive.com/orlando***
**Speakers and Sessions Subject to Change.*

Writing Silverlight and WPF Apps with Windows Runtime XAML in Mind

Pete Brown

Windows Runtime (WinRT) XAML for new Windows Store apps is the latest member of the XAML and C#/Visual Basic family many of us have come to love. It all officially started in 2006 with the Microsoft .NET Frameowork 3.0 and "Avalon" (later named Windows Presentation Foundation, or WPF). After that came several more revisions of WPF, including the latest, WPF 4.5, and alongside we've had seven named versions of Silverlight (including 1.1 and 5.1), several versions of Windows Phone and more. You'll even find part of the XAML stack available on .NET Micro Framework devices.

You might wonder why there are so many variations on XAML and the .NET Framework. Although many of the implementations have converged on similar uses (Silverlight to write desktop apps, for example), each platform was developed and optimized for different scenarios and target platforms. For example, Silverlight was designed to be cross-platform and Web-hosted. XAML on Windows Phone was designed for phone-specific scenarios and hardware, and WinRT XAML on Windows 8 was designed for high-performance, on the metal (x86/x64 and ARM), touch-first (but not touch-only) Windows Store apps.

Nevertheless, these implementations of XAML have far more in common than not. It's because of these similarities that the differences seem so pronounced. Of course, tiny differences can cause a lot of development challenges, something I know from personal experience and from talking with other developers. However, the fact that we can even talk about compatibility at a detail level illustrates the similarity between the languages, libraries and markup.

In this article, I'm targeting two important scenarios: sharing code with a companion app and future-proofing your current development.

Companion App This is a simultaneous code-sharing, or cross-compilation, scenario for WPF and Silverlight application developers who want to develop companion Windows Store apps for Windows 8 at the same time.

Future Proofing In this scenario, developers are creating new WPF and Silverlight applications today but are not currently targeting Windows 8. When the organization adopts Windows 8, the developers want to be ready; they want to help ensure that appropriate portions of their apps will be more easily ported to the new Windows UI.

Decades of programming experience have taught us that reuse and portability are never free. However, with the techniques covered here, you'll find much of the effort a minimal increment over what you would normally do to create well-architected apps.

This article discusses:

- Code sharing for Windows Store companion apps
- Portable Class Library projects
- Design principles for Windows Store apps

Technologies discussed:

Microsoft .NET Framework, Silverlight, Windows Presentation Foundation, Windows Runtime, XAML

Thoughtful Architecture Is Essential

Breaking large applications into smaller apps is possible only if you have good architecture to begin with. In fact, if your application has a lot of interdependencies between code modules, a lot of heavy class hierarchies, or otherwise feels like a ball of mud or throwaway code, reusing or porting anything will be extremely difficult. But don't despair! Code can be refactored, and new code can be written with the new architecture in mind.

When designing new apps, I encourage XAML developers to follow a few key approaches: binding, the Model-View-ViewModel (MVVM) pattern and service classes.

Binding The more you embrace data binding when developing in XAML, the easier it is to keep your logic separated from the UI. Ideally, you set the DataContext for the UI, and everything else is handled by binding with data or commands. In practice, few apps are able to attain this level of separation, but the closer you get, the easier your life will be.

The MVVM Pattern The MVVM pattern goes hand-in-hand with data binding. The ViewModel is what the UI will bind to. There's a ton of great information (and toolkits, which I'll cover later) available for free on the Internet and in books, so I won't rehash that here.

Service Classes This approach is not to be confused with Web services. Instead, these are classes that provide reusable functionality on the client. In some cases, they might call out to RESTful or other services. In other cases, they might interface with your business logic. In all cases, they encapsulate potentially volatile code and make swapping out implementations easier. For example, in **Figure 1**, the ViewModel talks to service classes in order to use both platform services and to resolve external dependencies.

I know. You're thinking, "Ugh! Another layer diagram." But you know how important these concepts are. The intent is to decouple yourself from the platform you're on as much as is reasonable within your budget and time constraints. By factoring out code that, for example, makes COM or p-invoke calls into desktop elements such as Windows Imaging or DirectShow, you can more easily replace that implementation with the WinRT camera API in your Windows Store app. Service classes are also a great place to encapsulate other platform differences, such as contract implementations: sending an e-mail from your Windows Store app would use a contract, but on the desktop it would likely mean automating Outlook or hooking into an SMTP server.

Of course, it's easy to go overboard with architecture and never actually deliver. Good architecture should make development easier,

Embrace the New Windows UI Design Aesthetic

When developers first consider building Windows Store versions of existing Windows Presentation Foundation (WPF) and Silverlight apps, they immediately run into the roadblock of having to rethink the UX and visual design. To most developers, the prospect of restyling an entire application isn't appealing. If you think Windows Store apps are in your future, a little work now to embrace the new Windows UI design aesthetic and guidelines will really pay off later.

The new Windows UI design provides a framework you can use to help guide your UI design choices. Several BUILD 2011 videos cover Windows Store app design. You can find them on MSDN Channel 9 at bit.ly/oB56Vf. In addition, here are a few things you'll want to do to embrace the new Windows UI design aesthetic in your desktop apps:

- **Be authentically digital.** Generally, shy away from the practice of designing the UI to be a fake analog of physical objects (skeuomorphism). You should avoid not only recreating physical objects, but also derived techniques such as glossy buttons, 3D shading, realistic shadows and glass backgrounds.
- **Have a clear type hierarchy.** Don't use a ton of different fonts, and for the fonts you do use, stick to a few key and easily distinguished sizes. Headings should be easily distinguished from field labels and help text.
- **Brand the application.** Use your colors, logos, type and more as appropriate. It helps to have your Web developers and designers involved here, as they often have more hands-on experience with branding.
- **Use a page-based navigation metaphor.** Silverlight developers will find this natural (the navigation framework is almost identical), but WPF developers might have a little more work to do to move from the traditional multi-window and dialog desktop approach.
- **Be task-focused.** Keep the UI task-focused, and try not to cram every last thing on a single form. Unfortunately, users often think they want every function on a single page, but over time that becomes difficult to use, difficult to maintain and difficult to learn. In some cases, consider breaking large applications into smaller, task-focused apps.
- **Avoid unnecessary decoration.** Keep the UI simple. You want to draw the user's eyes to the work he needs to focus on, not to menus, navigation elements, window borders and other chrome.

Think of these concepts much like you would the older Windows UI design guidelines. They are guidelines you should follow if you want your app to feel familiar to the user and fit in with the new Windows UI. There's more to it, of course, so I encourage you to look at the BUILD videos and to study examples of apps that other developers and designers have written.

One way to get a head start on using the UI styles is to copy the Windows Store app style resources into your own desktop apps. You can find many of the style resources in the Windows Kits folder. On my machine, that folder is located at C:\Program Files (x86)\Windows Kits\8.0\Include\winrt\xaml\design.

Some of the styles and resources are usable as is. Some will be unusable because of newer controls (ListView and GridView, for example), and some may require a fair bit of tweaking. However, this is a great way to get a modern look and feel for your app and to promote an easy transition between desktop and Windows Store styles. Even when you can't use a style or template directly, you can learn from it to get a head start on your own styling work.

Your UI will still need some work when you move it to the Windows Runtime, but if you embrace these concepts in your apps now, you'll go a long way toward ensuring that your transitions between the new Windows Store UI and desktop apps are not jarring to your users or your developers.

I'm all for less work. Style the app once, make the transition easier and get a modern look as a bonus.

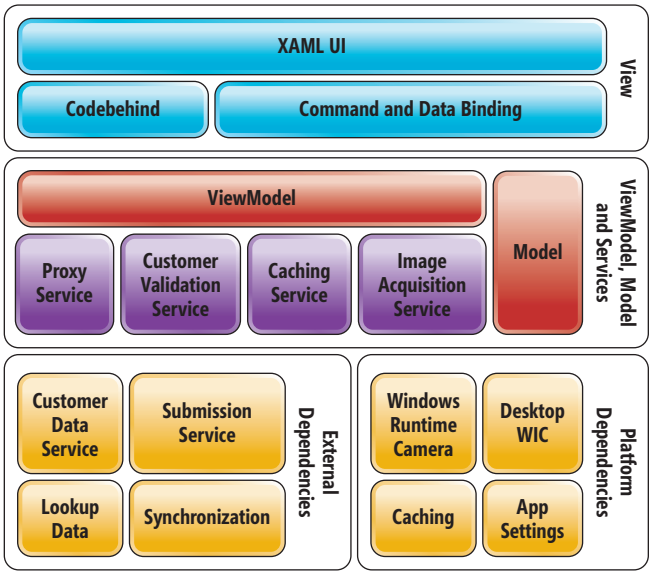


Figure 1 **Relationship Between the ViewModel and Service Classes**

not harder. If you find your team struggling with the minutiae of a particular architectural pattern, you're probably wasting time. Instead, understand the patterns and what they bring, and then make intelligent and informed decisions about the trade-offs. In most cases, implementing 85 percent of a great architecture is better than 0 percent of the same. Similarly, the cost involved in getting that last 15 percent implemented is often not worth it.

Once you separate out platform-dependent code, quite a bit of other nontrivial code can be reused.

Reusing Code and Components

The Windows Runtime gets a lot of attention in new Windows Store apps. It's new, it's fast and it has a lot of cool features. However, sometimes we forget that the Windows Runtime isn't the sum total of what you have to work with; Windows Store apps built with XAML use a lot of the .NET Framework 4.5 along with the Windows Runtime. In fact, the .NET Framework 4.5 has significant updates, including many that enable it to be used side-by-side with the Windows Runtime. It's still .NET, though—sharing source code, and even compiled binaries, with other .NET implementations is quite reasonable to do.

When sharing code with WPF apps, the best thing you can do right now is write your WPF code to work with the .NET Framework 4.5. Windows Store apps use a safe subset of .NET 4.5 alongside the Windows Runtime. When the .NET Framework provides the same functionality as the Windows Runtime (XAML is a great example), the .NET profile for Windows Store apps uses the WinRT version. If you stick to this subset as much as possible, your code will be extremely portable. You'll find this especially helpful when you're working with services and asynchronous code.

When sharing with Silverlight, you're a little more restricted in that Silverlight doesn't use the .NET Framework 4.5 or, for the most part, the Task Parallel Library. However, with the async targeting pack for Silverlight 5, Visual Studio 2012 users writing Silverlight applications gain access to some of the Task<T> and async/await functionality. The Add Service Reference code won't generate asynchronous client-side code, however. If that's important to you, you can port the generated proxy code to Silverlight. A better approach, using the architecture I proposed here, is to encapsulate the Web service interactions inside local service classes.

What about code that's identical? Model classes—and, in many cases, ViewModel classes—fall into this category. Here, you have two choices: Portable Class Library (PCL) projects and linked projects.

PCL Projects PCL is a great way to share compiled DLLs between different .NET targets. PCL enables you to share assemblies between .NET, .NET for Windows Store apps, Silverlight, Windows Phone and more. To use PCL, you need to restrict the assembly's references to those on the compatibility list. Find out more about PCL at bit.ly/z2r3eM.

Linked Projects with Shared Source Code and Conditional Compilation This is a Visual Studio feature that enables you to have multiple projects, each targeting different platforms but with a single shared copy of the source code. In the past, I used this approach primarily when sharing code between ASP.NET on the server and Silverlight on the client. You can find an example on my blog at bit.ly/RtLhe7.

Although I've traditionally used linked projects for most of my reuse scenarios, PCL has really grown up in recent versions. If you're targeting WPF and WinRT XAML, PCL will let you use almost everything in the .NET 4.5 profile for Windows Store apps. When in doubt, start with PCL.

One thing I do like about the linked project approach is that I can provide platform-specific additional functionality through partial classes and conditional compilation. Conditional compilation symbols are set on the Build tab of the project properties page, as shown in **Figure 2**.

By default, a number of conditional compilation symbols are defined for you, as shown in **Figure 3**.

Note that the full .NET Framework doesn't define any compilation symbols—it's considered the default platform. If that makes you uneasy, you can add your own compilation symbols; just make sure you do that for all .NET Framework projects in your solution.

Conditional compilation is also how, in code, you can pull in different namespaces, depending on the target platform:

```
#if NETFX_CORE
using Windows.UI.Xaml;
#else
using System.Windows.Xaml;
#endif
```

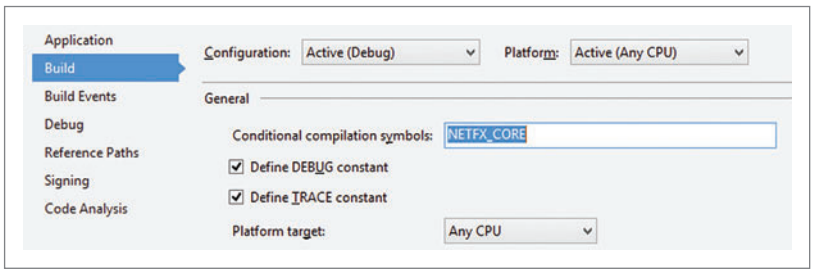


Figure 2 **Project Properties Page Showing Compilation Symbols**

Figure 3 **Conditional Compilation Symbols**

Platform	Compilation Symbol
Windows Runtime + .NET Framework	NETFX_CORE
WPF/.NET Framework 4.5 Desktop	(none)
Silverlight	SILVERLIGHT

In any nontrivial app, you'll likely find yourself using both PCL and linked projects for sharing code. Sharing XAML, however, is more complex.

Sharing XAML UIs and Resources

For several reasons, sharing XAML is more difficult than sharing code. The most important step for doing so is to tailor the UI for the form factor and platform you're targeting. (See "Embrace the New Windows UI Design Aesthetic" for some pointers here.) In most cases, your development time is better spent making sure the code is reusable, because you'll see more payoff for less effort. Nevertheless, there will be both a desire and a tendency to share XAML across projects.

XAML doesn't have the concept of conditional compilation. As a result, namespace differences are more difficult to manage than in code. In addition to the namespaces themselves, the way you import them into XAML has changed between WinRT XAML and the other versions.

Consider this .NET Framework XAML:

```
xmlns:localControls="clr-namespace:WpfApp.Controls"
```

And this WinRT XAML:

```
xmlns:localControls="using:WindowsApp.Controls"
```

Instead of the "clr-namespace" statement, WinRT XAML uses the "using" statement in XAML when importing namespaces. Why did the product teams do this? Imported namespaces in XAML might come from non-CLR code—C++, for example. Not only do you now have XAML support in C++, but you can write extension assemblies in C++ and use them in your .NET code. That means that the term "clr-namespace" is no longer accurate.

> When you dynamically load XAML you're working with strings.

One way to deal with the differences is to dynamically load the XAML. Back in the Silverlight 1.1 alpha days, this was the way user-controls were generated: the XAML was dynamically loaded and processed at run time. Since then, this approach has been used by app developers for flexibility across all platforms.

When you dynamically load XAML you're working with strings. That means you can replace or subset text at will before loading it into the visual tree. For example, let's say you have an empty usercontrol definition in your WPF or Silverlight project. The usercontrol is just a shell that defines namespaces and the class. Then you have an equivalent shell in the WinRT XAML project.

Here's the .NET XAML:

```
<UserControl x:Class="WpfApp.Controls.AddressControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Grid x:Name="LayoutRoot">
  </Grid>
</UserControl>
```

And here's the WinRT XAML:

```
<UserControl x:Class="WindowsApp.AddressControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Grid x:Name="LayoutRoot">
  </Grid>
</UserControl>
```

The two XAML user controls are plain-old-vanilla template output. The only change I made was to give the root Grid element a name and remove some unused namespaces to keep the code short. The codebehind for the controls can be shared using the code-sharing techniques, although I recommend partitioning the code into a separate shared class whenever possible.

Then, to load the XAML at run time, you can use code similar to what you see in **Figure 4**.

The code to load in WinRT XAML is even easier because of direct string loading support:

```
string xaml = "..."
LayoutRoot.Children.Add((UIElement)XamlReader.Load(xaml));
```

For simplicity, I loaded XAML from a hardcoded string here and used XAML that's already portable and didn't require any string manipulation. On the former, if you want designer support, make the code load text from a XAML resource file. Then, include all the usual class and xlmns definitions to support the designer, but throw them away during the XAML loading step.

Regardless of how the XAML gets into the visual tree, if you want to wire up events in the dynamically loaded XAML, you can do that from code as well:

```
LayoutRoot.Children.Add((UIElement)XamlReader.Load(xaml));

var t1 = FindName("t1") as TextBox;
if (t1 != null)
{
  t1.TextChanged += t1_TextChanged;
}
```

Just as you can load the XAML UI, you can also load resources at run time. You can create them from scratch or, using the same approaches shown here, load them from the XAML source:

```
private void LoadResources()
{
  string xaml =
    "<Style xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\" " +
      "TargetType=\"TextBox\">" +
      "<Setter Property=\"Margin\" Value=\"5\" />" +
    "</Style>";

  // Implicit styles use type for key
  var key = typeof(TextBox);

  App.Current.Resources.Add(key, XamlReader.Load(xaml));
}
```

When loading resources dynamically, be sure you have them loaded before they're referenced. In the case of implicit styles, they simply won't apply to anything already in the visual tree when the resource is created.

To be clear, this whole approach is a bit hacky, and it won't work smoothly in every scenario. (At some point, the string manipulation

Figure 4 **Code for Dynamically Loading XAML**

```
public partial class AddressControl : UserControl
{
  public AddressControl()
  {
    InitializeComponent();
    LoadXaml();
  }

  private void LoadXaml()
  {
    string xaml =
    "<Grid xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\" " +
      "xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\">" +
      "<Grid.RowDefinitions>" +
        "<RowDefinition Height=\"Auto\" />" +
        "<RowDefinition Height=\"30\" />" +
      "</Grid.RowDefinitions>" +
      "<TextBlock Text=\"Address\" Grid.Row=\"0\" FontSize=\"10\" />" +
      "<TextBox x:Name=\"t1\" Grid.Row=\"1\" FontSize=\"15\" />" +
    "</Grid>";

    var reader = new StringReader(xaml);
    var xmlReader = XmlReader.Create(reader);

    LayoutRoot.Children.Add((UIElement)XamlReader.Load(xmlReader));
  }
}
```

can be more work than it's worth.) But if you want maximum cross-platform reuse, this is one possible way to accomplish it. Personally, I'd use this only for critical UI components that are going to change a lot or are too complex or too numerous to simply copy and paste on a regular basis.

Of course, you could also write a custom build action or other Visual Studio extension to automatically handle the processing when files are saved, checked-out, built or for some other step.

The final way is to eschew XAML altogether and instead create your entire UI from code. Generally, I caution against this approach for most apps—it's a lot of extra work, and you get zero designer support. However, data-driven UIs do extremely well with this approach. If you can create it in XAML, you can create it in code, as in the WinRT XAML shown in **Figure 5**.

This code, with the exception of font settings (to keep the listing short), is equivalent to the dynamically loaded XAML that adds a TextBlock and TextBox to the LayoutRoot Grid, contained in a grid.

Figure 5 **Creating a UI with WinRT XAML**

```
private void CreateControls()
{
  Grid g = new Grid();
  RowDefinition r1 = new RowDefinition();
  RowDefinition r2 = new RowDefinition();
  r1.Height = new GridLength(1, GridUnitType.Auto);
  r2.Height = new GridLength(30.0);

  g.RowDefinitions.Add(r1);
  g.RowDefinitions.Add(r2);

  TextBlock t = new TextBlock();
  t.Text = "Address";
  Grid.SetRow(t, 0);
  TextBox tb1 = new TextBox();
  Grid.SetRow(tb1, 1);

  g.Children.Add(t);
  g.Children.Add(tb1);

  LayoutRoot.Children.Add(g);
}
```

Consider Open Source Toolkits

Many of the best XAML/C# toolkits are open source. In an active open source project you're likely to see interested people pick up and add the types of features that make moving between Silverlight, WPF and the Windows Runtime possible.

There are a number of MVVM toolkits, for example, which are mostly portable across the different flavors of XAML. By using one of these, you can help reduce the number of changes your source code needs when you're sharing or porting it. Similarly, you'll find controls, local databases and other toolkits available for use across platforms.

There's always a risk that open source developers won't work to target new versions of the platforms, but with access to the source code, you could fork the code and take that on should you want to. Just contribute the code back when you're done.

Deciding What Should Be a Windows Store UI App

When it comes time to create your new Windows Store app, keep in mind that not all aspects of all desktop applications translate directly into Windows Store apps. For example, a large application with 300 forms, performing an entire suite of different functions for different users, is not necessarily a good candidate for the new Windows UI. Instead, Windows Store apps should be task-focused and tailored to a specific use.

Consider an insurance application. You might have one app that's shared by a large number of users in the company. It handles supervisory functions for managing the security of users. It allows taking glass-damage claims over the phone, as well as on a laptop in the field. It has built-in functionality for creating new policies, and so on.

If you break up the application into a number of task-focused (or user-focused) smaller apps, it will be more appropriate for the new Windows UI. For example, you might want a separate field-adjuster app that's focused specifically on taking down accident information at the scene. Because it's focused, it includes support for taking photographs and recording video, as well as local caching of data for disconnected or poor-signal areas. This app probably shares a lot of code with the other apps in the enterprise, but its focus makes it easier to tailor to the scenarios for which it was built.

Wrapping Up

WinRT XAML, Silverlight and WPF have been created with different purposes in mind, but they're more similar than different. Sharing code between them is easy, and sharing XAML is possible. There are lots of additional techniques you can follow to target all three platforms and move from the desktop to the new Windows Store UI. I'd love to continue this conversation on Twitter and on my blog at 10rem.net. If you've used other techniques for multi-targeting or porting code to the Windows Runtime, I'd love to hear from you. ■

PETE BROWN *is the Windows 8 XAML and gadget guy at Microsoft. He's also the author of "Silverlight 5 in Action" (Manning Publications, 2012) and "Windows 8 XAML in Action" (Manning Publications, 2012). His blog and Web site are 10rem.net, and you can follow him on Twitter at twitter.com/pete_brown.*

THANKS to the following technical expert for reviewing this article:
Tim Heuer

NOTICE

AVOID CONTAMINATION WASH YOUR DATA

The simple act of washing your hands is the most effective way of preventing the spread of infection – and the same goes for your data. Prevent the spread of bad decisions, waste, and lost marketing and sales opportunities with the Data Quality Suite from Melissa Data.

Our Data Quality Suite enables you to:

- **Verify, correct, and standardize U.S. and global addresses**
- **Validate, parse, and standardize email addresses**
- **Verify phone numbers and identify as cell, landline, or VOIP**
- **Verify and genderize names, and detect company names and suspicious words**

The Data Quality Suite is available as multiplatform programming APIs or in the cloud for real-time and batch processing.

STOP the Spread of Dirty Data – START Your Free Trial Today!

Visit www.melissadata.com/dqs
or call 1-800-MELISSA (635-4772).

MELISSA DATA®
Your Partner in Data Quality

Using the MVVM Pattern in Windows 8

Laurent Bugnion

Any programmer with previous experience in any of the XAML-based frameworks has probably at least heard of the Model-View-ViewModel (MVVM) pattern. Some have been using it extensively in all of their Windows Presentation Foundation (WPF), Silverlight or Windows Phone applications. Others have avoided it, either because they misunderstand what the pattern does, exactly, or because they don't want to add what they see as a new level of complexity to their application.

This article discusses:

- The MVVM pattern in Windows 8
- C++ and XAML in the sample project Hilo
- MVC vs. MVVM
- The MVVM Light Toolkit
- Creating a new application in Windows 8
- Implementing and mocking the data service
- Registering and calling the services
- Setting up navigation

Technologies discussed:

Windows 8, Visual Studio 2012, MVVM Light Toolkit

Code download available at:

archive.msdn.microsoft.com/mag201210MVVM

You don't have to use the MVVM pattern to build XAML-based applications. It's absolutely possible to use traditional patterns such as event-based interactions to create compelling apps. However, decoupled patterns such as MVVM bring quite a few advantages. Notably, the MVVM pattern can tremendously enhance the experience in Expression Blend and facilitate the designer-developer workflow.

Recently—and especially with the extension of XAML to new platforms such as Windows Phone and, of course, Windows 8—the usage of MVVM has grown to a whole new level, from a niche pattern that only a few enthusiast coders were using, to a mainstream practice encouraged by Microsoft.

Preconditions

This article demonstrates how to use the MVVM pattern together with the MVVM Light Toolkit, an open source toolkit that facilitates common operations in XAML-based applications. To follow through the examples, Visual Studio 2012 is needed, as well as the latest version of the MVVM Light Toolkit. Note that you can use the Express edition of Visual Studio 2012, which is available for free. Of course, any other edition can also be used.

Visual Studio 2012 can be downloaded from bit.ly/vs12rc (Ultimate, Premium or Professional editions). The free Express edition can be downloaded from bit.ly/vs12express.

The MVVM Light Toolkit installer can be downloaded from the Download section at mvvmlight.codeplex.com. After installing the

Figure 1 **Expression Blend for Windows Store Apps with Design-Time Data**

MSI, a Readme page opens and you can choose to install project and item templates for the version of Visual Studio you're using.

The MVVM Pattern in Windows 8

Windows 8 relies on a new set of APIs named the Windows Runtime (WinRT). There are three possible ways (sometimes called "projections") to program applications running on the Windows Runtime.

The first way is probably the most intuitive for people with skills in other XAML-based frameworks such as WPF, Silverlight or Windows Phone. It uses XAML as a front end, and C# or Visual Basic for its logic. While there are a few differences between Silverlight and the WinRT XAML/C#/Visual Basic projection, the skills and most concepts are the same. In fact, MVVM is the pattern of choice for this projection, for the same reasons as in those older frameworks.

Figure 2 **Observable Property**

```
private string _firstName;

public string FirstName
{
  get
  {
    return _firstName;
  }

  set
  {
    if (_firstName == value)
    {
      return;
    }

    _firstName = value;

    var handler = PropertyChanged;
    if (handler != null)
    {
      handler(this, new PropertyChangedEventArgs("FirstName"));
    }
  }
}
```

Second, developers familiar with HTML and JavaScript can also code Windows Store apps with an HTML-based projection. With the growing popularity of MVVM in XAML, HTML developers have wanted to use data binding in HTML/JavaScript too. Because of the lack of data-binding mechanisms, however, it was necessary to start from scratch and to recreate this functionality. This is what frameworks such as knockout.js (knockoutjs.com) are doing. Such MVVM frameworks can also be used for developing Windows Store apps.

It's also possible to use a proprietary framework from Microsoft called WinJS. The controls in WinJS (such as GridView, SemanticZoom and so on) are similar to those in XAML and support data bindings too. For more information about WinJS, see bit.ly/winjsbinding. More data-binding frameworks for JavaScript are detailed at bit.ly/jsbinding.

Project Hilo (C++ and XAML)

The third WinRT projection uses XAML as the front end and unmanaged C++ (without the Microsoft .NET Framework) for its logic. Hilo (bit.ly/hilocode) is a sample project from Microsoft patterns & practices aiming to demonstrate the various aspects of creating a Windows Store app with C++ and XAML.

> The main advantage of a separation pattern is that it assigns clearly defined responsibilities to each of the layers.

Because of the availability of XAML and data binding, you can use the MVVM pattern with unmanaged C++. Hilo demonstrates how to use MVVM in C++, among other topics such as asynchronous programming in C++; using tiles; page navigation; using touch; handling activation, suspend and resume; creating a high-performance experience; testing the app; and certification.

Hilo comes with the full source code and complete documentation. It's a good case study for traditional C++ developers wanting to get started in Windows 8 with modern development patterns, as well as for C#/Visual Basic developers who wish to use the extra performance of the C++ stack.

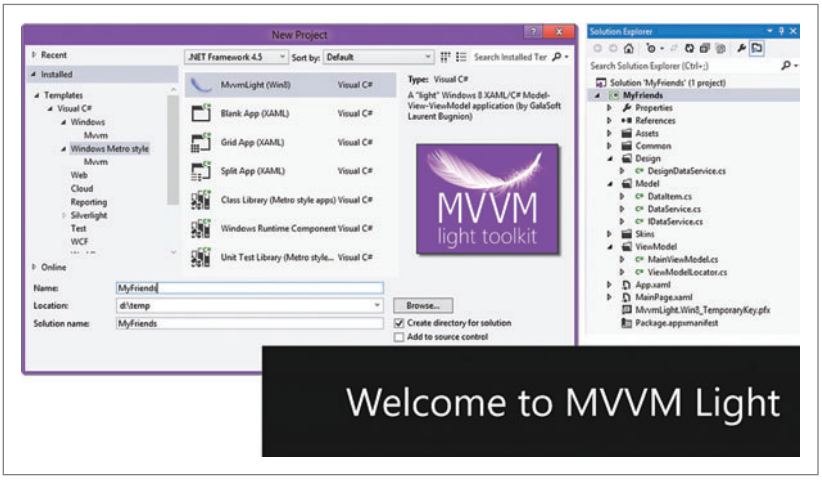


Figure 3 **Creating the New MVVM Light Application**

A Reminder About MVVM

The MVVM pattern is a variation of another well-known separation pattern named Model-View-Controller, or MVC. This pattern is used in a multitude of frameworks, notably the widely used Ruby on Rails Web application framework, as well as ASP.NET MVC by Microsoft. It's used not only in Web applications, but also widely from desktop applications to mobile apps (in iOS, for instance).

The main advantage of a separation pattern is that it assigns clearly defined responsibilities to each of the layers. The model is where the data comes from; the view is what the user is shown and what he actuates. As for the controller, it's a little like the director of an orchestra, and coordinates the actions and reactions of the application. For instance, the controller is often responsible for the coordination of actions such as displaying data in response to user input, starting and stopping animations, and so on.

> A nice consequence of the separation of concerns is that decoupled applications are also easier to test.

With these clearly defined responsibilities, members of the development team can concentrate on each part without stepping on each other's toes. Similarly, an interaction designer doesn't have to worry about the creation or persistence of the data and so on.

A nice consequence of the separation of concerns is that decoupled applications are also easier to test. For instance, classes of the model can be unit tested without having to take the view into account. Because the interaction between the model and the view is clearly defined and coordinated by the controller, it's possible to mock, or simulate, some of these interactions where needed to create consistent test conditions. This is why testability is often mentioned as one of the advantages of using the MVVM pattern in XAML-based applications.

From MVC to MVVM

While the MVC pattern has clear advantages for many frameworks, it's not the best suited for XAML-based frameworks because of (or thanks to) the data-binding system. This powerful infrastructure can bind properties of different objects and keep them synchronized. While this sounds like a simple feat, the implications are huge: By letting the data-binding system take care of this synchronization, the developer can concentrate on computing the value of data object properties without having to worry about updating the UI.

In addition, a binding is a loose coupling. The binding is only evaluated on demand, and the app doesn't break if the result of the binding is invalid. This "looseness" can sometimes cause headaches to the developers, because it makes it hard to find why some bindings are not evaluated properly. It is, however, an advantage in order to make it easier to work on the UI without being tightly coupled to a data structure. With loose bindings, it's very easy to move UI elements on the view and even to completely change an app's look and feel without touching the underlying layers.

In order to facilitate the data binding and to avoid very large objects, the controller is broken down in smaller, leaner objects called ViewModels, or Presentation Models. A common usage is to pair a view with a ViewModel by setting the ViewModel as the view's DataContext. In practice, however, this isn't always the case; it's not uncommon to have multiple views associated with a given ViewModel, or to have a complex view split in multiple ViewModels.

Thanks to the usage of XAML to build the UI, the data binding can be expressed in a declarative manner, directly in the body of

Figure 4 **Connecting to the Web Service and Deserializing the JSON Result**

```
public interface IDataService
{
  Task<IList<Friend>> GetFriends();
}

public class DataService : IDataService
{
  private const string ServiceUrl
    = "http://www.galasoft.ch/labs/json/JsonDemo.ashx";
  private readonly HttpClient _client;

  public DataService()
  {
    _client = new HttpClient();
  }

  public async Task<IList<Friend>> GetFriends()
  {
    var request = new HttpRequestMessage(
      HttpMethod.Post,
      new Uri(ServiceUrl));

    var response = await _client.SendAsync(request);
    var result = await response.Content.ReadAsStringAsync();

    var serializer = new JsonSerializer();
    var deserialized = serializer.Deserialize<FacebookResult>(result);

    return deserialized.Friends;
  }
}
```

Figure 5 **Design-Time Data Service**

```
public class DesignDataService : IDataService
{
  public async Task<IList<Friend>> GetFriends()
  {
    var result = new List<Friend>();

    for (var index = 0; index < 42; index++)
    {
      result.Add(
        new Friend
        {
          DateOfBirth = (DateTime.Now - TimeSpan.FromDays(index)),
          FirstName = "FirstName" + index,
          LastName = "LastName" + index,
          ImageUrl = "http://www.galasoft.ch/images/el20110730009_150x150.jpg"
        });
    }

    return result;
  }
}
```

the XAML document. This allows a decoupled process, where the developer worries about the model and the ViewModel, while an interaction designer takes over the creation of the UX at the same time, or even at a later time.

Finally, because XAML is XML-based, it works well with visual tools such as Expression Blend. When configured properly, MVVM makes it possible to visualize design-time data on the screen, which allows the designer to work on the UX without having to run the app, as shown in **Figure 1**.

The MVVM Light Toolkit

One disadvantage of the MVVM pattern is that some of the code required is what is sometimes called "boilerplate code"—infrastructure code that doesn't directly perform a function but is required for the internal "plumbing" to work. Probably the best example of boilerplate code is the code needed to make a property observable with the INotifyPropertyChanged interface implementation and its PropertyChanged event, as shown in **Figure 2**. This code shows that automatic properties (properties with a getter and a setter without body) cannot be used. Instead, the property has a backing field. A check is made in the setter, to avoid raising the PropertyChanged event too often. Then, if the property value does change, the PropertyChanged event is raised.

Figure 6 **Registering the Services and ViewModels**

```
static ViewModelLocator()
{
  ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

  if (ViewModelBase.IsInDesignModeStatic)
  {
    SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
    SimpleIoc.Default.Register<INavigationService,
      Design.DesignNavigationService>();
  }
  else
  {
    SimpleIoc.Default.Register<IDataService, DataService>();
    SimpleIoc.Default.Register<INavigationService>(() => new NavigationService());
  }

  SimpleIoc.Default.Register<MainViewModel>();
}
```

This is, of course, the worst-case scenario, where every property of an object needs about 20 lines of code. In addition, "magic strings" are used to identify the property's name, which can cause issues if there's a typo. In order to solve this, MVVM Light proposes a few solutions:

- The logic needed to raise the PropertyChanged event can be stored in a ViewModelBase class that every ViewModel inherits from.
- Properties can be identified by a lambda expression instead of a string. This prevents typos or errors when the name of the property changes.
- The remaining lines can be automated using a code snippet in Visual Studio.

> Thanks to the usage of XAML to build the UI, the data binding can be expressed in a declarative manner, directly in the body of the XAML document.

MVVM Light has quite a few components that are useful to make the creation of decoupled apps faster and easier, as you'll see in the sample app proposed further in this article.

Windows 8 also introduces a couple new classes to help with this problem, such as the BindableBase class and the CallerMemberName attribute. Unfortunately, these are not available for other XAML-based frameworks and cannot be used in shared code.

Creating a New Application in Windows 8

The easiest way to create a new application is to start Visual Studio 12 and to select the MvvmLight (Win8) project template. After the application is created, it's possible to start it immediately by pressing

Figure 7 **The RelayCommand Class**

```
private RelayCommand _refreshCommand;

public RelayCommand RefreshCommand
{
  get
  {
    return _refreshCommand
      ?? (_refreshCommand = new RelayCommand(ExecuteRefreshCommand));
  }
}

private async void ExecuteRefreshCommand()
{
  var friends = await _dataService.GetFriends();

  if (friends != null)
  {
    Friends.Clear();

    foreach (var friend in friends)
    {
      Friends.Add(new FriendViewModel(friend, _schema));
    }
  }
}
```

Ctrl+F5. The main page is minimal and simply displays "Welcome to MVVM Light," as shown in **Figure 3**.

What's more interesting is opening the same app in the Visual Studio Designer: In the Solution Explorer, right-click on the MainPage.xaml and select View Designer. Loading the visuals the first time takes a while, but once everything is ready, the screen shows "Welcome to MVVM Light [design]." Opening the app in Expression Blend also shows the same result: right-click on the same file and select Open in Blend to see the design-time text in the window. Note that Expression Blend for Windows Store apps is installed together with Visual Studio 2012 (even the Express edition). As with Windows Phone before, it's now possible to use Expression Blend at no cost.

What just happened has some interesting implications: The Visual Designer runs parts of the app, and there's a way to find out if the app is running in the designer or not. Seeing a different view in the Visual Designer than at run time is a crucial requirement for a good experience during the interaction design. This will allow bypassing Web service calls or database connections that wouldn't work at design time. It also allows creating well-known data—for instance, very long texts to verify how the layout looks in different orientations or resolutions without having to run the app. This saves time and a lot of trouble during the interaction design phase.

The application being developed is a viewer for a user's friends, connecting to a Web service inspired by Facebook (but heavily simplified). The Web service returns a JSON-formatted string with information about the user's friend such as first and last name, date of birth, a URI to a profile picture and so on. This information is stored in an instance of type Friend that's created in the model layer (the first "M" in MVVM). The Friend class inherits the ObservableObject class provided by the MVVM Light libraries (referenced by the new app). This allows the Friend class to easily raise the PropertyChanged event. Note that this makes sense even though Friend belongs to the model layer—it allows the UI to be data-bound to the Friend's properties without having to duplicate these properties in the ViewModel layer.

Figure 8 **The FriendViewModel Class**

```
public class FriendViewModel : ViewModelBase
{
  private FullNameSchema _schema = FullNameSchema.FirstLast;

  public Friend Model
  {
    get;
    private set;
  }

  public FriendViewModel(Friend model, FullNameSchema schema)
  {
    Model = model;

    Model.PropertyChanged += (s, e) =>
    {
      if (e.PropertyName == Friend.FirstNamePropertyName
          || e.PropertyName == Friend.LastNamePropertyName)
      {
        RaisePropertyChanged(() => FullName);
        return;
      }

      if (e.PropertyName == Friend.DateOfBirthPropertyName)
      {
        RaisePropertyChanged(() => DateOfBirthFormatted);
      }
    };

    Messenger.Default.Register<ChangeFullNameSchemaMessage>(
      this,
      msg =>
      {
        _schema = msg.Schema;
        RaisePropertyChanged(() => FullName);
      });
  }

  public string DateOfBirthFormatted
  {
    get
    {
      return Model.DateOfBirth.ToString("d");
    }
  }

  public string FullName
  {
    get
    {
      switch (_schema)
      {
        case FullNameSchema.LastFirstComma:
          return string.Format(
            "{0}, {1}",
            Model.LastName, Model.FirstName);
        default:
          return string.Format(
            "{0} {1}",
            Model.FirstName, Model.LastName);
      }
    }
  }
}
```

> Implementing observable properties requires some boilerplate code, which is annoying but mitigated by MVVM Light.

As I mentioned earlier, implementing observable properties requires some boilerplate code, which is annoying but mitigated by MVVM Light: Use a code snippet to add an INPC property (for INotifyPropertyChanged, the interface that defines the PropertyChanged event). Simply type mvvminpcset and then tab to expand the code snippet. Use tab to jump from field to field: enter the property's name (FirstName), its type (string), the backing field's name (_firstName) and, finally, the default value (string.Empty). Then type Esc to exit the snippet edit mode. In addition, the snippet adds a constant with the property's name. This will be useful later when we register to a PropertyChanged event.

All the MVVM Light snippets start with mvvm, which makes it convenient to find them in IntelliSense. There are a few other mvvminpc snippets, for variations of the property's implementation. Using mvvminpcset, you can also implement an observable property named LastName of type string.

Properties such as FirstName and LastName are quite straightforward. Sometimes, however, the data format that the app gets from

the Web service is not optimal, and a conversion needs to take place. In conventional XAML-based development, developers sometimes use a converter (implementation of IValueConverter). In MVVM, however, most converters can be replaced by simple properties. For example, let's consider the date of birth. The JSON field is in the format "MM/DD/YYYY," which is the U.S. way to express dates. However, the application might run in any locale, so a conversion is needed.

Let's start by adding an observable property of type string, named DateOfBirthString, using the same snippet mvvminpcset that was used before. Then, add another property as shown here (this property uses the DateOfBirthString as backing field; it's in charge of converting the value to a proper DateTime):

```
public DateTime DateOfBirth
{
  get
  {
    if (string.IsNullOrEmpty(_dateOfBirthString))
    {
      return DateTime.MinValue;
    }

    return DateTime.ParseExact(DateOfBirthString, "d",
      CultureInfo.InvariantCulture);
  }
  set
  {
    _dateOfBirthString = value.ToString("d",
      CultureInfo.InvariantCulture);
  }
}
```

One more thing is needed, though: Whenever the DateOfBirthString changes, the PropertyChanged event needs to be raised for DateOfBirth too. This way, data bindings will re-query the value, and force a conversion again. To do this, modify the DateOfBirthString property setter as shown in the following code:

```
set
{
  if (Set(DateOfBirthStringPropertyName, ref _dateOfBirthString, value))
  {
    RaisePropertyChanged(() => DateOfBirth);
  }
}
```

The MVVM Light ObservableObject Set method returns true if the value changed. This is the cue to raise the PropertyChanged event for the DateOfBirth property too! This conversion mechanism is quite convenient. The sample app uses it in a few places, such as to convert the profile picture URL (saved as a string) into a URI, for example.

Figure 9 **The MainViewModel SelectedFriend Property**

```
public const string SelectedFriendPropertyName = "SelectedFriend";

private FriendViewModel _selectedFriend;

public FriendViewModel SelectedFriend
{
  get
  {
    return _selectedFriend;
  }
  set
  {
    if (Set(SelectedFriendPropertyName, ref _selectedFriend, value)
        && value != null)
    {
      _navigationService.Navigate(typeof (FriendView));
    }
  }
}
```

Implementing and Mocking the Data Service

The app already has a simple DataService that can be reused to connect to the actual data service. The Web service is a simple .NET HTTP handler that returns a JSON-formatted file. Thanks to the usage of the asynchronous programming pattern introduced in the .NET Framework 4.5 (async/await keyword), connecting to the Web service is very simple, as show in **Figure 4** (together with the IDataService interface). The connection happens in an asynchronous manner, as the SendAsync method name shows. However, thanks to the await keyword, the method doesn't need a callback that would make the code more complex.

> MVVM Light also includes a simple Inversion of Control (IOC) container called SimpleIoc.

Finally, after the JSON string has been retrieved, a JSON serializer is used to deserialize the string into a .NET object of type FacebookResult. This class is just a helper that's discarded after the method returns.

MVVM Light also includes a simple Inversion of Control (IOC) container called SimpleIoc. An IOC container is a component that is useful not only in MVVM applications, but in any decoupled architecture. The IOC container acts like a cache for instances that can be created on demand, and resolved in various locations in the application.

The DataService implements the IDataService interface. Thanks to the IOC container, it's easy to mock the data service and create design-time data used in the designer. This class is called DesignDataService, as shown in **Figure 5**.

Registering and Calling the Services

Now that the services are implemented, it's time to instantiate them and create the ViewModels. This is the ViewModelLocator class task. This class is quite important in the application structure enabled by MVVM Light. It's created as a XAML resource in the file App.xaml. This creates the important link between XAML markup and source code, allowing the visual designers to create the ViewModels and to run the design-time code. The registration is shown in **Figure 6**.

Because the MainViewModel constructor takes an IDataService and an INavigationService as parameters, the SimpleIoc container is able to create all the objects automatically. The MainViewModel is exposed as a property of the ViewModelLocator and data-bound in XAML to the MainPage's DataContext. This binding is already created in the MVVM Light project template. When the page is opened in Expression Blend, the data binding is resolved, the MainViewModel instance is created (if needed) and the constructor is run, with the correct instance of the DataService (either runtime or design time).

The action of calling the data service is exposed with a RelayCommand, shown in **Figure 7**. This class is a component of the

MVVM Light Toolkit that implements the ICommand interface and offers an easy way to bind the Command property of a UI element (such as a Button) to a method implemented on the ViewModel.

The Friends collection to which the friends are added is of type ObservableCollection<FriendViewModel>. This collection type is often used in XAML-based frameworks, because any list control data-bound to such a collection automatically updates its view when its content changes. Instead of directly storing instance of the model's Friend class into the collection, they're wrapped into another class of the ViewModel layer called FriendViewModel. This allows adding view-specific properties that won't be persisted in the model. For instance, the FriendViewModel class exposes a property named FullName. Depending on the application's settings, it can return a string in the format "FirstName, LastName" or "LastName, FirstName." This kind of logic is typical from ViewModel classes and shouldn't be stored in the lower layers of the application.

In order to return the FullName property in the correct format, the FriendViewModel listens to a message of type ChangeFullNameSchemaMessage. This is done thanks to the Messenger component of MVVM, a loosely coupled event bus connecting a sender and a series of recipients without creating a strong connection between them. Note that the Messenger can send any message type, from simple values such as an int to complex objects with additional information.

When such a message is received, the PropertyChanged event is raised for the FullName property. The FriendViewModel class is shown in **Figure 8**.

The instruction to switch from "FirstName, LastName" to "LastName, FirstName" is sent by the MainViewModel as shown here:

```
private void SetSchema(FullNameSchema schema)
{
  _schema = schema;
  Messenger.Default.Send(new ChangeFullNameSchemaMessage(_schema));
}
```

Because of the decoupled manner with which the Messenger class is operating, it would be very easy to move it from the MainViewModel into a Settings class at a later point, for example.

Setting up Navigation

Navigation between pages in Windows 8 is quite easy when it's initiated from the view's codebehind, thanks to the NavigationService property that every Page exposes. In order to navigate from the ViewModel, however, a little setup is needed. It's quite easy, because the Frame that's responsible for the navigation is exposed in a static manner as ((Frame)Window.Current.Content). The application can expose a standalone navigation service:

```
public class NavigationService : INavigationService
{
  public void Navigate(Type sourcePageType)
  {
    ((Frame)Window.Current.Content).Navigate(sourcePageType);
  }

  public void Navigate(Type sourcePageType, object parameter)
  {
    ((Frame)Window.Current.Content).Navigate(sourcePageType, parameter);
  }

  public void GoBack()
  {
    ((Frame)Window.Current.Content).GoBack();
  }
}
```

The code in **Figure 8** shows how the navigation service is registered with the SimpleIoc container. It's injected as a parameter in the MainViewModel constructor, and can be used to conveniently start a UI navigation directly from the ViewModel layer.

In the MainViewModel, the navigation is initiated when the property called SelectedFriend changes. This property is an observable property just like others that were set up earlier. This property will be data-bound in the UI, so the navigation is initiated by the user's actions. To display the friends, a GridView is used, and it's data-bound to the Friends collection on the MainViewModel. Because this ViewModel is set as the MainPage DataContext, the binding is easy to create, as shown here:

```
<GridView
  ItemsSource="{Binding Friends}"
  ItemTemplate="{StaticResource FriendTemplate}"
  SelectedItem="{Binding SelectedFriend, Mode=TwoWay}"/>
```

Another TwoWay binding is created between the GridView SelectedItem property and the MainViewModel SelectedFriend property listed in **Figure 9**.

Finally, the navigation leads the user to a details page for the selected Friend. In this page, the DataContext is bound to the MainViewModel SelectedFriend property. This, again, ensures a good design-time experience. Of course, at design time the RefreshCommand is executed when the MainViewModel is constructed, and the SelectedFriend property is set:

```
#if DEBUG
private void CreateDesignTimeData()
{
  if (IsInDesignMode)
  {
    RefreshCommand.Execute(null);
    SelectedFriend = Friends[0];
  }
}
#endif
```

At this point, the UX can be designed in Expression Blend. Because the design-time code is run, Blend displays all the design-time friends in a GridView added to the MainPage, as was shown in **Figure 1**. Of course, creating the data templates takes some time and skill, but it can be easily done in a visual manner without having to run the application.

Wrapping Up

All of this may seem quite familiar to developers familiar with the MVVM pattern in WPF, Silverlight or Windows Phone. That's because things are similar with the Windows Runtime. The skills acquired in those previous frameworks are easily transferable to Windows Store app development. Of course, some concepts (such as the asynchronous programming with async/await) are new, and some work is needed to convert code to the Windows Runtime. But with the MVVM pattern and helpers such as the MVVM Light Toolkit, developers get a good head start and can fully enjoy the advantages of a decoupled application pattern. ■

Laurent Bugnion *is senior director for IdentityMine Inc., a Microsoft partner working with technologies such as Windows Presentation Foundation, Silverlight, Surface, Windows 8, Windows Phone and UX. He's based in Zurich, Switzerland.*

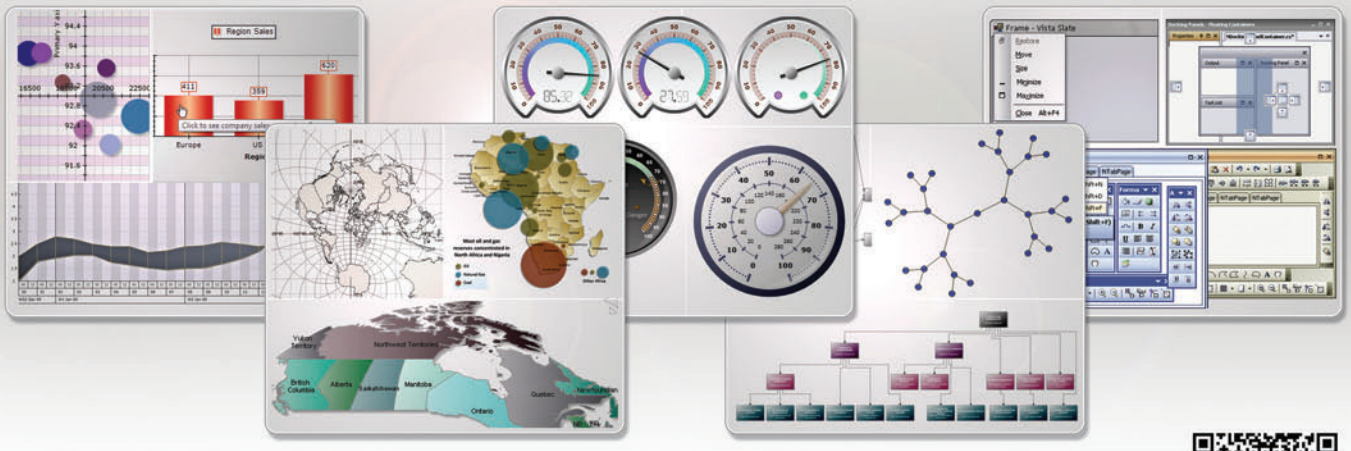
Thanks to the following technical experts for reviewing this article: *Karl Erickson and Rohit Sharma*

NEVRON
visualize your success

NEVRON
GAUGE for .NET, SSRS, SharePoint

NEVRON
CHART for .NET, SSRS, SharePoint

NEVRON
USER INTERFACE for .NET



NEVRON
MAP for .NET

NEVRON
DIAGRAM for .NET

2012vol.1 is here

Features new ThinWeb controls with JQuery and ASP.NET MVC integration, SQL Server and SharePoint "Expressions Everywhere" and more.

Nevron components integrate seamlessly in **Web** and **Desktop .NET** applications, **SQL Server Reporting Services 2005/2008** reports and **SharePoint 2007/2010** portals and deliver an unmatched set of enterprise-grade features. That is why Nevron is the trusted vendor by many Fortune 500 companies for their most demanding data visualization needs.

Download your free evaluation from

www.nevron.com

DEVELOPERS

.NET

IT PROFFESIONALS

SharePoint

SSRS

Microsoft .NET ASP.net AJAX Microsoft SharePoint 2007/2010 Microsoft SQL Server 2005/2008 Microsoft Visual Studio PARTNER

Introducing C++/CX and XAML

Andy Rich

Code First is great for designing a fast sorting algorithm, but it isn't necessarily the best way to make a good-looking UI, and the UI technology provided by Microsoft for C++ developers has reflected this preference. The last significant upgrade to Microsoft UI technology for C++, released in 2008, was a new set of MFC controls that provided styled Ribbon controls. Before that was Windows Forms, a Microsoft .NET Framework-based UI technology first released with Visual Studio .NET in February 2002. While powerful, both of these technologies were primarily focused on the underlying code that created the UI elements and demanded tight coupling of data and the UI.

In recent years, C# and Visual Basic developers have enjoyed significant upgrades to their UI technology in the form of XAML-based UI frameworks (Windows Presentation Foundation, or WPF, and Silverlight). These frameworks brought developers new options in UI design, giving them the freedom to code their algorithms without worrying about how the UI would be presented, and the freedom to design their UI without worrying about the code behind it. At last, with Windows 8, C++ developers can take advantage of a modern, XAML-based UI framework to build applications for Windows 8.

Why XAML?

XAML is an XML-based markup language that allows you to define the look of your application without needing to understand how the UI is built in code. The Windows Runtime (WinRT) parses this information at run time, constructs the appropriate control classes and builds the UI tree. Your code can manipulate these objects and, when authored correctly, the look and feel of the UI can be significantly altered without touching the codebehind at all. This facilitates a division of labor between a developer who works on the code behind the scenes and a UI designer who focuses on the look and feel of the application itself.

Visual Studio 2012 also is tightly integrated with Blend, a powerful designer tool specifically optimized for working with XAML and HTML. Blend provides more WYSIWYG control (including animation design) and exposes more properties on XAML objects than the Visual Studio integrated designer.

This article discusses:

- Support for building Windows Store apps with C++ and XAML
- Using C++ with the Windows Runtime
- Data binding in XAML

Technologies discussed:

XAML, C++/CX, Visual Studio 2012, Window 8

Code download:

archive.msdn.microsoft.com/mag201210XAMLCX

Why C++/CX?

Developers have a range of technologies to choose from when writing a Windows Store app: HTML and JavaScript, C# or Visual Basic with XAML, and C++. In C++, you can build apps with C++/CX and XAML or DirectX, or a combination of both. In addition, you can create an application that's a hybrid of these technologies, such as writing a component in C++/CX that's then used from a Windows Store app built with JavaScript/HTML.

There are a number of reasons for you to choose C++ for writing a Windows Store app. First, the C++/CX binding to the Windows Runtime is 100 percent native and reference-counted. This means there's very little overhead. Second, C++ has the ability to leverage existing libraries (such as Boost). Finally, C++ supports native Direct3D and Direct2D surfaces inside Windows Store apps, which enables you to build games and other graphically intensive applications with great performance. Having low overhead and high performance doesn't only mean that your application runs faster; it can also be leaner, using fewer instructions to accomplish the same tasks, which reduces power consumption and gives your users better battery life.

Hello World from C++/CX and XAML

Visual Studio 2012 provides a number of templates that demonstrate advanced features of Windows Store apps. The Grid template is a feature-packed application with multiple views, support for landscape and portrait transitions, advanced data binding, and suspend and

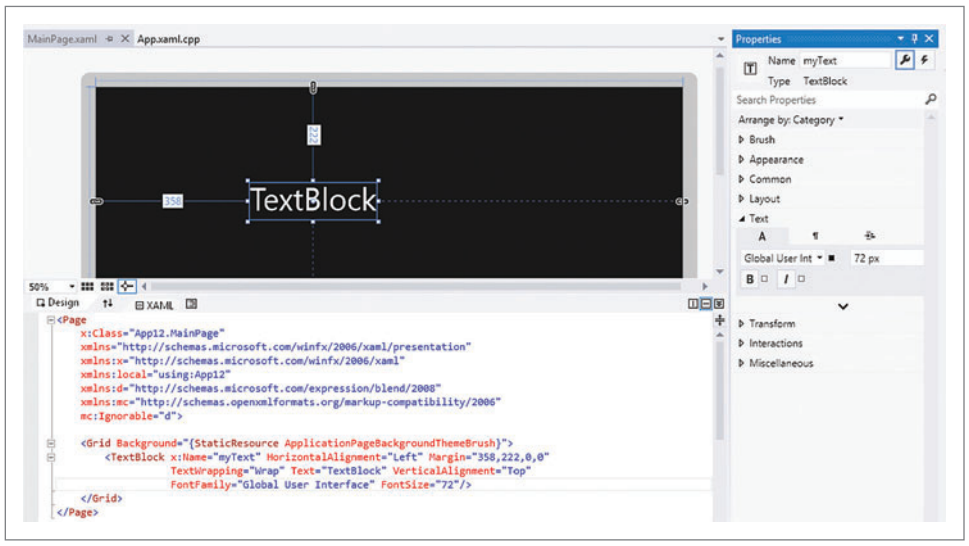


Figure 1 **TextBlock Control and Related XAML**

resume capabilities. However, it isn't very suitable for demonstrating the basics. Instead, create a new C++ Windows Store app using the Blank App (XAML) template.

Once the application is created, find MainPage.xaml in Solution Explorer and open it. It opens for editing in the integrated XAML designer, which is split into two panes: the Design view shows a rendering of the UI, and the XAML view shows the code. Updating either pane updates the other as well. This gives you the freedom to drag controls for ease of editing or to precisely control the layout by editing the code.

Open the toolbox, expand Common XAML Controls and drag a TextBlock control onto the design surface. Open the Properties window, and you'll find options for styling this instance of the TextBlock control. In the Name field, name the TextBlock "myText." This correlates to the x:Name attribute in XAML. This attribute allows you to refer to the TextBlock by name, both elsewhere in the XAML file and in the codebehind. And, because this is the only control on the page, you might as well make it very large: expand the Text category and change the font size to 72px. The end result should resemble **Figure 1**. You'll notice that as you make these changes in the property inspector, the XAML markup is also updated.

Next, drag a Button control from the toolbox and put it to the left of the TextBlock. In the Properties window, view the available events for the Button control by clicking the lightning-bolt icon to the right of the Name field, and then double-click the field for the Click event. Visual Studio automatically generates an appropriate callback and switches the view to the codebehind for this event. In the XAML source view, the generated tag should look something like the following:

```
<Button Click="Button_Click_1" />
```

You could accomplish the same thing in code by adding the Click= attribute for the XAML tag and then providing a name or allowing autocomplete to select one for you. Either way, you can right-click the event and choose Navigate To Event Handler to navigate to the codebehind.

Now add the following code to the callback:

```
this->myText->Text =
  "Hello World from XAML!";
```

Build and run the application to verify that when you click the button, the callback modifies the text in the TextBlock (**Figure 2**).

A lot goes on behind the scenes here, so I want to spend some time deconstructing the magic that makes this happen.

What Is C++/CX?

C++/CX is the language extension that provides interoperability between native C++ and the Windows Runtime. It's a fully native binding that enables you to define, instantiate and use objects in the runtime while still providing access to native C++ functionality and syntax with which you're familiar. **Figure 3** lists the basic concepts. (Although C++/CX is similar in many ways to the C++/Common Language Infrastructure, or CLI, syntax, it's a separate and wholly native language extension.)

Partial Classes and XAML Code Generation In Solution Explorer, navigate to MainPage.xaml.h, which contains a partial definition of the MainPage XAML class:

```
public ref class MainPage sealed
{
public:
  MainPage();

protected:
  virtual void OnNavigatedTo(
    Windows::UI::Xaml::Navigation::NavigationEventArgs^ e) override;
private:
  void Button_Click_1(
    Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e);
};
```

This is the portion of the definition that you can edit and modify (for example, if you want to add some data members to the MainPage

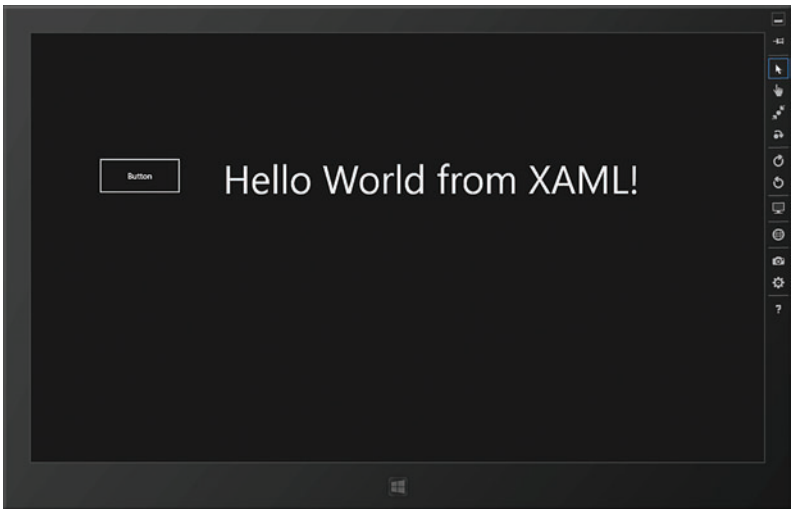


Figure 2 **Updated TextBlock**

Figure 3 **C++/CX Concepts**

Concept	Description
interface class	An abstract declaration of the methods that an object implementing the interface will support calls into.
ref class	C++/CX keyword that's used to declare a reference-counted class in the Windows Runtime that's used for implementing interfaces.
^	Called a "hat" or "handle," this is a smart pointer for use with ref classes and interfaces; it automatically increments and decrements reference counts appropriately.
ref new	Expression that "activates" (constructs) a WinRT class.
default namespace	Contains fundamental type definitions (uint32, float64) that map to WinRT types.
Platform namespace	Contains type definitions that provide the basic interoperability model with the Windows Runtime (Object^, String^).
Property	A pseudo-field on a class; it's composed of a getter function and an optional setter function and behaves like a field. (For example, a TextBlock object has a Text property.)

class). Note that the declaration of the Button_Click_1 event handler was inserted into this class definition by Visual Studio when you double-clicked the Click event earlier.

Right-click on the class name MainPage and choose Go To Definition. You should see two results: the definition in MainPage.xaml.h and a partial definition in MainPage.g.h. The second definition is generated during a build when the XAML compiler parses MainPage.xaml and generates MainPage.g.h. MainPage.xaml.cpp includes MainPage.xaml.h, which in turn includes this generated file. These class definitions are combined to produce the final class. This technique permits both users and the code generator to inject information into the class definition.

Navigate to the partial definition of MainPage found in MainPage.g.h:

```
partial ref class MainPage : public ::Windows::UI::Xaml::Controls::Page,
  public ::Windows::UI::Xaml::Markup::IComponentConnector
{
public:
  void InitializeComponent();
  virtual void Connect(int connectionId, ::Platform::Object^ target);

private:
  bool _contentLoaded;
  private: ::Windows::UI::Xaml::Controls::TextBlock^ myText;
};
```

You can see that in this generated file, the x:Name="myText" attribute on the TextBlock has caused the XAML compiler to generate a class member named myText. Here you can also observe that MainPage inherits from the Windows XAML base class Page and implements the interface IComponentConnector. (IComponentConnector and its associated method Connect are what wire up the Click event to the Button_Click_1 callback.)

Figure 4 **Definition of the Person Class**

```
// person.h
#pragma once
namespace App12 {
  [Windows::UI::Xaml::Data::Bindable]
  public ref class Person sealed
  {
  public:
    Person(void);
    property Platform::String^ Name;
    property Platform::String^ Phone;
  };
}

// person.cpp
#include "pch.h"
#include "person.h"
using namespace App12;

Person::Person(void)
{
  Name = "Maria Anders";
  Phone = "030-0074321";
}
```

Data Binding in C++/CX Data binding is a powerful concept that enables you to easily wire up code and data to your UI. It also enables high-level design patterns such as Model-View-ViewModel (MVVM) that provide excellent abstraction between data, code and the UI.

In C++/CX, you can enable a class to be data-bound in a number of ways:

- Attribute the class with the Bindable attribute
- Implement the ICustomPropertyProvider interface
- Implement the IMap<String^, Object^> interface

I'll focus on the Bindable attribute, but I've mentioned the other interfaces because in some scenarios they're more appropriate. In particular, the Bindable attribute generates appropriate data-binding information only if the data-bound class is marked public. Whenever you want a private class to be bindable, you need to implement one of the other interfaces listed.

In your application, add a new class by right-clicking the project and choosing Add, Class to bring up the Add Class Wizard. There's no option for a C++/CX class (which is what you need), so add a native class, naming it Person. Visual Studio should generate a Person.h and Person.cpp class and add them to your project.

The Person class needs several modifications (demonstrated in **Figure 4**):

- The class definition is changed from class to ref class.
- The ref class is marked as public and sealed. "Public" is necessary for the XAML compiler to find this class and generate data-binding information. Also, C++/CX doesn't support public unsealed classes without a base class, so the class must be sealed.
- The destructor ~Person(void) is removed because no destructor is necessary for this class.
- The class is moved into the namespace used by the application (in the example shown in **Figure 4**, the namespace is "App12," but this will vary based on the name you gave your project). For the Bindable attribute to work (and to satisfy WinRT metadata rules), the class must be in a namespace.
- The class is attributed with [Windows::UI::Xaml::Data::Bindable]. This attribute indicates to the XAML compiler that it should generate data-binding information for the class.

Visual Studio LIVE!

EXPERT SOLUTIONS FOR .NET DEVELOPERS

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Orlando, FL **December 10-14**

Royal Pacific Resort at Universal Orlando | **vslive.com/orlando**

Visual Studio Live! provides education and training on what's now, new and next on the .NET development platform.

Session Topics Include:

- HTML5
- Windows 8 / WinRT
- Mobile
- WPF / Silverlight
- Visual Studio / .NET
- XAML

Save Up To $400!

Register Before November 7

Use Promo Code VSOCT

vslive.com/orlando

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Buy 1 Event, Get 3 Free!

Visual Studio Live! is co-located with:

SharePoint LIVE! TRAINING FOR COLLABORATION

SQL Server LIVE! TRAINING FOR DBAs AND IT PROS

Cloud & Virtualization LIVE! THE FUTURE OF COMPUTING

GOLD SPONSOR axosoft

SUPPORTED BY Microsoft msdn Visual Studio MAGAZINE Redmond MAGAZINE

MEDIA SPONSOR THE CODE PROJECT WWW.CODEPROJECT.COM

PRODUCED BY 1105 MEDIA

- Two properties of type Platform::String^: Name and Phone are added. (Only properties can be data-bound.)

Now, in MainPage.xaml, add a second TextBlock below the first and style it similarly. In the XAML, change the Text attribute of the two TextBlocks to Text="{Binding Name}" and Text="{Binding Phone}", respectively. This tells the UI to find properties named Name and Phone on the data-bound object and apply them to the Text field of the class. The XAML should look similar to **Figure 5**. (When the Text attributes are changed to be data-bound, they no longer have contents when they're shown in the designer; this is expected.)

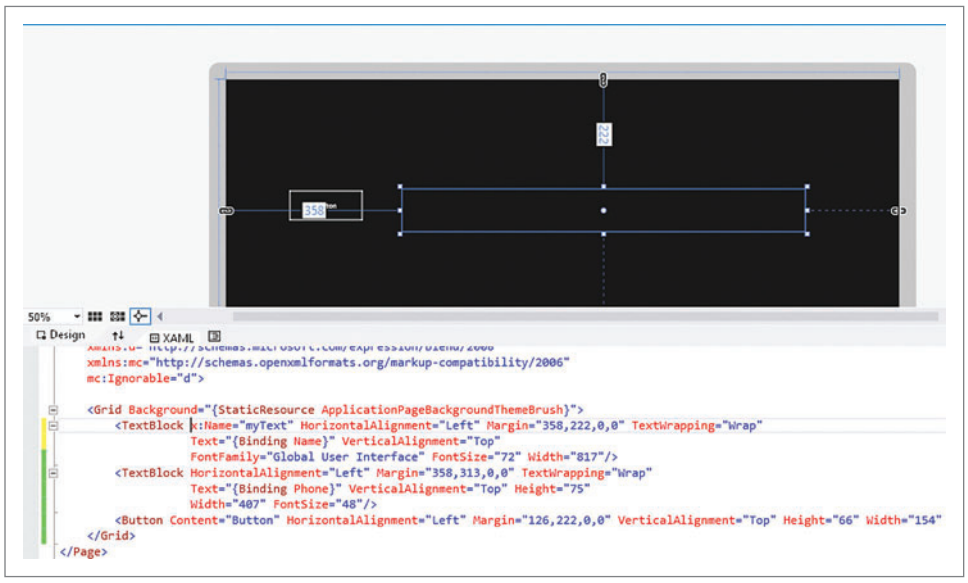


Figure 5 **Data Binding in XAML**

Next, in MainPage.xaml.h, include Person.h and add a private datamember of type Person^ to the MainPage class. Name it m_Person. Finally, in MainPage.xaml.cpp, in the OnNavigatedTo method, add the following code:

```
m_Person = ref new Person;
this->DataContext = m_Person;
```

The XAML runtime invokes the OnNavigatedTo method just before this page is displayed on the screen, so it's an appropriate place to set up this datacontext. This code will create a new instance of the Person class and bind it to your MainPage object. Run the application again, and you should see that the Name and Phone properties have been applied to the TextBlock Text attributes.

The data-binding infrastructure also provides for the data-bound objects to be notified if the property's contents change. Change the Button_Click_1 method to the following code:

```
this->m_Person->Name = "Anders, Maria";
```

If you run the application again and click the button, you'll see that the TextBlock contents do not change. For these contents to be updated, the Person class needs to implement the INotifyPropertyChanged interface.

INotifyPropertyChanged has one item in it: an event named PropertyChanged, of type PropertyChangedEventHandler^. The Name property also needs to be changed from a trivial property to a fully defined property (including a user-provided backing store) to support firing this event when the property is modified. In addition, I added a convenience function named NotifyPropertyChanged that fires this event when it's called. This is a commonplace refactoring, because multiple properties often require notification. The implementation is shown in **Figure 6**. (This example only provides for the Name property to be notifiable. A similar change would need to be made for the Phone property.)

Running the application again with the INotifyPropertyChanged modifications made should result in the name change when the button is clicked.

Figure 6 **Person Class with Notifiable Name Property**

```
// person.h
#pragma once
namespace App12 {
  namespace WUXD = Windows::UI::Xaml::Data;
  [WUXD::Bindable]
  public ref class Person sealed : WUXD::INotifyPropertyChanged
  {
  public:
    Person(void);
    property Platform::String^ Name {
      Platform::String^ get();
      void set(Platform::String^ value);
    }
    property Platform::String^ Phone;
    virtual event WUXD::PropertyChangedEventHandler^ PropertyChanged;
  private:
    void NotifyPropertyChanged(Platform::String^ prop);
    Platform::String^ m_Name;
  };
}

// person.cpp
#include "pch.h"
#include "person.h"
using namespace App12;
using namespace Platform;
using namespace WUXD;

Person::Person(void)
{
  Name = "Maria Anders";
  Phone = "030-0074321";
}

String^ Person::Name::get(){ return m_Name; }
void Person::Name::set(String^ value) {
  if(m_Name != value) {
    m_Name = value;
    NotifyPropertyChanged("Name");
  }
}

void Person::NotifyPropertyChanged(String^ prop) {
  PropertyChangedEventArgs^ args = ref new PropertyChangedEventArgs(prop);
  PropertyChanged(this, args);
}
```

SharePoint LIVE!

TRAINING FOR COLLABORATION

LIVE! 360

IT EVENTS WITH PERSPECTIVE

Orlando, FL December 10-14

Royal Pacific Resort at Universal Orlando | **splive360.com**

SharePoint Live! provides training for those who must customize, deploy and maintain SharePoint Server and SharePoint Foundation to maximize the business value. Both current & newly released versions will be covered!

Session Topics Include:

- Strategy, Governance, Adoption
- Management and Administration
- Information & Content Management
- BI, BPA, Search, and Social
- SharePoint and the Cloud
- SharePoint 2010 & SharePoint 2013

Save Up To $400!

Register Before November 7

Use Promo Code SPOCT

splive360.com

LIVE! 360

IT EVENTS WITH PERSPECTIVE

Buy 1 Event, Get 3 Free!

SharePoint Live! is co-located with:

Cloud & Virtualization LIVE! — THE FUTURE OF COMPUTING

SQL Server LIVE! — TRAINING FOR DBAs AND IT PROS

Visual Studio LIVE! — EXPERT SOLUTIONS FOR .NET DEVELOPERS

GOLD SPONSOR axosoft

SUPPORTED BY Microsoft msdn VIRTUALIZATION REVIEW Redmond MAGAZINE

MEDIA SPONSOR THE CODE PROJECT WWW.CODEPROJECT.COM

PRODUCED BY 1105 MEDIA

Figure 7 **ListView Inside MainPage with Styled ItemTemplate**

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <ListView x:Name="myListView">
    <ListView.ItemTemplate>
      <DataTemplate>
        <StackPanel Orientation="Vertical">
          <TextBlock Text="{Binding Name}" FontSize="20" />
          <TextBlock Text="{Binding Phone}" FontSize="12" />
        </StackPanel>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</Grid>
```

ItemsControls and Collections in C++/CX and XAML

XAML provides rich UI controls, including controls that can data bind to a collection of objects. These are collectively referred to as ItemsControls. I'll focus on one in particular, called ListView.

Open MainPage.xaml and delete the contents of the root Grid object, replacing it with just a ListView class. Name that class myListView, as shown here:

```
<!-- ListView inside MainPage -->
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">*
  <ListView x:Name="myListView" />
</Grid>
```

Collections in C++/CX The Windows Runtime does not provide collection classes; it only provides interfaces for the collections. Each projection (JavaScript, C#/Visual Basic and C++/CX) is responsible for providing its own collection classes. In C++/CX, these implementations are found in the collection.h header (which is already included in the pch.h of the Blank application template). There are two major collection classes defined in the Platform::Collections namespace: Platform::Collections::Vector<T> for representing lists of data, and Platform::Collections::Map<T,U> for representing dictionaries. ItemsControls generally expect an iterable list of objects, so Vector<T> is appropriate in this scenario.

In MainPage.xaml.cpp, change the OnNavigatedTo method to "ref new" a Vector<Person^> and fill it with several Person classes. Then assign it to the ItemsSource property of myListView:

```
void MainPage::OnNavigatedTo(NavigationEventArgs^ e)
{
  (void) e; // Unused parameter
  auto vec =
    ref new Platform::Collections::Vector<Person^>;
  vec->Append(ref
    new Person("Maria Anders","030-0074321"));
  vec->Append(ref new Person("Ana Trujillo","(5) 555-4729"));
  // etc. ...

  myListView->ItemsSource = vec;
}
```

Styling the ListView Finally, we need to tell the ListView how to style each individual element. The ListView will then iterate over its ItemsSource and, for each element in the collection, generate the appropriate UI and bind its DataContext to that element. To style the ListView, you must define its ItemTemplate, as shown in **Figure 7**. Note that the Text properties inside the template have the data bindings appropriate for a Person object.

When you build and run this application, it should resemble **Figure 8**.

Find Out More

References on XAML for Windows 8 and C++/CX are somewhat scarce at the moment. To learn more about XAML, I recommend looking for references that talk about XAML in Silverlight, as many of the same concepts apply and many controls are named similarly or the same and have the same functionality. The codebehind for these references will be C#, so some translation or interpretation is necessary. This is chiefly a mechanical operation, but where you run into functionality that's represented by the .NET Base Class Library, you need to look to the C Runtime Libraries, Standard Template Library or other C++ libraries (or the Windows Runtime) to accomplish these tasks. Fortunately, all of these libraries play nicely with C++/CX.

Other great resources are the templates available in Visual Studio 2012 for creating Windows Store apps, in particular the Grid App and Split App. These templates demonstrate many advanced features of a well-made application, and deciphering their secrets can lead to greater understanding of the underlying frameworks.

Finally, I recommend that you look at project "Hilo," a large-scale application written using C++ and XAML. The code and documentation for this application can be found at hilo.codeplex.com. This project was developed by the Microsoft patterns & practices team and demonstrates modern, standard C++; the porting of existing C++ libraries so that they can be used in Windows Store applications; and many best practices when using C++/CX.

I hope I've given you a glimpse at what you can do with C++/CX and with XAML, but this is just the surface. As with any new technology, you can expect to spend many late nights browsing documentation and forums and with your nose buried in technical books. But the power and capability of C++/CX and XAML are worth learning more about. They will enable you to write rich, fast applications in far less time. ■

ANDY RICH *is a software development engineer at Microsoft and has spent the last nine years testing the C++ front-end compiler. Most recently, he was one of the primary testers of the C++/CX language extensions and worked closely with the XAML team in testing the combined XAML/C++ scenario.*

THANKS to the following technical experts for reviewing this article: *Tim Heuer, Marian Luparu and Tarek Madkour*

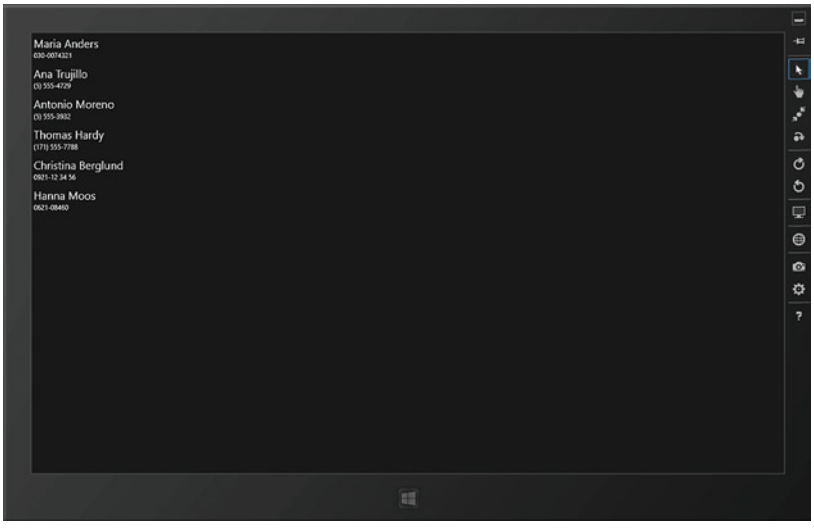


Figure 8 **The Finished ListView**

SQL Server® LIVE!

TRAINING FOR DBAs AND IT PROS

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Orlando, FL December 10-14

Royal Pacific Resort at Universal Orlando | **sqllive360.com**

SQL Server Live! provides comprehensive education on SQL Server database management, data warehouse/BI model design, Big Data, analytics, performance tuning, troubleshooting and coding against SQL Server.

Session Topics Include:

- BI and Big Data
- Hadoop
- T-SQL, SSIS Packages, Disaster Recovery
- Monitoring, Maintaining, Tuning
- SQL Azure & Windows Azure
- SQL Server 2012

Save Up To $400!

Register Before November 7

Use Promo Code SQOCT

sqllive360.com

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Buy 1 Event, Get 3 Free!

SQL Server Live! is co-located with:

SharePoint LIVE! TRAINING FOR COLLABORATION

Cloud & Virtualization LIVE! THE FUTURE OF COMPUTING

Visual Studio LIVE! EXPERT SOLUTIONS FOR .NET DEVELOPERS

GOLD SPONSOR axosoft

SUPPORTED BY Microsoft msdn VIRTUALIZATION REVIEW Redmond MAGAZINE

MEDIA SPONSOR THE CODE PROJECT WWW.CODEPROJECT.COM

PRODUCED BY 1105 MEDIA

Porting Desktop Applications to the Windows Runtime

Diego Dagum

Windows 8 embodies a new design philosophy for Microsoft platforms. In Windows 8, you can create apps using UI technologies such as XAML and HTML5. Microsoft provides two new app models: the Windows Runtime Library (WRL), which helps you develop Windows Store apps with C#, Visual Basic and C++, and the Windows Library for JavaScript (WinJS), which lets you create HTML5 and JavaScript apps.

The WRL is to Windows 8 what the Microsoft Foundation Class framework (MFC) or the C-like Win32 APIs are to the desktop environment. Consequently, existing desktop applications must be adapted to run under the Windows Runtime. The dilemma comes with applications that depend heavily on MFC, Win32 or other application frameworks. What happens with them if they're ported to the Windows Runtime? What happens thereafter? Is it necessary to maintain both codebases—tablet and desktop?

In this article, I'll show you how you can identify and extract substantial portions from the application codebase and share them between the two environments. You'll see how this refactoring activity is also an opportunity to leverage some new C++11 features for conciseness and maintainability. And the benefits are more than just gaining a new Windows Store version of an existing app; the existing application codebase is upgraded as well.

This article discusses a prerelease version of Visual Studio 2012. All information is subject to change.

This article discusses:

- Problems involved in porting
- Separation of concerns
- Refactoring to decouple reusable components
- Creating a XAML view of the app
- Using HTML for the app UI

Technologies discussed:

Windows 8, Visual Studio 2012, C++, XAML, HTML, JavaScript, Windows Runtime

Code download available at:

archive.msdn.microsoft.com/mag201210CPP

Portability and Its Dilemmas

It's always easier to build an application with portability and other non-functional requirements from scratch, architecting it accordingly. In practice, though, application developers are often challenged by unforeseen requirements that emerge after the application has been deployed. Satisfying these later needs may prove problematic if the application is architected in a way that makes new features difficult to implement without rewriting large portions. The new parts can eventually lead to an application breakage in production if not carefully tested.

Because of this, I decided to use as an example an existing application instead of creating my own demo. As you'll see, I chose an MFC calculator sample published by Microsoft for Visual Studio 2005 (bit.ly/00494l).

Cloud & Virtualization LIVE!

THE FUTURE OF COMPUTING

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Orlando, FL **December 10-14**

Royal Pacific Resort at Universal Orlando | **virtlive360.com**

Cloud & Virtualization Live! is a new event that provides in-depth training with real-world applicability on today's cloud and virtualization technologies.

Session Topics Include:

- Becoming a Virtualization Expert
- Managing the Modern Cloud Enabled Datacenter
- Tactics in Cloud Application Development
- Constructing & Managing an Application Delivery Infrastructure
- Integrating Mobile Devices and Consumerization with Cloud Computing

Save Up To $400!

Register Before November 7

Use Promo Code CVOCT

virtlive360.com

LIVE! 360
IT EVENTS WITH PERSPECTIVE

Buy 1 Event, Get 3 Free!

Cloud & Virtualization Live! is co-located with:

SharePoint LIVE! TRAINING FOR COLLABORATION

SQL Server LIVE! TRAINING FOR DBAs AND IT PROS

Visual Studio LIVE! EXPERT SOLUTIONS FOR .NET DEVELOPERS

GOLD SPONSOR axosoft

SUPPORTED BY Microsoft msdn VIRTUALIZATION REVIEW Redmond MAGAZINE

MEDIA SPONSOR THE CODE PROJECT WWW.CODEPROJECT.COM

PRODUCED BY 1105 MEDIA

The option of rewriting the entire application seems appealing at first, because you want to get rid of code you don't want to maintain—you'd rather redo it from scratch, but this time do it well. But management can't be convinced because it erodes the return on investment (ROI) expected from the original application, unless that app stays in production for users whose platforms can't run the new application. If this is the case, two similar codebases will need to be maintained, increasing costs (twice the work—or more—to implement new features or fix bugs, for example).

It's unlikely that all of the original code can be reused in different environments (Windows desktop and the Windows Runtime, in this case). However, the more that can be shared, the lower the costs and, consequently, the higher the profits.

Figure 1 **The Microsoft MFC Calculator**

Back to the Basics: Separation of Concerns

Separation of concerns (SoC) is an established concept today, published in many software architecture books. One of its natural consequences is that API-related code is cohesively grouped (not to say hidden) into well-segmented components that offer an abstract interface to the rest. Thus, a concrete data repository is never exposed explicitly to code that performs domain logic, presentation logic and so forth. These components just "talk" to the abstract interface.

SoC is nowadays widely adopted among developers. As a consequence of the Web explosion that began in the late '90s, many standalone applications were broken up in modules that were then distributed across layers and tiers.

If you have applications developed without regard for SoC, you can bring them into the modern world by refactoring. Refactoring is a healthy practice performed nowadays thanks to the wide adoption of Agile practices, which promote building the simplest things that can possibly work, getting all tests passed and maximizing software throughput, time to market and so forth. However, agility doesn't leave much room for enabling new channels until this becomes a need. Here we are, then.

A Typical Porting Scenario

The sample MFC application I mentioned earlier, a basic calculator, is shown in **Figure 1**.

This sample is great to illustrate the porting process for the following reasons:

- It's small enough for you to get the overall idea of what it does.
- It's large enough to allow me to show the process in detail. The average application will probably have a larger codebase, but the porting will consist of a repetition of the steps I'll describe here.
- The code is coupled enough to show decoupling through refactoring. It was probably intentionally coupled to keep the codebase compact and comprehensible. I'll decouple the code until I get a common codebase that's used by MFC and Windows 8 versions. I won't decouple further, but I'll suggest how much more the code can be decoupled.

The original calculator application contains two classes: CCalcApp and CCalcDlg. CCalcApp models the calculator as a running process whose InitInstance function instantiates a CCalcDlg (see **Figure 2**). CCalcDlg models the main window, its controls (panel and buttons) and associated events.

CCalcDlg derives from MFC CDialog and its implementation does everything, from its basic mapping of window messages, to its functions and the implementation of the calculator logic triggered as a response to these events. **Figure 3** shows what happens when the equal sign button is clicked (presumably after two operands and an operator were entered). **Figure 4** shows the CCalcDlg functions that add behavior at all levels: event reaction, domain logic and presentation.

Because CCalcDlg is tied to MFC, the logic I want to use in the Windows Store version of the application can't be ported as is. I'll have to do some refactoring.

> If you have applications developed without regard for SoC, you can bring them into the modern world by refactoring.

Refactoring to Decouple Reusable Components

To create a Windows Store version of this calculator, I don't need to recode everything. Much of the behavior, such as what's shown in **Figure 4**, could be reused if it were not so tied to the MFC-based CCalcDlg. I'll refactor the application in a way that reusable parts (the calculator behavior in this case) are isolated from implementation-specific components.

I'll assume you've not only heard about the Model-View-Controller (MVC) architecture pattern but have also applied it. I'll just recap the pattern here: the Model consists of domain objects (both stateless and not) and doesn't know or care about the View technology. The View is implemented in some user-interaction technology

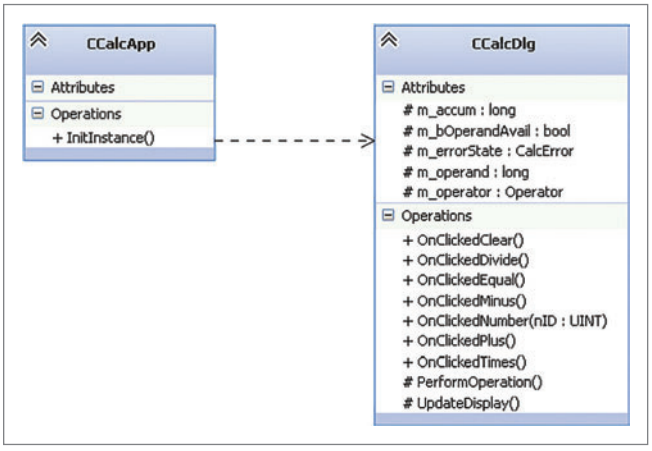


Figure 2 **Original Calculator Sample Class Diagram Showing Essentials**

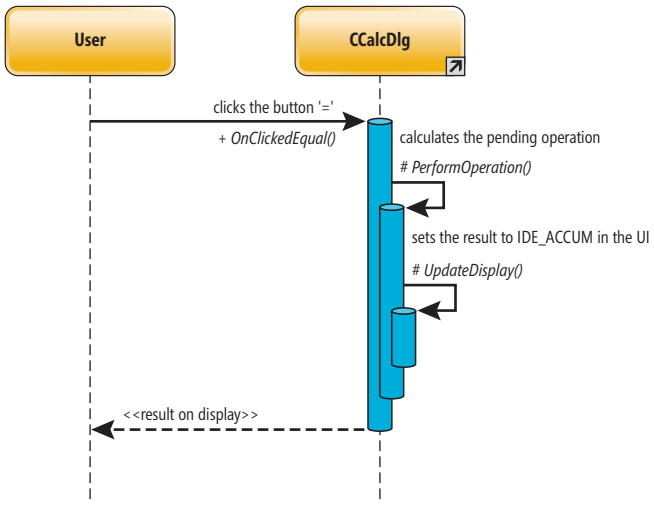


Figure 3 **Sequence Diagram for an Event—Clicking the Equal Sign Button**

(HTML, Qt, MFC, Cocoa, among others) that's suitable for the application device. It doesn't know how the domain logic is implemented; its function is just to display domain data structures or part of them. The Controller acts as a go-between by capturing user input to trigger actions in the domain, which causes the View to update to reflect a newer status.

MVC is widely known, but it's not the only way to isolate the UI from the domain. In the calculator scenario I'll rely on an MVC variation known as Presentation Model (bit.ly/1187Bk). I initially considered Model-View-ViewModel (MVVM), another variation that's popular among Microsoft .NET Framework developers. Presentation Model was a better fit for this scenario, though. Presentation Model is ideal when you want to implement a new user-interaction technology (Windows 8, in this case), but no changes are to be made in terms of the UI behavior. This pattern still considers a Model and a View as before, but the Controller role is played by an abstract representation of the View named the Presentation Model. This component implements the common behaviors of the View, including part of its status, without regard for the technology of the View.

Figure 5 depicts the modified version of the MFC application.

CalculatorPresentationModel keeps a reference to the View (modeled as the interface ICalculatorView), because once it's determined that the View status has changed, it invokes the function UpdateDisplay. In the MFC sample, the View is CCalcDlg itself because that's the class that deals directly with MFC.

CCalcDlg creates its Presentation Model in its constructor:

```
CCalcDlg::CCalcDlg(CWnd* pParent) : CDialog(CCalcDlg::IDD, pParent)
{
  presentationModel_ = unique_ptr<CalculatorPresentationModel>(
    new CalculatorPresentationModel(this));
  ...
}
```

> Smart pointers have the ability to release the object they reference when it's no longer required.

As you can see, I leveraged a C++11 smart pointer here called unique_ptr (see bit.ly/KswVGy for more information). Smart pointers have the ability to release the object they reference when it's no longer required. I used the smart pointer here to ensure that the Presentation Model is destroyed when the View lifecycle ends. The View keeps capturing window events, delegating to the Presentation Model with or without massaging the input, as shown in **Figure 6**.

Figure 4 **CCalcDlg**

```
// CCalcDlg.cpp
// Window messages trigger CCalcDlg function invocations
BEGIN_MESSAGE_MAP(CCalcDlg, CDialog)
  ON_WM_PAINT()
  ON_COMMAND_RANGE(IDB_0, IDB_9, OnClickedNumber)
  ON_BN_CLICKED(IDB_CLEAR, OnClickedClear)
  ON_BN_CLICKED(IDB_DIVIDE, OnClickedDivide)
  ON_BN_CLICKED(IDB_EQUAL, OnClickedEqual)
  ON_BN_CLICKED(IDB_MINUS, OnClickedMinus)
  ON_BN_CLICKED(IDB_PLUS, OnClickedPlus)
  ON_BN_CLICKED(IDB_TIMES, OnClickedTimes)
  ON_EN_SETFOCUS(IDE_ACCUM, OnSetFocusAccum)
END_MESSAGE_MAP()

...

// Event reaction
void CCalcDlg::OnClickedEqual() {
  PerformOperation();
  m_operator = OpNone;
}

// Domain logic
void CCalcDlg::PerformOperation() {
  if (m_bOperandAvail) {
    if (m_operator == OpNone)
      m_accum = m_operand;
    else if (m_operator == OpMultiply)
      m_accum *= m_operand;
    else if (m_operator == OpDivide) {
      if (m_operand == 0)
        m_errorState = ErrDivideByZero;
      else
        m_accum /= m_operand;
      }
    else if (m_operator == OpAdd)
      m_accum += m_operand;
    else if (m_operator == OpSubtract)
      m_accum -= m_operand;
  }

  m_bOperandAvail = FALSE;
  UpdateDisplay();
}

// Presentation logic
void CCalcDlg::UpdateDisplay() {
  CString str;
  if (m_errorState != ErrNone)
    str.LoadString(IDS_ERROR);
  else {
    long lVal = (m_bOperandAvail) ? m_operand : m_accum;
    str.Format(_T("%ld"), lVal);
  }
  GetDlgItem(IDE_ACCUM)->SetWindowText(str);
}
```

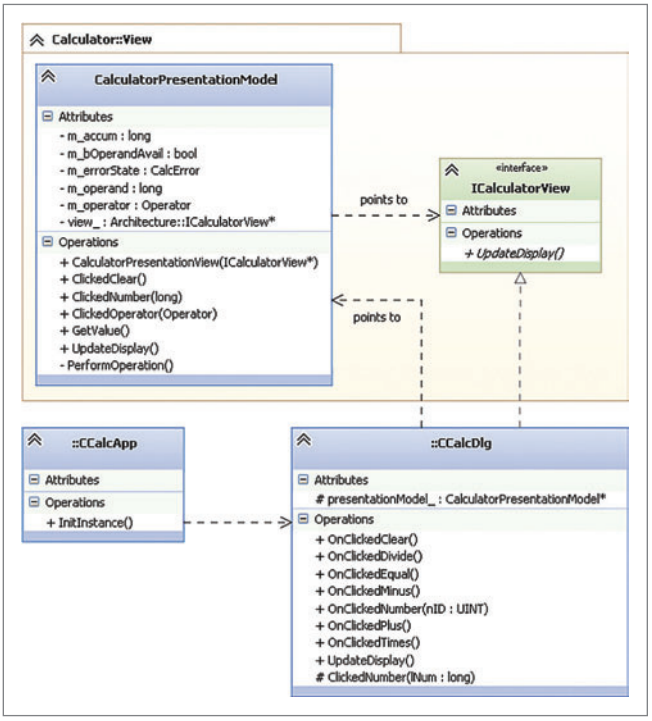


Figure 5 **The Namespace Calculator::View Consolidates the View Behavior in Its Associated Presentation Model**

You'll find this reworked version of the MFC sample in the folder mfccalc in the downloadable companion sample. You might notice there's no Model. In this example, the domain logic would've been a class containing functions for the four fundamental arithmetic operations, something like what's shown in **Figure 7**.

> The exception thrown by the Model is a last-resort measure when nothing else is available.

I decided to skip the Model in this small scenario, leaving its logic inside the Presentation Model. This will not be typical in most cases, and the Model logic will have to be implemented in its own class or set of classes. If you like, you can refactor the sample toward a purist approach as an exercise. If you do, you must pay special attention when implementing the function CalculatorModel::Divide because the Model is a reusable component that might be called by an automated process instead of some user interaction. In that case, it wouldn't matter that a division by zero throws an exception; at that point it would be an unexpected situation. But you don't have to remove the non-zero divisor-checking from the Presentation Model. Preventing erroneous data from forwarding to inner layers is always healthy. The exception thrown by the Model is a last-resort measure when nothing else is available.

Figure 6 **Some View Functions Showing Delegation**

```
// The parameter nID contains the ASCII code of a digit
void CCalcDlg::OnClickedNumber(UINT nID) {
  ASSERT(nID >= IDB_0 && nID <= IDB_9);
  presentationModel_->ClickedNumber(nID - IDB_0);
}

// Unchanged delegation
void CCalcDlg::OnClickedClear() {
  presentationModel_->ClickedClear();
}

enum Operator { OpNone, OpAdd, OpSubtract, OpMultiply, OpDivide };

// The Presentation Model contains a single method for all binary operations
void CCalcDlg::OnClickedDivide() {
  presentationModel_->ClickedOperator(OpDivide);
}

void CalculatorPresentationModel::ClickedOperator(Operator oper) {
  // PerformOperation is now in the PresentationModel;
  // it was in CCalcDlg (now being "the View")
  PerformOperation();
  m_operator = oper;
}

void CalculatorPresentationModel::PerformOperation() {
  if (m_errorState != ErrNone)
    return;

  if (m_bOperandAvail) {
    if (m_operator == OpNone)
      m_accum = m_operand;
    else if (m_operator == OpMultiply)
      m_accum *= m_operand;
  ... // Same as in Figure 4

  m_bOperandAvail = false;
  // This is an inline function defined just below
  UpdateDisplay();
}

// The UI refresh is deferred back to the actual View
inline void CalculatorPresentationModel::UpdateDisplay() {
  if (view_)
    view_->UpdateDisplay();
}
```

Go ahead and run the refactored solution mfccalc in the companion code, and notice that it still behaves as before, which is just what I wanted: to set up the code for a port without losing functionality. A serious refactoring process must consider a set of automated tests to confirm that the application behavior wasn't affected as a consequence. Native unit testing is available in Visual Studio 2012 and all its developer editions, including the free Express for Windows 8 (which doesn't come with MFC or other frameworks for desktop development, though).

Take a look at the solution Calculator\CalculatorTests in the companion code. The namespace Calculator::Testing contains a class PresentationModelTest whose methods test those of CalculatorPresentationModel, as shown in **Figure 8**.

By the way, unit testing is one of the most appreciated consequences of this Presentation Model pattern: The UI behavioral test automation is the same for the domain logic that, in this sample, I embedded in the Presentation Model. Thus, you can open new channels to your application (for example, Cocoa, Qt, wxWidgets or even non-native ones such as HTML5 or Windows Presentation Foundation) with the certainty that errors, if any, are happening in the new user-interaction components rather than the existing ones. You can leverage the code coverage feature to make sure that all lines of code are tested at least once.

Opening a XAML Façade to the Calculator Application

With the code refactored, I'm ready to create a new Windows UI for the MFC calculator. It's simply a matter of creating a new View for the Presentation Model pattern described earlier.

Windows 8 offers three technologies: XAML, HTML and DirectX.

- XAML is the XML-based markup language that lets you declare visual UI elements, data to bind to UI controls and handlers to call in response to events. These events are typically defined in so-called codebehind components that can be created in an extended C++ syntax known as C++ Component Extensions for the Windows Runtime (C++/CX), or created in other programming languages. I'll present a XAML-based calculator face.
- HTML lets you set which UI behaviors defined in JavaScript run in the Internet Explorer "Chakra" engine. It's possible—as I'll demonstrate in the next section—to invoke C++-based components from the JavaScript code.
- DirectX is ideal for multimedia-intensive apps. Because the calculator isn't a multimedia app, I won't discuss this here. DirectX apps can use XAML through interop, which you can read more about at bit.ly/NeUh04.

Figure 7 **A Pure Implementation That Includes a Calculator "Model"**

```
// Namespace Calculator::Model
class CalculatorModel {
public:
  long Add(const long op1, const long op2) const {
    return op1+op2;
  }

  long Subtract(const long op1, const long op2) const {
    return op1-op2;
  }

  long Multiply(const long op1, const long op2) const {
    return op1*op2;
  }

  long Divide(const long op1, const long op2) const {
    if (operand2)
      return operand1/operand2;
    else
      throw std::invalid_argument("Divisor can't be zero.");  }
};

// Namespace Calculator::View
class CalculatorPresentationModel {
public:
  ...
  void PerformOperation();
  ...
private:
  // The Presentation Model contains a reference to a Model
  unique_ptr<Model::CalculatorModel> model_;
  ...
}

void CalculatorPresentationModel::PerformOperation()
{
  if (m_errorState != ErrNone)
    return;

  // Same like before, but this time the PresentationModel asks
  // the model to execute domain activities instead of doing it itself
  if (m_bOperandAvail) {
    if (m_operator == OpNone)
      m_accum = m_operand;
    else if (m_operator == OpMultiply)
      m_accum = model_->Multiply(m_accum, m_operand);
    else if (m_operator == OpDivide) {
      if (m_operand == 0)
        m_errorState = ErrDivideByZero;
      else
        m_accum = model_->Divide(m_accum, m_operand);
    }
    else if (m_operator == OpAdd)
      m_accum = model_->Add(m_accum, m_operand);
    else if (m_operator == OpSubtract)
      m_accum = model_->Subtract(m_accum, m_operand);
  }

  m_bOperandAvail = false;
  UpdateDisplay();
}
```

To create a XAML view, I chose the basic Blank App from the list of Visual C++ Windows 8 templates and created a project called XamlCalc.

This blank template contains just an empty MainPage.xaml, which I'll fill with controls to make the Windows 8 counterpart of the former CCalcDlg control in the MFC version. This is all that's necessary to port the calculator application because it consists of a single window only, with no navigation. When porting your application, you might consider the other templates so you can offer a page-navigation mechanism that provides an intuitive and predictable UX. You'll find guidance about this on the "Designing UX for Apps" page (bit.ly/Izbxky).

> It's possible to invoke C++-based components from the JavaScript code.

The blank template also comes with an App.xaml file, similar in purpose to the class CCalcApp in the MFC version (see **Figure 2**). It's a bootstrap loader that initializes the rest of the components and passes control over to them. The function CCalcApp::InitInstance creates a CCalcDlg window, which then serves as a UI. (You'll find all of these in the XamlCalc solution in the downloadable companion code.) In the XAML case, App::OnLaunched is generated by default in the codebehind source file, App.xaml.cpp, and triggers an initial navigation to MainPage:

```
void App::OnLaunched(
  Windows::ApplicationModel::Activation::LaunchActivatedEventArgs^ pArgs) {
  ...
  // Create a Frame to act navigation context and navigate to the first page
  auto rootFrame = ref new Frame();
  if (!rootFrame->Navigate(TypeName(MainPage::typeid))) {
    throw ref new FailureException("Failed to create initial page");
  }
  ...
}
```

I use the Visual Studio built-in XAML editor to create an immersive calculator page by dragging controls from the toolbox and completing some manual editing, such as user-defined styles, data binding, associated events and so forth. The resulting XAML looks like the code in **Figure 9**. The calculator is shown in **Figure 10**.

I defined the style of the buttons (for both numbers and operators) in App.xaml, so I don't need to reiterate colors, alignments, fonts and other properties for each button. Similarly, I associated event handlers to the Click property of each button; the handlers

Figure 8 **Isolating Errors with Native Unit Testing**

```
// Namespace Calculator::Testing
TEST_CLASS(PresentationModelTest) {
private:
  CalculatorPresentationModel presentationModel_;
public:
  TEST_METHOD_INITIALIZE(TestInit) {
    presentationModel_.ClickedClear();
  }

  TEST_METHOD(TestDivide) {
    // 784 / 324 = 2 (integer division)
    presentationModel_.ClickedNumber(7);
    presentationModel_.ClickedNumber(8);
    presentationModel_.ClickedNumber(4);

    presentationModel_.ClickedOperator(OpDivide);

    presentationModel_.ClickedNumber(3);
    presentationModel_.ClickedNumber(2);
    presentationModel_.ClickedNumber(4);

    presentationModel_.ClickedOperator(OpNone);

    Assert::AreEqual<long>(2, presentationModel_.GetAccum(),
      L"Divide operation leads to wrong result.");
    Assert::AreEqual<CalcError>(ErrNone, presentationModel_.GetErrorState(),
      L"Divide operation ends with wrong error state.");
  }

  TEST_METHOD(TestDivideByZero) {
    // 784 / 0 => ErrDivideByZero
    presentationModel_.ClickedNumber(7);
    presentationModel_.ClickedNumber(8);
    presentationModel_.ClickedNumber(4);

    presentationModel_.ClickedOperator(OpDivide);

    presentationModel_.ClickedNumber(0);

    presentationModel_.ClickedOperator(OpNone);

    Assert::AreEqual<CalcError>(ErrDivideByZero, presentationModel_.GetErrorState(),
      L"Divide by zero doesn't end with error state.");
  }

  ... // More tests for the remaining calculator operations
};
```

Figure 9 **A XAML Version of the MFC Calculator**

```
<Page
  Loaded="Page_Loaded"
  x:Class="XamlCalc.MainPage" ...>

  <Grid Background="Maroon">
    ...
    <Border Grid.Row="1" Background="White" Margin="20,0">
      <TextBlock x:Name="display_" TextAlignment="Right" FontSize="90"
        Margin="0,0,20,0" Foreground="Maroon" HorizontalAlignment="Right"
        VerticalAlignment="Center"/>
    </Border>
    <Grid Grid.Row="2">
      ...
      <Button Grid.Column="0" Style="{StaticResource Number}"
        Click="Number_Click">7</Button>
      <Button Grid.Column="1" Style="{StaticResource Number}"
        Click="Number_Click">8</Button>
      <Button Grid.Column="2" Style="{StaticResource Number}"
        Click="Number_Click">9</Button>
      <Button Grid.Column="3" Style="{StaticResource Operator}"
        Click="Plus_Click">+</Button>
    </Grid>
    ...
    <Grid Grid.Row="5">
      ...
      <Button Grid.Column="0" Style="{StaticResource Number}"
        Click="Number_Click">0</Button>
      <Button Grid.Column="1" Style="{StaticResource Operator}"
        Click="Clear_Click">C</Button>
      <Button x:Name="button_equal_" Grid.Column="2"
        Style="{StaticResource Operator}" Click="Equal_Click"
        KeyUp="Key_Press">=</Button>
      <Button Grid.Column="3" Style="{StaticResource Operator}"
        Click="Divide_Click">/</Button>
    </Grid>
  </Grid>
</Page>
```

are MainPage methods defined in the codebehind source file MainPage.xaml.cpp. Here are a couple of examples, one for clicked numbers and one for the divide operation:

```
void MainPage::Number_Click(Platform::Object^ sender,
  Windows::UI::Xaml::RoutedEventArgs^ e)
{
  Button^ b = safe_cast<Button^>(sender);
  long nID = (safe_cast<String^>(b->Content)->Data())[0] - L'0';
  presentationModel_->ClickedNumber(nID);
}

void MainPage::Divide_Click(Platform::Object^ sender,
  Windows::UI::Xaml::RoutedEventArgs^ e)
{
  presentationModel_->ClickedOperator(OpDivide);
}
```

As you can see, these C++/CX methods in MainPage just take event information and give it to an instance of my standard C++ class, CalculatorPresentationModel, to perform the actual UI activity. This demonstrates that it's possible to take standard C++ logic from existing native applications and reuse it in a brand-new C++/CX Windows Store application. This reusability lowers the cost of maintaining both versions, as they can both leverage any update inside the common components—the CalculatorPresentationModel in this case.

Implementation-specific components can be called back by the common components as long as they implement well-defined abstract interfaces. In my example, for instance, CalculatorPresentationModel::UpdateDisplay delegates the actual job into an instance of ICalculatorView:

```
inline void CalculatorPresentationModel::UpdateDisplay(void) {
  if (view_)
    view_->UpdateDisplay();
}
```

In the MFC version, ICalculatorView is implemented by the MFC-based CCalcDlg class. Take a look at the refactored sequence diagram in **Figure 11** and compare it with the original in **Figure 3**.

To keep the XAML version analogous to the MFC case, I should've implemented ICalculatorView in MainPage. Instead, I had to implement ICalculatorView as a different class because MainPage is a C++/CX class and, therefore, can't derive from a standard C++ class. C++ and its projection into the Windows Runtime (C++/CX) have different type systems—which interoperate nicely in any case. Implementing a pure C++ ICalculatorView interface wasn't a big deal:

```
namespace XamlCalc {
  class CalcView : public ICalculatorView {
  public:
    CalcView() {}
    CalcView(MainPage^ page) : page_(page) {}
    inline void UpdateDisplay()
      { page_->UpdateDisplay(); }
  private:
    MainPage^ page_;
  };
}
```

Figure 10 **The Look and Feel of the XAML Calculator**

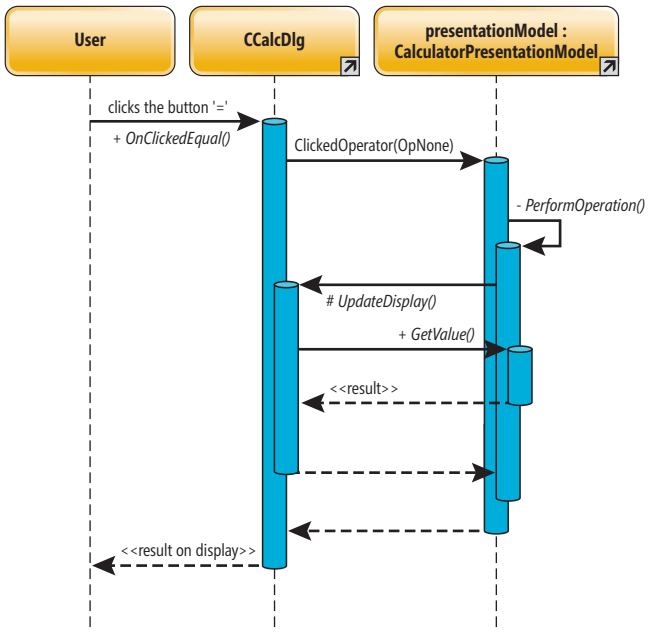


Figure 11 **The Sequence Diagram for the Equal Sign Button in the Decoupled MFC Version**

Standard C++ and C++/CX classes can't derive from each other, but they can still hold references to each other—in this case the private member page_, which is a reference to a C++/CX MainPage. To update the display control in the XAML version, I just change the Text property of the MainPage.xaml TextBlock control called display_:

```
void MainPage::UpdateDisplay() {
  display_->Text = (presentationModel_->GetErrorState() != ErrNone) ?
    L"ERROR!" :
    safe_cast<int64_t>(presentationModel_->GetValue()).ToString();
}
```

Figure 12 shows the XAML calculator class diagram.

Figure 13 shows the sequence corresponding to the action of pressing the equal sign button in the XAML version.

I should mention that the WRL heavily promotes asynchronous processing through all of its APIs for event handling. My code is 100 percent synchronous, though. I could've made it asynchronous by using the task-based Parallel Patterns Library, which implements tasks and continuations based on Windows 8 asynchrony concepts. This wasn't justified in my small example, but it's worth reading more about it at the "Asynchronous Programming in C++" page in the Windows Developer Center (bit.ly/Mi84D1).

Press F5 and run the application to see the XAML version in action. When migrating or creating your Windows Store applications, it's important that you design your application UI based on the new Windows Experience design patterns, described on the Microsoft Dev Center (bit.ly/0xo3S9). Follow the recommended patterns for commanding, touch, flipped orientation, charms and more in order to keep your application UX-intuitive for first-time users.

Another Example: A New Windows UI HTML Calculator

The XAML example is enough to get the initial MFC-based calculator sample running side by side with Windows 8. However, under certain circumstances (such as the expertise of your team or to leverage existing assets), you might consider HTML and JavaScript instead of XAML for the UI.

The Presentation Model design pattern described in this article is still useful, even if your UI contains logic in a non-C++ language like JavaScript. This miracle is possible because, in the Windows 8 environment, JavaScript projects to the Windows Runtime much as C++ does, making it possible for both to interoperate as they both share the type system established by the WRL.

In the companion code, you'll find a solution called HtmlCalc that contains a default.html page similar to MainPage.xaml. **Figure 14** shows a UI description comparable to the XAML version showed in **Figure 9**.

The codebehind role in HTML pages is played by JavaScript code. Indeed, you'll find such code in the file js\default.js. My Calculator-

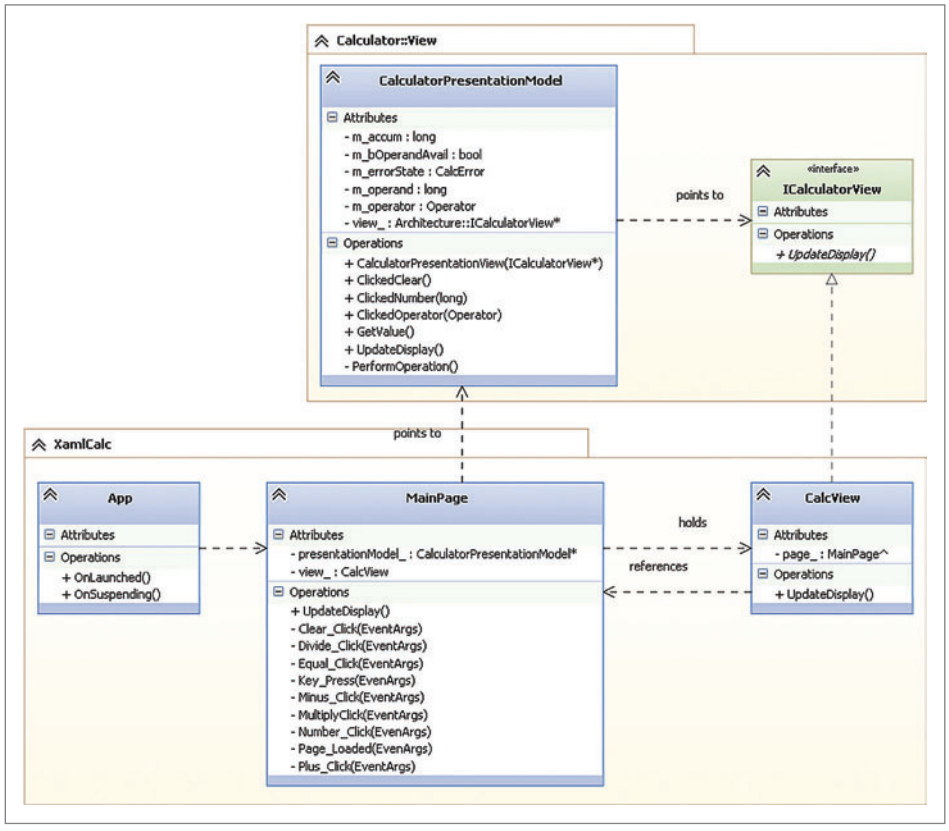


Figure 12 **The XAML-C++/CX Calculator Application Class Diagram**

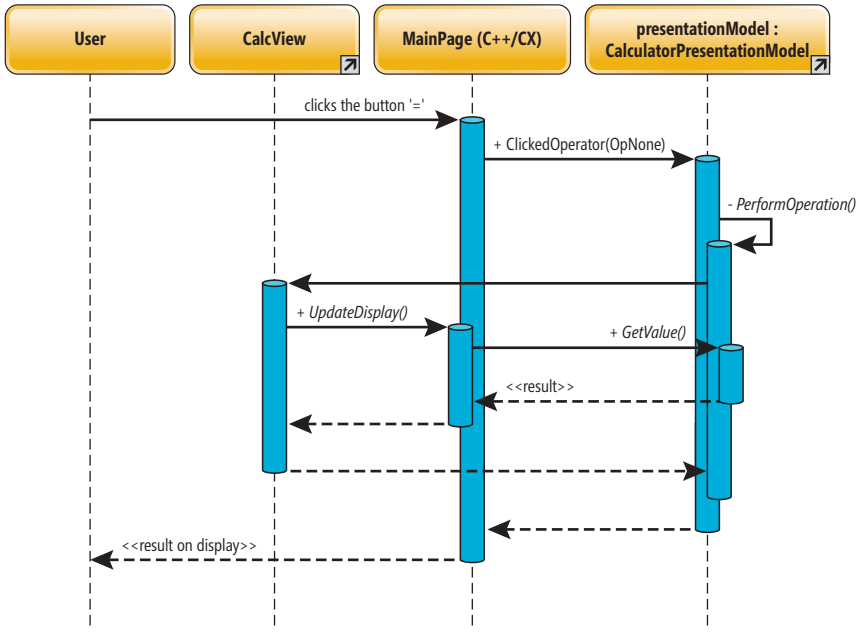


Figure 13 **The Sequence Diagram for the Equal Sign Button in the XAML version.**

PresentationModel, because it's a standard C++ class, can't be invoked directly from the JavaScript portion, but you can do it indirectly via a bridging C++/CX component—Calculator::View::CalcView.

Instancing this component from JavaScript is as easy as declaring the following in default.js:

```
// This is JavaScript code, instancing a C++/CX proxy to my PresentationModel
var nativeBridge = new Calculator.View.CalcView();
```

As an example of this approach, pressing the equal sign button triggers a call to the following JavaScript function:

```
function Equal_Click() {
    display_.textContent = nativeBridge.equal_click();
}
```

This propagates the call to CalcView::equal_click, which "talks" natively with my standard C++ CalculatorPresentationModel:

```
String^ CalcView::equal_click() {
  presentationModel_->ClickedOperator(OpNone);
  return get_display();
}

String^ CalcView::get_display() {
  return (presentationModel_->GetErrorState()
!= ErrNone) ?
    L"ERROR!" :
    safe_cast<int64_t>(presentationModel_-
>GetValue()).ToString();
}
```

In this particular scenario, the C++/CX component CalcView just forwards every request to the standard C++ PresentationModel (see the sequence diagram in **Figure 15**). We can't avoid it on our way to the reusable C++ component, though (see **Figure 15**).

Because the C++/CX proxy must be created manually, the associated cost shouldn't be ignored. Still, you can balance it against the benefit of reusing components, as I do in my scenario with CalculatorPresentationModel.

Go ahead and press F5 to see the HTML version in action. I've shown how to reuse existing C++ code to expand its reach to the novel framework in Windows 8, without dropping the original channels (MFC in my case). We're now ready for some final reflections.

Hello (Real) World!!

My porting scenario is a particular case, which may not be your particular case, which may not be someone else's particular case. Much of what I showed here is applicable to the MFC Calculator scenario, and I might make different decisions if I were porting a different application to the WRL. Consequently, here are some general conclusions about porting applications:

- Standard plain objects—those that have no specific relation with third-party APIs—have maximum reusability and, therefore, no- or low-cost portability. In contrast,

Figure 14 **The HTML Markup for the Calculator UI**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>HTMLCalc</title>
    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
    <script src="//Microsoft.WinJS.0.6/js/base.js"></script>
    <script src="//Microsoft.WinJS.0.6/js/ui.js"></script>
    <!-- HTMLCalc references -->
    <link href="/css/default.css" rel="stylesheet">
    <script src="/js/default.js"></script>
  </head>
  <body onkeypress="Key_Press()">
    <table border="0">
      <tr>
        <td class="Display" colspan="7" id="display_">0 </td>
      </tr>
      <tr>
        <td>
          <button class="Number" onclick="Number_Click(7)">7</button>
        </td>
        <td>
          <button class="Number" onclick="Number_Click(8)">8</button>
        </td>
        <td>
          <button class="Number" onclick="Number_Click(9)">9</button>
        </td>
        <td>
          <button class="Operator" onclick="Plus_Click()">+</button>
        </td>
      </tr>
      ...
      <tr>
        <td>
          <button class="Number" onclick="Number_Click(0)">0</button>
        </td>
        <td>
          <button class="Operator" onclick="Clear_Click()">C</button>
        </td>
        <td>
          <button class="Operator" onclick="Equal_Click()">=</button>
        </td>
        <td>
          <button class="Operator" onclick="Divide_Click()">/</button>
        </td>
      </tr>
    </table>
  </body>
</html>
```

reusability is constrained when objects have explicit ties to non-standard APIs, such as MFC, Qt and the WRL. For example, MFC is available only on the Windows desktop. Qt, on the other hand, is present in other environments, though not in all. In such cases, avoid mixtures that erode reusability by making application objects "talk" to abstract classes. Then derive from these classes to create third-party-aware implementations. Look at what I did with ICalculatorView (the abstract class) and its implementations CCalcDlg and XamlCalc::CalcView. For Windows 8 development, get familiar with the WRL APIs and which Win32 APIs they're replacing. You'll find more information at bit.ly/IBKphR.

- I applied the Presentation Model pattern because my goal was to mimic in the Windows Runtime what I already had on the desktop. You might decide to cut features if they don't make much sense—such as applying styles to text in a mobile e-mail app. Or you might add features that leverage your new target platform—think, for instance, about image stretching via multi-touch in an image-viewer application. In such cases, another design pattern might be more suitable.

  The Presentation Model I used is great for maintaining business continuity and for low maintenance costs. This lets me deliver Windows Store versions of the app without cutting ties with customers who prefer the original MFC option. Maintaining two channels (XAML and MFC or HTML and MFC) isn't twice the cost as long as I have reusable components like CalculatorPresentationModel.
- The overall reusability of an application is determined by the ratio of code lines that are common to all versions versus code lines for maintaining specific versions (components maintained by third parties aren't considered). There are cases where applications rely heavily on non-standard APIs (such as an augmented-reality app that leverages OpenGL and iOS sensors). The reusability ratio can be so low that you might eventually decide to port the application without component reusability other than the conceptual one.

The Presentation Model I used is great for maintaining business continuity and for low maintenance costs.

- Don't start asking who could have so poorly architected the existing application, making your porting job so difficult. Start refactoring it instead. Keep in mind that Agile methodologies aren't aimed at mature, robust, highly reusable architectures; what they emphasize is software delivery. Making software generic and extensible for future reusability and portability requires a lot of experience, as it's not easy to make design decisions in the dark.
- You might be porting your application to Windows 8, iOS or Android with the intention to sell it through the marketplaces of these platforms. In that case, keep in mind that your application must pass a certification process before being accepted (bit.ly/L0sY9i). This could force you to support UI behaviors you never contemplated in your original version (such as touch-first, charms and so forth). Failing to meet certain standards could result in your application being rejected. Don't overlook such "regulatory compliance" when estimating costs.

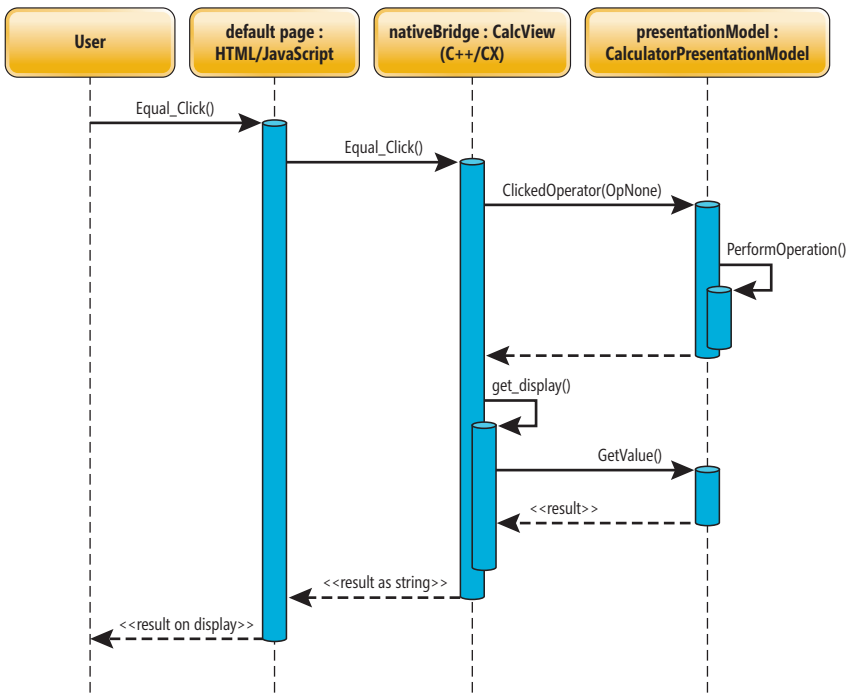


Figure 15 **The Sequence Diagram for the Equal Sign Button in the Hybrid HTML-C++ Version**

Meeting the Challenge

The new Windows Runtime and the still-ubiquitous Windows desktop pose a challenge to developers who don't want to pay the extra cost of maintaining a separate application per platform. Throughout this article, I demonstrated that existing codebases can be leveraged not only to enable new channels but also to improve through refactoring the quality of existing codebases. ■

DIEGO DAGUM *is a software architect and trainer with more than 20 years of experience in the industry. He can be reached at email@diegodagum.com.*

THANKS to the following technical experts for reviewing this article: *Marius Bancila, Angel Jesus Hernandez and the Windows 8 team*

Managing Memory in Windows Store Apps

David Tepper

Windows 8 is designed to feel fluid and alive, letting users rapidly switch among multiple apps to accomplish various tasks and activities. Users expect to quickly pop in and out of different experiences, and they never want to feel like they have to wait for an app when they need to use it. In this model, apps are rarely terminated by the user; instead they're frequently toggled between a state of execution and suspension. Apps are brought to the foreground for use and then moved to the background when the user switches to another app—and all the while users expect their machines not to slow down or feel sluggish, even as they open more and more apps.

In the Microsoft Windows Application Experience team investigations, we've seen that some Windows Store apps begin to encounter resource issues during prolonged use. Memory management bugs in apps can compound over time, leading to unnecessary memory usage and negatively impacting the machine overall. In our efforts to squash these bugs in our own products, we've identified a number of repeating problem patterns, as well as common fixes and techniques to escape them. In this article, I'll discuss how to think about memory management in your Windows Store apps as well as ways to identify potential memory leaks. I'll also provide some codified solutions to common issues the team has observed.

This article discusses:

- Different kinds of working memory
- Discovering memory leaks
- Common sources of memory leaks
- Architecting Windows Store apps to avoid leaks using JavaScript

Technologies discussed:

Windows 8, JavaScript

What Are Memory Leaks?

Any scenario in an app that leads to resources that can be neither reclaimed nor used is considered a memory leak. In other words, if the app is holding a chunk of memory that the rest of the system will never be able to use until the app is terminated, and the app itself is not using it, there's a problem. This is a broader definition than the typical explanation of a memory leak, "Dynamically allocated memory that's unreachable in code," but it's also more useful because it encompasses other, similar resource-utilization problems that can negatively affect both the user and the system. For example, if an app is storing data that's reachable from all parts of the code, but the data is used only once and never released afterward, it's a leak according to this definition.

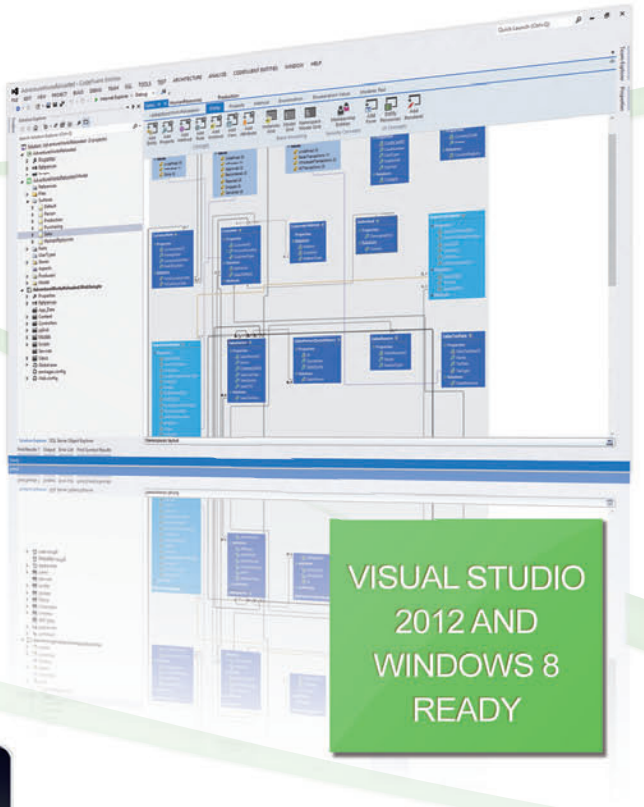
It's important to keep in mind that sometimes data is stored in memory that will never be used simply due to the user's actions in that particular instance. So long as this information is potentially useable throughout the lifetime of the app or is freed when it's no longer needed, it's *not* considered a leak, despite never being used.

CodeFluent Entities

LESS PLUMBING CODE, MORE FEATURES

CodeFluent Entities is a Visual Studio 2008/2010/2012 integrated environment that allows you to model your business entities, and generate consistent foundation code, continuously, across all chosen layers (database, business tier, services, user interface).

Using this model-first approach, your business logic is decoupled from the technology and your foundations will automatically benefit from upcoming innovation.



CodeFluent Entities is designed for the .NET platform and empowers users to streamline developments on major Microsoft platforms and technologies such as C#, VB.NET, SQL Server, Azure, ASP.NET, WPF, WCF, JSON/REST Services, SharePoint, Silverlight,Oracle Database, MySQL and PostgreSQL.

Your application deserves rock-solid foundations, let CodeFluent Entities generate them, and keep the fun part for you! Focus on what makes the difference.

DOWNLOAD YOUR FREE LICENSE

www.softfluent.com/landings_cfe_msdn

SoftFluent

TOOLS FOR DEVELOPERS, BY DEVELOPERS

SoftFluent is a software publisher providing solutions to help developers produce software code fluently, with users in more than 100 countries.

More information: *www.softfluent.com* - Contact: *info@softfluent.com*

What Is the Impact?

Gone are the days when machines were in a race to the sky for resource availability. PCs are getting smaller and more portable, with fewer available resources than their predecessors. This is fundamentally at odds with increasingly common usage patterns that involve switching among multiple experiences rapidly, with the expectation of a snappy UI and all content immediately available. Today, apps are multitudinous and alive for longer periods of time. At the same time, machines have less memory to support them all, and user expectations of performance have never been higher.

But does leaking a few megabytes really make that big a difference? Well, the issue isn't that a few megabytes leaked once, it's that memory leaks in code often compound over time as use of the app continues. If a scenario leads to unrecoverable resources, the amount of unrecoverable resources will grow, usually without bounds, as the user continues to repeat that scenario. This rapidly degrades the usability of the system as a whole as less memory is available for other processes, and it leads users to attribute poor system performance to your app. Memory leaks are most severe when they appear in:

- Frequent tasks (such as decoding the next frame of a video)
- Tasks that don't require user interaction to initiate (for example, auto-saving a document periodically)
- Scenarios that run for extended periods (such as background tasks)

Leaks in these situations (and in general) can dramatically increase the memory footprint of your app. Not only can this lead to a resource-utilization crisis for the entire system, it also makes your app much more likely to be terminated instead of suspended when not in use. Terminated apps take longer to reactivate than suspended apps, reducing the ease with which users can experience your scenarios. For full details on how Windows uses a process lifetime manager to reclaim memory from unused apps, see the Building Windows 8 blog post at bit.ly/JAqexg.

So, memory leaks are bad—but how do you find them? In the next few sections I'll go over where and how to look for these issues, and then take a look at why they occur and what you can do about them.

Different Kinds of Memory

Not all bits are allocated equally. Windows keeps track of different tallies, or views, of an app's memory use to make performance-analysis tasks easier. To better understand how to detect memory leaks, it's useful to know about these different memory classifications. (This section assumes some knowledge of OS memory management via paging.)

Private Working Set The set of pages your app is currently using to store its own unique data. When you think of "my app's memory usage," this is probably what you're thinking of.

Shared Working Set The set of pages your app is utilizing but not owned by your process. If your app is using a shared runtime or framework, common DLLs or other multi-process resource, those resources will take up some amount of memory. Shared working set is the measure of those shared resources.

Total Working Set (TWS) Sometimes simply called "working set," this is the sum of the private working set and the shared working set.

The TWS represents your app's full impact on the system, so the measurement techniques I'll describe will use this number. However, when tracking down potential issues, you may find it useful to investigate the private or shared working sets separately, as this can tell you whether it's your app that's leaking, or a resource that the app is using.

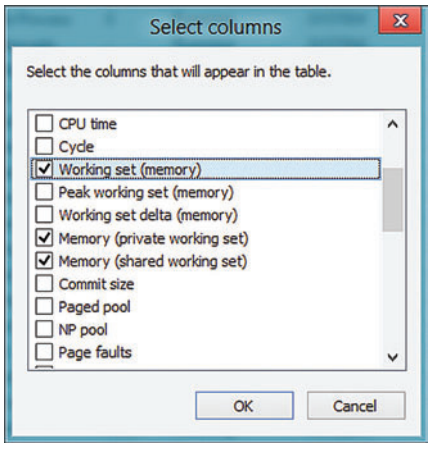


Figure 1 **Checking the Total Working Set in Windows Task Manager**

Discovering Memory Leaks

The easiest way to discover how much memory your app is using in each category is to use the built-in Windows Task Manager.

1. Launch the Task Manager by pressing Ctrl+Shift+Esc, and click on More Details near the bottom.
2. Click on the Options menu item and make sure that "Always on top" is checked. This prevents your app from going to the background and suspending while you're looking at Task Manager.
3. Launch your app. Once the app appears in Task Manager, right-click on it and click "Go to details."
4. Near the top, right-click on any column and go to "Select Columns."
5. You'll notice options here for shared and private working set (among others), but for the time being, just make sure that "Working set (memory)" is checked and click OK (see **Figure 1**).
6. The value you'll see is the TWS for your app.

To quickly discover potential memory leaks, leave your app and Task Manager open and write down your app's TWS. Now pick a scenario in your app that you want to test. A scenario consists of actions a typical user would execute often, usually involving no more than four steps (navigating between pages, performing a search and

Figure 2 **LeakyApp**

```
public sealed partial class ItemDetailPage : LeakyApp.Common.LayoutAwarePage
{
  public ItemDetailPage()
  {
    this.InitializeComponent();
  }

  Protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    Window.Current.SizeChanged += WindowSizeChanged;
  }
  private void WindowSizeChanged(object sender,
    Windows.UI.Core.WindowSizeChangedEventArgs e)
  {
    // Respond to size change
  }
  // Other code
}
```

so forth). Perform the scenario as a user would, and note any increase in the TWS. Then, without closing the app, go through the scenario again, starting from the beginning. Do this 10 times and record the TWS after each step. It's normal for the TWS to increase for the first few iterations and then plateau.

Did your app's memory usage increase each time the scenario was performed, without ever resetting to its original level? If so, it's possible you have a memory leak in that scenario and you'll want to take a look at the following suggestions. If not, great! But make sure to check other scenarios in your app, particularly those that are very common or that use large resources, such as images. Avoid performing this process on a virtual machine or over Remote Desktop, however; these environments can lead to false positives when looking for leaks and increase your memory usage numbers beyond their real value.

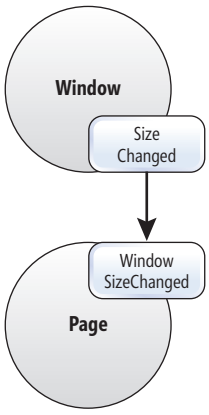


Figure 3 A Reference to the Event Handler

Using Pre-Windows 8 Memory-Leak Detection Tools

You might wonder if you can use existing memory-leak detection tools to identify issues with your Windows Store app. Unless these tools are updated to work with Windows 8, it's very likely they'll be "confused" by the app's lack of normal shutdown (which has been replaced by suspension). To get around this, you can use the AppObject "Exit" functionality to directly close the app in an orderly fashion, rather than forcefully closing it via external termination:

- C++—CoreApplication::Exit();
- C#—Application.Current.Exit();
- JavaScript—window.close();

When using this technique, make sure you don't ship your product with this code in place. Your app won't invoke any code that triggers on suspension and will need to be reactivated (instead of resumed) each time it's opened. This technique should be used only for debugging purposes and removed before you submit the app to the Windows Store.

Common Sources of Memory Leaks

In this section I'll discuss some common pitfalls we've seen developers run into across all kinds of apps and languages, as well as how to address these issues in your apps.

Event Handlers Event handlers are by far the most common sources of memory leaks we've seen in Windows Store apps. The fundamental issue is a lack of understanding about how event handlers work. Event handlers are not just code that gets executed; they are allocated data objects. They hold references to other things, and what they hold references to may not be obvious. Conceptually, the instantiation and registration of an event handler consists of three parts:

1. The source of the event
2. The event handler method (its implementation)
3. The object that hosts the method

As an example, let's look at an app called LeakyApp, shown in **Figure 2**.

The LeakyApp code shows the three parts of an event handler:

- Window.Current is the object that originates (fires) the event.
- An ItemDetailPage instance is the object that receives (sinks) the event.
- WindowSizeChanged is the event handler method in the ItemDetailPage instance.

After registering for the event notification, the current window object has a reference to the event handler in an ItemDetailPage object, as shown in **Figure 3**. This reference causes the ItemDetailPage object to remain alive as long as the current window object remains alive, or until the current window object drops the reference to the ItemDetailPage instance (ignoring, for now, other external references to these objects).

Note that, for the ItemDetailPage instance to operate properly, while it's alive the Windows Runtime (WinRT) transitively keeps all resources the instance is using alive. Should the instance contain references to large allocations such as arrays or images, these allocations will stay alive for the lifetime of the instance. In effect, registering an event handler extends the lifetime of the object instance containing the event handler, *and all of its dependencies*, to match the lifetime of the event source. Of course, so far, this isn't a resource leak. It's simply the consequence of subscribing to an event.

The ItemDetailPage is similar to all pages in an app. It's used when the user navigates to the page, but is no longer needed when they navigate to a different page. When the user navigates back to the ItemDetailPage, the application typically creates a new instance of the page and the new instance registers with the current window to receive SizeChanged events. The bug in this example, however, is that when the user navigates away from the ItemDetailPage, the page fails to unregister its event handler from the current window SizeChanged event. When the user navigates away from the ItemDetailPage, the current window still has a reference to the previous page and the current window continues to fire SizeChanged events to the page. When the user navigates back to the ItemDetailPage, this new instance also registers with the current window, as shown in **Figure 4**.

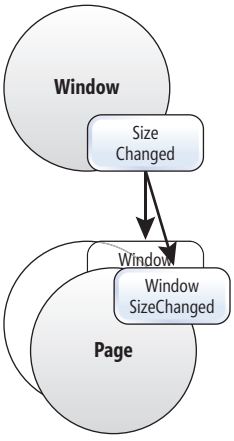


Figure 4 A Second Instance Registered with the Current Window

Five navigations later, five ItemDetailPage objects are registered with the current window (see **Figure 5**) and all their dependent resources are kept alive.

These no-longer-used ItemDetailPage instances are resources that can never be used or reclaimed; they are effectively leaked. If you take one thing away from this article, make sure it's that unregistering event handlers when they're no longer needed is the best way to prevent the most common memory leaks.

To fix the problem in LeakyApp, we need to remove the reference to the SizeChanged event handler from the current window. This can be done by

unsubscribing from the event handler when the page goes out of view, like so:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
  Window.Current.SizeChanged -= WindowSizeChanged;
}
```

After adding this override to the ItemDetailPage class, the ItemDetailPage instances no longer accumulate and the leak is fixed.

Note that this type of problem can occur with any object—any long-lived object keeps alive everything it references. I call out event handlers here because they are by far the most common source of this issue—but, as I'll discuss, cleaning up objects as they're no longer needed is the best way to avoid large memory leaks.

Circular References in Event Handlers that Cross GC Boundaries When creating a handler for a particular event, you start by specifying a function that will be called when the event is triggered, and then you attach that handler to an object that will receive the event in question. When the event actually fires, the handling function has a parameter that represents the object that initially received the event, known as the "event source." In the button click event handler that follows, the "sender" parameter is the event source:

```
private void Button_Click(object sender, RoutedEventArgs e)
{

}
```

By definition, the event source has a reference to the event handler or else the source couldn't fire the event. If you capture a reference to the source inside the event handler, the handler now has a reference back to the source and you've created a circular reference. Let's look at a fairly common pattern of this in action:

```
// gl is declared at a scope where it will be accessible to multiple methods
Geolocator gl = new Geolocator();

public void CreateLeak()
{
  // Handle the PositionChanged event with an inline function
  gl.PositionChanged += (sender, args) =>
    {
      // Referencing gl here creates a circular reference
      gl.DesiredAccuracy = PositionAccuracy.Default;
    };
}
```

In this example, gl and sender are the same. Referencing gl in the lambda function creates a circular reference because the source is referencing the handler and vice versa. Normally this kind of circular reference wouldn't be a problem because the CLR and JavaScript garbage collectors (GCs) are intelligent enough to handle such cases. However, issues can emerge when one side of the circular reference doesn't belong to a GC environment or belongs to a different GC environment.

Geolocator is a WinRT object. WinRT objects are implemented in C/C++ and therefore use a reference-counting system instead of a GC. When the CLR GC tries to clean up this circular reference, it can't clean up gl on its own. Similarly, the reference count for gl will never reach zero, so the C/C++ side of things won't get cleaned up either.

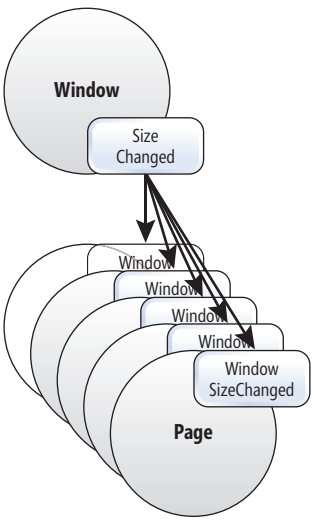


Figure 5 **Five Objects Registered with the Current Window**

Of course, this is a very simple example to demonstrate the issue. What if it wasn't a single object but instead a large grouping of UI elements such as a panel (or in JavaScript, a div)? The leak would encompass all of those objects and tracking down the source would be extremely difficult.

There are various mitigations in place so that many of these circularities can be detected and cleaned up by the GC. For example, circular references involving a WinRT event source that's in a cycle with JavaScript code (or circular references with a XAML object as the event source) are correctly reclaimed. However, not all forms of circularities are covered (such as a JavaScript event with a C# event handler), and as the number and complexity of references to the event source grow, the GC's special mitigations become less guaranteed.

> Unregistering event handlers when they're no longer needed is the best way to prevent the most common memory leaks.

If you need to create a reference to the event source, you can always explicitly unregister the event handler or null out the reference later to tear down the circularity and prevent any leaks (this goes back to reasoning about the lifetime of objects you create). But if the event handler never holds a reference to the source, you don't need to rely on platform-supplied mitigation or explicit code to prevent what can be a very large resource-utilization issue.

Using Unbounded Data Structures for Caching In many apps, it makes sense to store some information about the user's recent activities to improve the experience. For example, imagine a search app that displays the last five queries the user entered. One coding pattern to achieve this is to simply store each query in a list or other data structure and, when the time comes to give suggestions, retrieve the top five. The problem with this approach is that if the app is left open for long periods, the list will grow without bounds, eventually taking up a large amount of unnecessary memory.

Unfortunately, a GC (or any other memory manager) has no way to reason about very large, yet reachable, data structures that will never be used. To avoid the problem, keep a hard limit on the number of items you store in a cache. Phase out older data regularly and don't rely on your app being terminated to release these kinds of data structures. If the information being stored is particularly time-sensitive or easy to reconstitute, you might consider emptying the cache entirely when suspending. If not, save the cache to local state and release the in-memory resource; it can be reacquired on resume.

Figure 6 **Using Function Scope to Avoid Circular Closure References**

```
function getOnClick(paramA, paramB) {
  // This function's closure contains references to "paramA" and "paramB"
  return function () {
    paramA.doSomething();
    paramB.somethingElse();
  };
}

function addClickHandlerCorrectly(domObj, paramA, paramB, largeObject) {
  domObj.addEventListener(
    "click",
  // Because largeObject isn't passed to getOnClick, no closure reference
  // to it will be created and it won't be leaked
  getOnClick(paramA, paramB),
  false);
}
```

Avoid Holding Large References on Suspend

No matter the language, holding large references while suspended can lead to UX problems. Your app will stay suspended for as long as the system is able to service the requests of other running processes without needing additional memory that can only be retrieved by terminating apps. Because staying suspended means your app can be accessed more easily by the user, it's in your best interest to keep your memory footprint small during suspension.

A simple way to accomplish this is to simply free any references to large objects when suspending that can be reconstituted on resume. For example, if your app is holding an in-memory reference to local application data, releasing the reference may significantly lower your private working set, and it's easy to reacquire on resume because this data isn't going anywhere. (For more information on application data, see bit.ly/MDzzIr.)

To release a variable completely, assign the variable (and all references to the variable) to null. In C++, this will immediately reclaim the memory. For Microsoft .NET Framework and JavaScript apps, the GC will run when the app is suspended to reclaim the memory for these variables. This is a defense-in-depth approach to ensuring correct memory management.

Note, however, that if your app is written in JavaScript and has some .NET components, then the .NET GC won't be run on suspend.

Memory Management in JavaScript Windows Store Apps

Here are some tips for creating resource-efficient Windows Store apps in JavaScript. These are the recommended fixes for common issues we've seen in our own apps, and designing with them in mind will help stave off many potential issues before they cause headaches.

Use Code-Quality Tools An often-overlooked resource, freeware code-quality tools are available to all JavaScript developers on the Web. These tools inspect your code for lots of common issues, including memory leaks, and can be your best bet for catching issues early. Two useful tools are JSHint (jshint.com) and JSLint (jslint.com).

Use Strict Mode JavaScript has a "strict" mode that limits the way you can use variables in your code. These limitations present themselves as runtime errors that get thrown when the extra rules are violated. Such coding restrictions can help you avoid common memory leaks, such as implicitly declaring variables at global scope.

Figure 7 **A JavaScript Memory Leak**

```
function addOptionsChangedListener () {
  // A WinRT object
  var query = Windows.Storage.KnownFolders.picturesLibrary.createFileQuery();
  // 'data' is a JS object whose lifetime will be associated with the
  // behavior of the application. Imagine it is referenced by a DOM node, which
  // may be released at any point.
  // For this example, it just goes out of scope immediately,
  // simulating the problem.
  var data = {
    _query: query,
    big: new Array(1000).map(function (i) { return i; }),
    someFunction: function () {
      // Do something
    }
  };
  // An event on the WinRT object handled by a JavaScript callback,
  // which captures a reference to data.
  query.addEventListener("optionschanged", function () {
    if (data)
      data.someFunction();
  });

  // Other code ...
}
```

For more information on strict mode, its use and the imposed restrictions, check out the MSDN Library article, "Strict Mode (JavaScript)," at bit.ly/RrnjeU.

Avoid Circular Closure References JavaScript has a fairly complicated system of storing references to variables whenever a lambda (or inline) function is used. Basically, in order for the inline function to execute correctly when it's called, JavaScript stores the context of available variables in a set of references known as a closure. These variables are kept alive in memory until such time that the inline function itself is no longer referenced. Let's take a look at an example:

```
myClass.prototype.myMethod = function (paramA, paramB) {
  var that = this;
  // Some code
  var someObject = new someClass(
    // This inline function's closure contains references to the "that" variable,
    // as well as the "paramA" and "paramB" variables
    function foo() {
      that.somethingElse();
    }
  );
  // Some code: someObject is persisted elsewhere
}
```

After someObject is persisted, the memory referenced by "that," "paramA" and "paramB" won't be reclaimed until someObject is destroyed or releases its reference to the inline function it was passed in the someClass constructor.

Issues can arise with the closures of inline functions if the reference to the inline function isn't released, as the closure references will reside permanently in memory, causing a leak. The most common way this occurs is when a closure contains a circular reference to itself. This usually happens when an inline function references a variable that references the inline function:

```
function addClickHandler(domObj, paramA, paramB, largeObject) {
  domObj.addEventListener("click",
  // This inline function's closure refers to "domObj", "paramA",
  // "paramB", and "largeObject"
    function () {
      paramA.doSomething();
      paramB.somethingElse();
    },
  false);
}
```

In this example, domObj contains a reference to the inline function (through the event listener), and the inline function's closure contains a reference back to it. Because largeObject isn't being used, the intent is that it will go out of scope and get reclaimed; however, the closure reference keeps it and domObj alive in memory. This circular reference will result in a leak until domObj removes the event listener reference or gets nulled out and garbage collected. The proper way to accomplish something like this is to use a function that *returns* a function that performs your tasks, as shown in **Figure 6**.

With this solution, the closure reference to domObj is eliminated, but the references to paramA and paramB still exist, as they're necessary for the event handler implementation. To make sure to not leak paramA or paramB, you still need to either unregister the event listener or just wait for them to get automatically reclaimed when domObj gets garbage collected.

Revoke All URLs Created by URL.createObjectURL A common way to load media for an audio, video or img element is to use the URL.createObjectURL method to create a URL the element can use. When you use this method, it tells the system to keep an internal reference to your media. The system uses this internal reference to stream the object to the appropriate element. However, the system doesn't know when the data is no longer needed, so it keeps the internal reference alive in memory until it's explicitly told to release it. These internal references can consume large amounts of memory, and it's easy to accidently retain them unnecessarily. There are two ways to release these references:

1. You can revoke the URL explicitly by calling the URL.revokeObjectURL method and passing it the URL.
2. You can tell the system to automatically revoke the URL after it's used once by setting the oneTimeOnly property of URL.createObjectURL to true:

```
var url = URL.createObjectURL(blob, {oneTimeOnly: true});
```

Use Weak References for Temporary Objects Imagine you have a large object referenced by a Document Object Model (DOM) node that you need to use in various parts of your app. Now suppose that at any point the object can be released (for example, node.innerHTML = ""). How do you make sure to avoid holding references to the object so it can be fully reclaimed at any point? Thankfully, the Windows Runtime provides a solution to this problem, which allows you to store "weak" references to objects. A weak reference doesn't block the GC from cleaning up the object it refers to and, when dereferenced, it can return either the object or null. To better understand how this can be useful, take a look at the example in **Figure 7**.

Figure 8 **Using Weak References to Avoid a Memory Leak**

```
function addOptionsChangedListener() {
  var query = Windows.Storage.KnownFolders.picturesLibrary.createFileQuery();
  var data = {
    big: new Array(1000).map(function (i) { return i; }),
    someFunction: function () {
      // Do something
    }
  };
  msSetWeakWinRTProperty(query, "data", data)
  query.addEventListener("optionschanged", function (ev) {
    var data = msGetWeakWinRTProperty(ev.target, "data");
    if (data) data.someFunction();
  });
}
```

In this example, the data object isn't being reclaimed because it's being referenced by the event listener on query. Because the intent of the app was to clear the data object (and no further attempts to do so will be made), this is now a memory leak. To avoid this, the WeakWinRTProperty API group can be used with the following syntax:

```
msSetWeakWinRTProperty(WinRTObj, "objectName", objectToStore);
```

WinRTObj is any WinRT object that supports IWeakReference, objectName is the key to access the data and objectToStore is the data to be stored.

To retrieve the info, use:

```
var weakPropertyValue = msGetWeakWinRTProperty(WinRTObj, "objectName");
```

WinRTObj is the WinRT object where the property was stored and objectName is the key under which the data was stored.

The return value is null or the value originally stored (objectToStore).

Figure 8 shows one way to fix the leak in the addOptionsChangedListener function.

Because the reference to the data object is weak, when other references to it are removed, it will be garbage collected and its memory reclaimed.

Architecting Windows Store Apps Using JavaScript

Designing your application with resource utilization in mind can reduce the need for spot fixes and memory-management-specific coding practices by making your app more resistant to leaks from the start. It also enables you to build in safeguards that make it easy to identify leaks when they do happen. In this section I'll discuss two methods of architecting a Windows Store app written with JavaScript that can be used independently or together to create a resource-efficient app that's easy to maintain.

Dispose Architecture The Dispose architecture is a great way to stop memory leaks at their onset by having a consistent, easy and robust way to reclaim resources. The first step in designing your app with this pattern in mind is to ensure that each class or large object implements a function (typically named dispose) that reclaims memory associated with each object it references. The second step is to implement a broadly reachable function (also typically named dispose) that calls the dispose method on an object passed in as a parameter and then nulls out the object itself:

```
var dispose = function (obj) {
  /// <summary>Safe object dispose call.</summary>
  /// <param name="obj">Object to dispose.</param>
  if (obj && obj.dispose) {
    obj.dispose();
  }
  obj = null;
};
```

The goal is that the app takes on a tree-like structure, with each object having an internal dispose method that frees up its own resources by calling the dispose method on all objects it references, and so on. That way, to entirely release an object and all of its references, all you need to do is call dispose(obj)!

At each major scenario transition in your app, simply call dispose on all of the top-level objects that are no longer necessary. If you want to get fancy, you can have all of these top-level objects be part of one major "scenario" object. When switching among scenarios, you simply call dispose on the top-level scenario object and instantiate a new one for the scenario to which the app is switching.

Bloat Architecture The "Bloat" architecture allows you to more easily identify when memory leaks are occurring by making objects really large right before you release them. That way, if the object isn't actually released, the impact on your app's TWS will be obvious. Of course, this pattern should only be used during development. An app should never ship with this code in place, as spiking memory usage (even temporarily) can force a user's machine to terminate other suspended apps.

To artificially bloat an object, you can do something as simple as attaching a very large array to it. Using the join syntax quickly fills the entire array with some data, making any object it's attached to noticeably larger:

```
var bloatArray = [];
bloatArray.length = 50000;
itemToBloat.leakDetector = bloatArray.join("#");
```

To use this pattern effectively, you need a good way to identify when an object is supposed to be freed by the code. You can do this manually for each object you release, but there are two better ways. If you're using the Dispose architecture just discussed, simply add the bloat code in the dispose method for the object in question. That way, once dispose is called, you'll know whether the object truly had all of its references removed or not. The second approach is to use the JavaScript event DOMNodeRemoved for any elements that are on the DOM. Because this event fires *before* the node is removed, you can bloat the size of these objects and see if they're truly reclaimed.

Note that sometimes the GC will take some time to actually reclaim unused memory. When testing a scenario for leaks, if the app appears to have grown very rapidly, wait a while to confirm a leak; the GC may not have done a pass yet. If, after waiting, the TWS is still high, try the scenario again. If the app's TWS is still large, it's extremely likely there's a leak. You can hone in on the source by systematically removing this bloat code from the objects in your app.

Going Forward

I hope I've given you a strong foundation for identifying, diagnosing and repairing memory leaks in your Windows Store apps. Leaks often result from misunderstandings of how data allocation and reclamation occur. Knowledge of these nuances—combined with easy tricks such as explicitly nulling out references to large variables—will go a long way toward ensuring efficient apps that don't slow down users' machines, even over days of use. If you're looking for more information you can check out an MSDN Library article by the Internet Explorer team that covers related topics, "Understanding and Solving Internet Explorer Leak Patterns," at bit.ly/Rrta3P. ■

DAVID TEPPER *is a program manager on the Windows Application Experience team. He has been working on application model design and application deployment since 2008, primarily focusing on the performance of Windows Store apps and how those apps can extend Windows to provide deeply integrated functionality.*

THANKS to the following technical experts for reviewing this article: *Jerry Dunietz, Mike Hillberg, Mathias Jourdain, Kamen Moutafov, Brent Rector and Chipalo Street*

Data Binding in a Windows Store App with JavaScript

Chris Sells and Brandon Satrom

In this article we'll explore data binding in a Windows Store app built with JavaScript. This is a hybrid of other kinds of apps built for Windows 8. It's like a desktop app in that it's installed on your computer, unlike a Web site. On the other hand, it's like a Web site in that you can build it using HTML5, JavaScript and CSS3. However, instead of generating the UI on the server side, the JavaScript framework for building Windows Store apps and the underlying Windows Runtime allows you to build apps with client-side state, offline storage, controls, templates and binding—along with a whole host of other services.

This article discusses:

- The basics of data binding
- Binding objects
- Using initializers
- Creating a binding list
- Sorting and filtering data
- Grouping data
- Using templates

Technologies discussed:

JavaScript, HTML, Windows Runtime, Windows 8

Code download available at:

archive.msdn.microsoft.com/mag201210Binding

Data binding is the ability to take the value of a list of data, such as a set of RSS feeds, and use it to populate a control, such as a ListView. We can tailor which data we extract using templates. We'll show how binding is even more flexible than that, allowing updating of the UI as the underlying data changes, along with sorting, filtering and grouping.

When we use data binding, it's often in the context of a control. We're going to take a look at the ListView and its support for data binding and templates.

Data Binding 101

The use of binding allows you to set up an association between two properties on two objects, most often associating the property of an object with a property on an HTML Document Object Model (DOM) element, as **Figure 1** shows.

The purpose of establishing a binding is to copy data between the two objects involved: the source from where the data comes and the destination to which the data goes. You might think that you can accomplish this goal using a simple assignment statement:

```
myDestElem.value = mySourceObj.name;
```

Fundamentally, assignment is what binding does. However, binding is not just a description of what data to copy and to where, but also *when*. The "when" of a binding is generally one of the following:

- **One-way binding:** Copy the data to the DOM element when the object changes. This is the default in the Windows Library for JavaScript (WinJS).

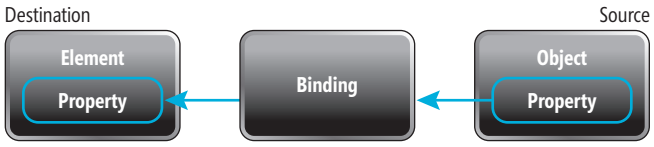


Figure 1 **Binding Between an Attribute on a Destination Element and a Property from a Source Object**

- **One-time binding:** Copy the data to the DOM element when the binding is first established. This is the equivalent of assignment.
- **Two-way binding:** Copy the data to the DOM element when the object changes and copy the data to the object when the DOM element changes. This isn't supported in WinJS.

By default, WinJS binding is one-way binding, although one-time binding is supported as well. Although WinJS doesn't support two-way binding, there's a nice place to hook in your favorite two-way binding engine, such as the one in jQuery, as you'll see.

Binding Objects

To get started, let's say we want to build a little browser for the people in our lives, as shown in **Figure 2**.

The idea is that we can have a number of people through whom we can navigate, each with a name, age and a favorite color. As we use the previous and next buttons, we navigate to other people in the list, and if we press the button in the middle—the clock—we celebrate a birthday by increasing the age of the currently shown person. The following represents the set of people we'll be browsing in our sample:

```
var people = [
  { name: "John", age: 18, favoriteColor: "red" },
  { name: "Tom", age: 16, favoriteColor: "green" },
  { name: "Chris", age: 42, favoriteColor: "blue" },
];
```

Starting from a Navigation Application project template in Visual Studio 2012 provides HTML for us to populate, as shown in **Figure 3**.

The interesting part of this HTML is the use of the data-win-bind attribute, which uses the following format:

```
<div data-win-bind="destProp1: sourceProp1; destProp2: sourceProp2;..."></div>
```

The data-win-bind syntax is a semicolon-delimited list of binding expressions. Each expression is the combination of a destination DOM element property and a source object property. The dotted syntax we're using to set the background color on the style for the favorite color div—that is, style.backgroundColor—works for properties on both the source and the destination, and it drills into subobjects, as you'd expect.

Each destination property in a binding expression is resolved against the HTML element that contains the data-win-bind attribute. The question is: Where does the object come from against which the property names are resolved? And the answer is: the data context.

Setting the data context is part of the binding operation, which establishes the association between the HTML element and whatever object is used as the data context, as shown in **Figure 4**.

The processAll function in the WinJS.Binding namespace is the function that parses the data-win-bind attributes for a given hierarchy of HTML elements, the section containing the input

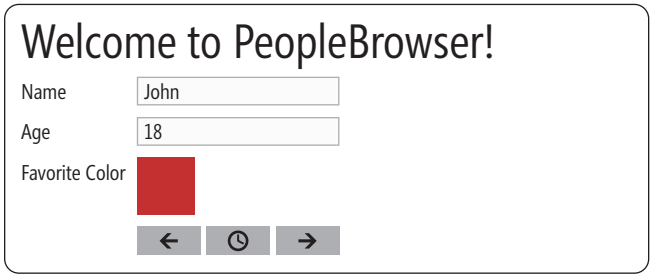


Figure 2 **Binding an Object to a Set of HTML Elements**

and div elements in our sample. The call to processAll establishes the bindings between the HTML elements and the data context as described in each data-win-bind attribute. The data context is the second argument to processAll and is used to resolve the property names found in each binding expression. The results already look like what we're after, as shown in **Figure 2**.

By default, if we do nothing else, we've established a one-way binding connection between the data context object and the DOM element. That means that as properties on the data source object change, we expect the output to change as well, as shown in **Figure 5**.

Figure 3 **The Main Form**

```
<!DOCTYPE html>
<!-- homePage.html -->
<html>
<head>
  <meta charset="utf-8" />
  <title>homePage</title>

  <!-- WinJS references -->
  <link href="//Microsoft.WinJS.1.0/css/ui-light.css" rel="stylesheet" />
  <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
  <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

  <link href="/css/default.css" rel="stylesheet" />
  <link href="/pages/home/home.css" rel="stylesheet" />
  <script src="/pages/home/home.js"></script>
</head>
<body>
  <!-- The content that will be loaded and displayed. -->
  <div class="fragment homepage">
    <header aria-label="Header content" role="banner">
      <button class="win-backbutton" aria-label="Back"
        disabled type="button"></button>
      <h1 class="titlearea win-type-ellipsis">
        <span class="pagetitle">Welcome to PeopleBrowser!</span>
      </h1>
    </header>
    <section aria-label="Main content" role="main">
      <!-- display each person -->
      <div id="nameLabel">Name</div>
      <input id="name" readonly="true" type="text" data-win-bind="value: name" />

      <div id="ageLabel">Age</div>
      <input id="age" readonly="true" type="text" data-win-bind="value: age" />

      <div id="colorLabel">Favorite Color</div>
      <div id="color" data-win-bind="style.backgroundColor:
        favoriteColor"></div>

      <div id="buttons">
        <button id="previousButton"></button>
        <button id="birthdayButton"></button>
        <button id="nextButton"></button>
      </div>
    </section>
  </div>
</body>
</html>
```

Figure 4 **Setting the Data Context**

```
// homePage.js
(function () {
  "use strict";

  WinJS.UI.Pages.define("/pages/home/home.html", {
    ready: function (element, options) {
      var people = [
        { name: "John", age: 18, favoriteColor: "red" },
        { name: "Tom", age: 16, favoriteColor: "green" },
        { name: "Chris", age: 42, favoriteColor: "blue" },
      ];

      // Bind the current person to the HTML elements in the section
      var section = element.querySelector("section[role=main]");
      var current = 0;
      WinJS.Binding.processAll(section, people[current]);
    }
  });
})();
```

Our implementation of the click handler for the birthday button looks up the current element and changes a property, specifically the age. However, we're not quite where we need to be to get the UI to update automatically.

The problem is that a standard JavaScript object doesn't support any notification protocol to inform interested parties—such as a binding—that its data has changed. To add the notification protocol is a matter of calling the as method from the WinJS.Binding namespace, as shown in **Figure 6**.

Figure 5 **As Properties on the Data Source Change, So Does the Output**

```
function ready(element, options) {
  var people = [
    { name: "John", age: 18, favoriteColor: "red" },
    { name: "Tom", age: 16, favoriteColor: "green" },
    { name: "Chris", age: 42, favoriteColor: "blue" },
  ];

  var section = element.querySelector("section[role=main]");
  var current = 0;
  WinJS.Binding.processAll(section, people[current]);

  birthdayButton.onclick = function () {
    var person = people[current];
    person.age++; // Changing a bound property doesn't work yet ...
  };
}
```

Figure 6 **Adding the Notification Protocol**

```
WinJS.UI.Pages.define("/pages/home/home.html", {
  ready: function (element, options) {
    var people = [
      // Notify binding listeners when these objects change
      WinJS.Binding.as({ name: "John", age: 18, favoriteColor: "red" }),
      WinJS.Binding.as({ name: "Tom", age: 16, favoriteColor: "green" }),
      WinJS.Binding.as({ name: "Chris", age: 42, favoriteColor: "blue" }),
    ];

    // Bind the current person to the HTML elements in the section
    var section = element.querySelector("section[role=main]");
    var current = 0;
    WinJS.Binding.processAll(section, people[current]);

    birthdayButton.onclick = function () {
      var person = people[current];
      person.age++; // Now this works!
    };
  }
});
```

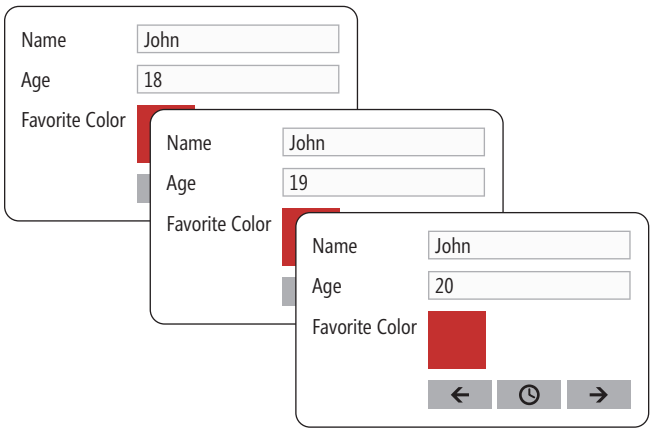


Figure 7 **Changing the Underlying Data Automatically Updates the Bound HTML Elements**

The "as" method wraps each object, providing the binding notification protocol so each property change notifies any binding listeners (such as our bound HTML elements). With this code in place, changing a person's age is reflected in the UI, as **Figure 7** shows.

We've been binding a single person object to the set of HTML elements for display, but you can process the data-binding expressions in the data-win-bind attributes with a different kind of data context, as shown in **Figure 8**.

Here's where we make a shift between passing in a single piece of data from our app's "model" (that is, the set of data that has nothing to do with the display of itself) to grouping the data that's appropriate for our "view" of that model (in this case, the current person to view). We call this variable the "view model" after a famous and useful technique for implementing UIs called Model-View-ViewModel

Figure 8 **Processing the Data-Binding Expressions**

```
WinJS.UI.Pages.define("/pages/home/home.html", {
  ready: function (element, options) {
    var people = [
      // Notify binding listeners when these objects change
      WinJS.Binding.as({ name: "John", age: 18, favoriteColor: "red" }),
      WinJS.Binding.as({ name: "Tom", age: 16, favoriteColor: "green" }),
      WinJS.Binding.as({ name: "Chris", age: 42, favoriteColor: "blue" }),
    ];

    // Bind the current person to the HTML elements in the section
    var section = element.querySelector("section[role=main]");
    var current = 0;
    var viewModel = WinJS.Binding.as({ person: people[current] });
    WinJS.Binding.processAll(section, viewModel);

    birthdayButton.onclick = function () {
      viewModel.person.age++;
    };

    // Bind to the previous object
    previousButton.onclick = function () {
      current = (people.length + current - 1) % people.length;
      viewModel.person = people[current];
    };

    // Bind to the next object
    nextButton.onclick = function () {
      current = (people.length + current + 1) % people.length;
      viewModel.person = people[current];
    };
  }
});
```

Figure 9 **Updating HTML to Use the View Model**

```
<section aria-label="Main content" role="main">
  <!-- display each person -->
  <div id="nameLabel">Name</div>
  <input id="name" readonly="true" type="text" data-win-bind="value: person.name" />
  <div id="ageLabel">Age</div>
  <input id="age" readonly="true" type="text" data-win-bind="value: person.age" />
  <div id="colorLabel">Favorite Color</div>
  <div id="color" data-win-bind="style.backgroundColor: person.favoriteColor"></div>
  <div id="buttons">
    <button id="previousButton"></button>
    <button id="birthdayButton"></button>
    <button id="nextButton"></button>
  </div>
</section>
```

(MVVM). Once we've built our view model as a bindable object, as the underlying properties change (the person, in our case) the view is updated. We then bind our view to the view model so that in our Next and Back button handlers—as we change which person we'd like to view—the view is notified. For this to work, we'll need to update the HTML to use the view model instead of the person directly, as shown in **Figure 9**.

Figure 10 shows the result.

In addition to binding objects to elements, binding also allows you to simply listen for a value to change. For example, right now as we change the index to the current value when the user clicks the *previous* or *next* buttons, we have to remember to write the code to change the view model's person field to match. However, you could use binding itself to help here, as shown in **Figure 11**.

Instead of a simple variable to hold the index to the currently shown person, we add the index to the currently viewed person to our bindable view model. All bindable objects have a bind method, which allows us to listen for changes to the object's properties (the current property in this example). When the user clicks on the *previous* or *next* buttons, we simply change the index of the current person to be shown and let the bind handler process the data-win-bind attributes for the HTML that shows the current person.

If you want to stop listening for bind events, you can call the unbind method.

Now, as we mentioned, everything you've seen works against the default: one-way binding. However, if you'd like to set up other

Figure 11 **Reprocessing Binding Expressions**

```
WinJS.UI.Pages.define("/pages/home/home.html", {
  ready: function (element, options) {
    var people = [
      // Notify binding listeners when these objects change
      WinJS.Binding.as({ name: "John", age: 18, favoriteColor: "red" }),
      WinJS.Binding.as({ name: "Tom", age: 16, favoriteColor: "green" }),
      WinJS.Binding.as({ name: "Chris", age: 42, favoriteColor: "blue" }),
    ];

    // Bind the current person to the HTML elements in the section
    var section = element.querySelector("section[role=main]");

    // Listen for the current index to change and update the HTML
    var viewModel = WinJS.Binding.as({ current: 0, person: null });
    WinJS.Binding.processAll(section, viewModel);
    viewModel.bind("current", function (newValue, oldValue) {
      viewModel.person = people[newValue];
    });

    birthdayButton.onclick = function () {
      viewModel.person.age++;
    };

    // Bind to the previous object
    previousButton.onclick = function () {
      // Set the current index and let the binding do the work
      viewModel.current = (people.length + viewModel.current - 1) % people.length;
    };

    // Bind to the next object
    nextButton.onclick = function () {
      // Set the current index and let the binding do the work
      viewModel.current = (people.length + viewModel.current + 1) % people.length;
    };
  }
});
```

kinds of binding or participate in the binding process itself, you can do so with a binding initializer.

Initializers

An initializer is the optional third parameter to a binding expression that specifies a function to call when the binding is established:

```
<div data-win-bind="destProp: sourceProp init" ...></div>
```

You provide an initializer as part of a binding expression if you want to participate in or even replace the existing binding behavior—for example, to perform a data conversion or even hook up jQuery two-way binding. A binding initializer is a function that

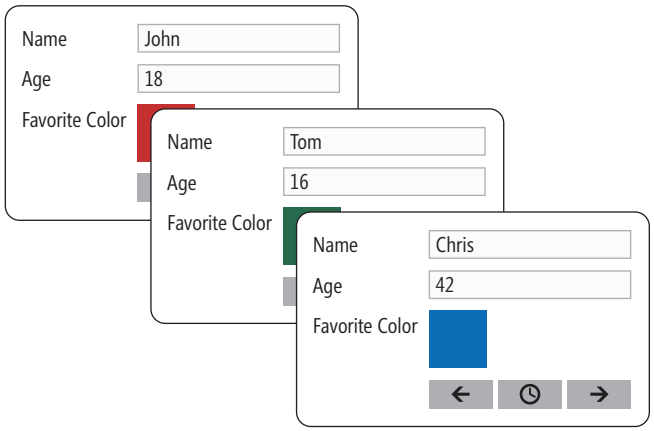


Figure 10 **You Can Rebind Different Objects to the Same Set of HTML Elements**

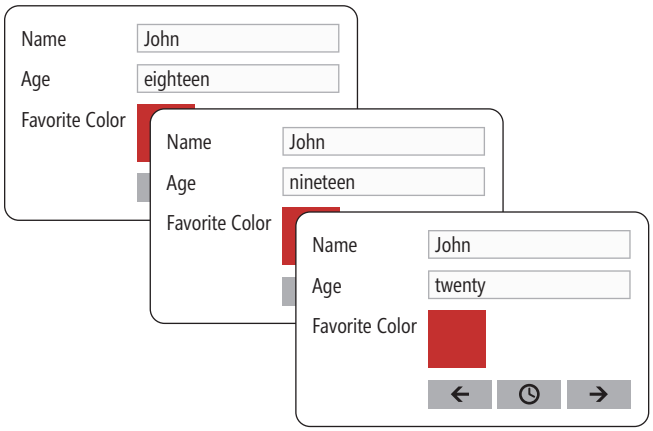


Figure 12 **Translating a Numeric Age into Words via Data Conversion**

takes over for the default one-way binding and has the following signature:

```
function myInit(source, sourceProperties,
  dest, destProperties) {...}
```

The details of implementing a custom initializer are beyond the scope of this article, but WinJS has several initializers built-in of which you can take advantage. For example, if you'd like to do one-time instead of one-way binding, you can use the oneTime function from the WinJS.Binding namespace:

```
<input data-win-bind=
  "value: person.name WinJS.Binding.oneTime"></input>
```

On the other end of the spectrum, when performing one-way binding, it's often useful to perform data conversion (also known as data transformation). For example, imagine we wanted to spell out the ages of the people in our life:

```
function getWordsFromNumber(i) {...}

// Convert ages to words
window.ageToWords =
  WinJS.Binding.converter(function (value) { return getWordsFromNumber(value); });
```

The converter function on the WinJS.Binding namespace provides the hard part of the initializer implementation; all you have to do is provide a function to perform the actual conversion and it will be called as the source value changes. Using it in the HTML looks like you'd expect:

```
<input data-win-bind="value: person.age ageToWords"></input>
```

You can see the results in **Figure 12**.

Binding values one at a time is useful, but even more useful—especially for Windows Store apps—is binding collections of values all at once. For that we have the binding list.

A Binding List

Imagine a simple example of using a ListView with a collection of values:

```
<div data-win-control="WinJS.UI.ListView"
  data-win-options="{itemDataSource: items.dataSource}">
</div>
```

Here we're creating a ListView control declaratively in HTML that's bound to the dataSource property of a global items object. In our JavaScript, it's easy to create and populate the items object as a binding list:

```
// A global binding list of items
window.items = new WinJS.Binding.List();

[0, 1, 2].forEach(function (i) {
  WinJS.Promise.timeout(500 * (i+1)).done(function () {
    // Add an item to the binding list, updating the ListView
    items.push(i);
  });
});
```

In this code we create a binding list (an instance of the List type from the WinJS.Binding namespace) and we loop over an array of values, using each to create a timeout, which is a promise that will fire at some time in the future according to the argument you pass (milliseconds). When each timeout completes, we'll see the ListView get an additional value, as **Figure 13** shows.

0	0	0
	1	1
		2

Figure 13 **The ListView Updating as the Underlying Binding List Updates**

```
Sorted by name (asc)
{"name":"Chris","age":42,"favoriteColor":"blue"}
{"name":"John","age":18,"favoriteColor":"red"}
{"name":"Tom","age":16,"favoriteColor":"green"}

Filtered by age (adults)
{"name":"John","age":18,"favoriteColor":"red"}
{"name":"Chris","age":42,"favoriteColor":"blue"}
```

Figure 14 **Filtering a Binding List**

The HTML doesn't provide an ID for the ListView, nor does the JavaScript interact with the ListView in any way. Instead, the HTML and the JavaScript both rendezvous on the items binding list. It's the binding list that implements the collection binding API (called IListDataSource). Instead of making you program against IListDataSource directly—it's not particularly programmer-friendly—the binding list exposes its IListDataSource implementation via the dataSource property.

To provide an API that's as familiar as possible to JavaScript programmers, the binding list implements a similar API and similar semantics as the built-in array supports—for example, push, pop, slice, splice and so on. As a comparison, here's how the built-in JavaScript array works:

```
// Using the built-in array
var array = [];
array.push("one");
array.push("two");
array.push("three");

var x = array[0]; // x = "one"
var y = array[1]; // y = "two"

array[2] = "THREE";
var z = array[2]; // z = "THREE";
```

The binding list behaves similarly except that instead of using an indexer—that is, square brackets—it exposes items via setAt and getAt:

```
// Using the WinJS binding list
var list = new WinJS.Binding.List();
list.push("one");
list.push("two");
list.push("three");

var x = list.getAt(0); // x = "one"
var y = list.getAt(1); // y = "two"

list.setAt(2, "THREE");
var z = list.getAt(2); // z = "THREE";
```

Sorting and Filtering

In addition to the familiar API, the binding list was built with Windows Store app building needs in mind. You've already seen how the dataSource property lets you connect to the ListView control. In addition, you might be interested in sorting and filtering your data. For example, here's our list of people again, this time wrapped in a binding list:

```
Sorted by name (asc)
{"name":"Chris","age":42,"favoriteColor":"blue"}
{"name":"John","age":18,"favoriteColor":"red"}
{"name":"Pete","age":17,"favoriteColor":"black"}
{"name":"Tom","age":16,"favoriteColor":"green"}
```

Figure 15 **Sorting a Binding List**

```
Filtered by name (minors) and
Sorted by name (asc)
{"name":"Pete","age":16.5,"favoriteColor":"black"}
{"name":"Tom","age":16,"favoriteColor":"green"}
```

Figure 16 **Sorting and Filtering a Binding List**

```
Grouped by age
{"name":"John","age":18,"favoriteColor":"red"}
{"name":"Chris","age":42,"favoriteColor":"blue"}
{"name":"Tom","age":16,"favoriteColor":"green"}
{"name":"Pete","age":17,"favoriteColor":"black"}
```

Figure 17 **Grouping a Binding List**

Grouped by age

adult	minor
{"name":"John","age":18,"favoriteColor":"red"}	{"name":"Tom","age":16,"favoriteColor":"green"}
{"name":"Chris","age":42,"favoriteColor":"blue"}	{"name":"Pete","age":17,"favoriteColor":"black"}

Figure 18 **Grouping a Binding List with ListView Headers**

```
var people = new WinJS.Binding.List([
  { name: "Tom", age: 16 },
  { name: "John", age: 17 },
  { name: "Chris", age: 42 },
]);

// Sort by name
window.sortedPeople = people.
  createSorted(function (lhs, rhs) { return lhs.name.localeCompare(rhs.name); });

// Filter by age (adults only)
window.filteredPeople = people.
  createFiltered(function (p) { return p.age > 17; });
```

We can sort a binding list by calling the createSorted method, passing in a sorting function, or filter it by calling the createFiltered method, passing in a function for filtering. The result of calling one of these create methods is a view over the data that looks and acts like another instance of the binding list, and we can bind it to a ListView, as **Figure 14** shows.

Instead of providing a copy of the underlying data, the createSorted and createFiltered methods return a live view over the existing data. That means that any changes to the underlying data are reflected in the views and will be shown in any bound controls. For example, we could add a minor to the underlying list:

```
// Changes to the underlying data are reflected in the live views
var person = { name: "Pete", age: 17, favoriteColor: "black" };
people.push(person);
```

Even though we're changing the underlying data, the sorted view will be updated and the ListView will be notified of the change, as **Figure 15** shows.

Also, because the view from the create methods looks and feels like a binding list, you can further sort or filter the view in a stacked way:

```
// Filterd by age (minors) and sorted by name
window.filteredSortedPeople = people.
  createFiltered(function (p) { return p.age < 18; }).
  createSorted(function (lhs, rhs) { return lhs.name.localeCompare(rhs.name); });
```

The results of binding to the resultant filtered and sorted view shouldn't be a surprise (see **Figure 16**).

Be careful when stacking that you do so in the correct order. In the case of filtering, if you sort or group first and then filter, you're doing more work than you have to. Also, every view layer is overhead, so you'll want to make sure you stack them only as deep as necessary.

Grouping

In addition to sorting and filtering over the binding list data, you can also group it by some criteria of the data. For example, adults and minors would work well for our sample data:

```
// Group by age
window.groupedPeople = people.
  createGrouped(function (p) { return p.age < 18 ? "minor" : "adult" });
```

The return value of the grouping function is a string value that uniquely identifies the group—in our case, either the string "minor" or the string "adult." Binding to the grouped view of the data shows the data arranged by group—that is, all items from each group before moving onto the next group, as **Figure 17** shows.

This example shows just two groups, but you can have any number of groups. Also, just like createSorted and createFiltered, the createGrouped method returns a view over live data, so changes in the underlying data will be reflected in bind destinations.

However, while it's clear to us, the developers, that we're sorting the groups—first the adults and then the minors—it's not clear to users of our program because there's no visual indicator of the grouping. To ask the ListView to provide a visual grouping indicator, the ListView allows you to specify two sets of data: the list of grouped items and the groups themselves. Configuring the ListView with both sets of data looks like this:

```
<div data-win-control="WinJS.UI.ListView"
  data-win-options="{itemDataSource: groupedPeople.dataSource,
                     groupDataSource: groupedPeople.groups.dataSource}">
</div>
```

Once we've created the grouped view of the binding list, the groups property exposes a list of those groups so the ListView can use them. However, before we can make it work, we also need to go back to our use of the createGrouped function:

```
// Group by age
function groupKeySelector(p) { return p.age < 18 ? "minor" : "adult"; };
function groupDataSelector(p) { return p.age < 18 ? "minor" : "adult"; };
window.groupedPeople = people.createGrouped(groupKeySelector, groupDataSelector);
```

Previously, when we were grouping, we just needed to provide a function that could do the grouping based on some unique group identifier. However, if we want to actually display the groups for the user, we need another method that can extract the groups themselves (if you wanted to choose the order of the groups, you could provide a third method to the createGroup method that does the sorting). In both cases, we're providing a string to represent the group as well as to build the list of groups, which works just fine, as you can see in **Figure 18**.

As **Figure 18** shows, the group is actually used to decorate the grouped data, making it very easy to see which data falls into which group. Unfortunately, even with this visual indication of grouping, our JSON-formatted objects aren't really the UI we'd want to show our users. For this to work the way we want requires templates.

Templates

A template is a single-rooted hierarchy of HTML elements with optional "holes" to fill in dynamic data. The data-win-bind attribute

Figure 19 **A Template for the Group Header**

```
<div id="headerTemplate" data-win-control="WinJS.Binding.Template">
  <span data-win-bind="textContent: name"></span>
</div>

<div id="itemTemplate" data-win-control="WinJS.Binding.Template">
  <span data-win-bind="style.color: favoriteColor">
    <span data-win-bind="textContent: name"></span>
    <span> is </span>
    <span data-win-bind="textContent: age"></span>
    <span> years old</span>
  </span>
</div>

<h1>Fancy Grouped</h1>
<div data-win-control="WinJS.UI.ListView"
  data-win-options="{
    groupDataSource: fancyGroupedPeople.groups.dataSource,
    groupHeaderTemplate: select('#headerTemplate'),
    itemDataSource: fancyGroupedPeople.dataSource,
    itemTemplate: select('#itemTemplate')}">
</div>
```

on the div defines the holes using data-win-bind attributes. For example, we could create a template for our person objects like so:

```
<div id="itemTemplate" data-win-control="WinJS.Binding.Template">
  <span data-win-bind="style.color: favoriteColor">
    <span data-win-bind="textContent: name"></span>
    <span> is </span>
    <span data-win-bind="textContent: age"></span>
    <span> years old</span>
  </span>
</div>
```

What makes this chunk of HTML a template is the data-win-control attribute set to the WinJS.Binding.Template control type. Once we have our template, you need to render it to fill in the holes with data:

```
// Manual template rendering
var person = { name: "Pete", age: 17, favoriteColor: "black" };
var template = itemTemplate.winControl;
template.render(person).
  done(function (element) { peteDiv.appendChild(element); });
```

Because the template is just another control (although a control that hides itself until manually rendered), we can reach into the div that defines it and gain access to its functionality via a well-known property called winControl. The functionality we want to access is the template's render method, which takes a data context for use in binding the data-win-bind attributes and produces a fully formed HTML element for us to do what we want with.

If you provide the ListView a template, it will render each item in the ListView with that template. In fact, the ListView can have templates for rendering both items and group headers. To see that in action, let's first update the groups to objects, as is more typical:

```
// Fancy group by age
var groups = [{ key: 1, name: "Minor" }, { key: 2, name: "Adult" }];
function groupDataSelector(p) { return p.age < 18 ? groups[0] : groups[1]; };
function groupKeySelector(p) { return groupDataSelector(p).key; };
window.fancyGroupedPeople = people.createGrouped(groupKeySelector,
groupDataSelector);
```

In this code, we've got two group objects, each with a key and a name, and we've got separate data and key selector functions that return the group itself or the group's key, respectively. With our group data made a little more real-world-like, a template for the group header is created, as shown in **Figure 19**.

The template for the group header is created just like an item template. The group header and item templates are passed to the ListView control via the groupHeaderTemplate and itemTemplate properties in the data-win-options attribute. Our fancier grouped data looks like **Figure 20**.

OK, we admit that's not very fancy, but the combination of groups and items, including templates, shows the power of the binding list. In fact, the binding list is so useful, it's the core of the asynchronous data model exposed from the data.js file generated by the Grid Application and Split Application project templates, as shown in **Figure 21**.

You can see the creation of the empty binding list, the creation of a grouped version of that list using functions that do key and data selection over the groups, a little forEach loop that adds the items to the binding list and finally, a helper function that does filtering.

Fancy Grouped

Minor	Adult
Tom is 16 years old	John is 18 years old
Pete is 17 years old	Chris is 42 years old

Figure 20 **Grouping a Binding List with ListView Headers and Templates**

Figure 21 **The Data.js File Generated by the Grid Application and Split Application Project**

```
// data.js
(function () {
  "use strict";

  var list = new WinJS.Binding.List();
  var groupedItems = list.createGrouped(
      function groupKeySelector(item) { return item.group.key; },
      function groupDataSelector(item) { return item.group; }
  );

  // TODO: Replace the data with your real data.
  // You can add data from asynchronous sources whenever it becomes available.
  generateSampleData().forEach(function (item) {
    list.push(item);
  });

  WinJS.Namespace.define("Data", {
    items: groupedItems,
    groups: groupedItems.groups,
    getItemsFromGroup: getItemsFromGroup,
    ...
  });

  ...

  // This function returns a WinJS.Binding.List containing only the items
  // that belong to the provided group.
  function getItemsFromGroup(group) {
    return list.createFiltered(function (item) {
    return item.group.key === group.key; });
  }

  ...

  });
})();
```

Where Are We?

We started this article by digging into binding, the ability to associate a property on an object with an attribute on an HTML element. Out of the box, WinJS supports one-way, one-time and custom binding initializers, but not two-way binding. Binding is supported on single bindable objects as well as lists of objects that implement the IListDataSource interface. The easiest place to get an implementation of the IListDataSource interface is via the binding list (WinJS.Binding.List) object, which fully supports sorting, filtering and grouping. We also saw how useful templates are when combined with binding and how useful both templates and binding are when it comes to controls that support list binding, such as the ListView. ■

CHRIS SELLS *is the vice president of the Developer Tools Division at Telerik. He is coauthor of "Building Windows 8 Apps with JavaScript" (Addison-Wesley Professional, 2012), from which this article was adapted. More information about Sells, and his various projects, is available at sellsbrothers.com.*

BRANDON SATROM *is a program manager in the Kendo UI Division at Telerik. He is coauthor of "Building Windows 8 Apps with JavaScript" (Addison-Wesley Professional, 2012), from which this article was adapted. You can follow him on Twitter at twitter.com/BrandonSatrom.*

THANKS to the following technical experts for reviewing this article: *Chris Anderson, Jonathan Antoine, Michael Weinhardt, Shawn Wildermuth and Josh Williams*

SharePoint Live Virtual Conference & Expo

Now On-Demand

SharePointVCX.com

PLATINUM SPONSOR

AvePoint
Unleashing the Power of SharePoint™

GOLD SPONSORS

Colligo Networks

Metalogix

thawte

SharePoint Live Virtual Conference & Expo offers in-depth technical training for IT professionals and developers. Learn best practices from the field as our technical experts show you how to deploy, manage and build solutions for Microsoft SharePoint.

Keynote Address:
Steve Fox, Director in MCS, Microsoft

Top Sessions Include:

- Automating SharePoint Governance and Management
- Implementing SharePoint 2013 ECM Solutions
- How We Built Our App for SharePoint 2013 Marketplace
- What's New with Web Content Management in SharePoint 2013

Access this FREE virtual event today: **SharePointVCX.com**

PRODUCED BY

Redmond THE INDEPENDENT VOICE OF THE MICROSOFT IT COMMUNITY

Microsoft TechNet

Visual Studio MAGAZINE

msdn magazine

Authoring Windows Store Apps in Blend

Christian Schormann

Blend for Visual Studio 2012 is a powerful visual authoring tool for creating Windows Store apps built using XAML or HTML. Blend is included with Visual Studio 2012, and it's available directly from the Start screen.

If you're familiar with creating XAML applications for Windows Presentation Foundation (WPF), Silverlight or Windows Phone, you'll quickly feel at home. This version of Blend provides tools that are similar to previous versions, but it also includes support for the Windows 8 XAML platform. In addition, Blend provides new, innovative visual authoring tools for creating Windows Store apps using HTML, CSS and JavaScript.

I'll first take a quick look at what's new for building Windows Store apps with XAML, and then do a more in-depth exploration of the tools in Blend for creating Windows Store apps with HTML.

This article discusses:

- Support for the Windows 8 XAML platform
- Using Blend to create Windows Store apps with HTML
- Using WinJS controls
- Editing a data template and ListView
- Working in interactive mode

Technologies discussed:

Blend, CSS, HTML, Visual Studio 2012, Windows 8, XAML

Using Blend to Create Windows Store Apps with XAML

Windows 8 provides a new XAML platform to support app development. The platform supports not only traditional managed languages (C# and Visual Basic), but also native development in C++. The XAML feature set is similar to that in WPF and Silverlight and includes a set of Windows 8-specific controls and target device properties that Blend supports. In addition, the XAML designers in both Visual Studio 2012 and Blend now use the same code base, resulting in much better compatibility than before. (Overall, Visual Studio 2012 and Blend offer a great, integrated workflow. You can open the same project in both tools at the same time and switch smoothly. I like to write code in Visual Studio to take advantage of the great code editor and debugger, and to design, author and style UX in Blend.)

Here are a few of the highlights of Windows 8 functionality for building Windows Store apps with XAML in Blend.

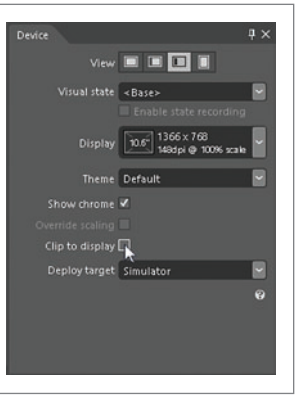


Figure 1 **Device Panel in Blend for Windows Store Apps Built with XAML**

Figure 2 **Blend Runs JavaScript on the Design Surface**

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
      <script>
        function init() {
          var p = document.querySelector('#placeholder');
          if (p) {
            var fruit = ['apples', 'oranges',
                         'lemons', 'pears', 'strawberries'];
            for (var i=0; i<fruit.length; i++) {
              var e = document.createElement("div");
                e.textContent = fruit[i];
                e.className = 'fruit';
                p.appendChild(e);
            }
          }
        }
      </script>
  </head>
  <body onload="init()">
    <div id="placeholder"></div>
  </body>
</html>
```

Built-in Support for Windows 8 Controls Blend has built-in authoring support for Windows 8 controls such as AppBar and ListView, making it easy to build applications with these iconic controls. The AppBar is a fly-in control that's normally hidden, but the Blend design surface helps you bring the control into view for interactive editing. ListView, with all the associated templates for data items, group styles and more, is also fully supported in Blend.

View Authoring Windows Store apps are expected to run on a variety of form factors, at different display sizes and pixel densities, and to adjust to different views (portrait, landscape, snapped and full). The Device panel (**Figure 1**) helps you create, edit and modify the app with different views selected on the design surface. The Device panel is also integrated with the Visual State Manager.

Using Blend to Create Windows Store Apps with HTML

Blend is a unique environment for creating, styling and iterating HTML-based UI design and for creating clean, professional, standards-compliant markup. Unlike traditional HTML authoring tools, Blend can handle design scenarios for apps that use JavaScript to create or modify content on the fly. The Blend design surface uses the same rendering engine as the Windows Runtime, but it also runs JavaScript from the moment you load a page, which ensures accurate visualization of both static and dynamic page elements. The Live DOM panel and the rich, productive CSS tools in Blend help you work with the elements you create in markup as well as with every element that comes from script or is loaded from a fragment.

In the next few sections, I'll cover these and other highlights of the HTML functionality in Blend.

Code on the Design Surface Running code on the design surface for accurate rendering of code-generated pages is the bedrock of building Windows Store apps with HTML in Blend. For example, the markup in **Figure 2** includes both standard HTML and JavaScript. When you paste this markup into an empty page in Blend, the JavaScript runs and is executed directly on the design surface in edit mode.

The elements generated by the JavaScript are rendered correctly on the design surface. Because the design surface runs code in edit mode, you can select JavaScript-generated elements just as you would any other element.

The elements also show up in the Live DOM panel (**Figure 3**), which represents the dynamic state of the Document Object Model (DOM) tree, not just the content of the markup document. Elements not present in markup, such as those created by JavaScript or dynamically loaded from a fragment, are marked with lightning-bolt icons to indicate that these elements were generated by script.

The Live DOM panel also shows which classes are attached to each element. For example, the code in **Figure 2** generates the class name "fruit" for each generated element. While you can't directly modify JavaScript-generated elements (there's no markup for Blend to modify), you can still style against the classes, provided that the generating code uses CSS class names or the generated elements can otherwise be accessed with a CSS selector.

Rich CSS Property Editors Blend provides many rich property editors that make complex CSS properties accessible, as well as lots of support for interactive manipulation of elements on the design surface. For example, Windows 8 supports a new standard CSS layout model called the CSS grid. The CSS grid is an amazingly useful layout, especially for resizable applications that need to run on multiple form factors. Grid layout works with a set of rows and columns, similar to a table, where each row and column can be assigned a specific resize behavior. Rows or columns can be set to a fixed size, to resize proportionally, or to automatically fit their content. If you're familiar with the XAML grid, you'll be right at home with HTML grids.

CSS grid editing in Blend takes the stress out of many CSS layout scenarios (see **Figure 4**). You can draw and modify grids right on the design surface, see in-place measurements [1], modify sizes and units using in-place on-object UI [2], insert and delete rows and columns, and much more.

Blend also provides color editors and support for multilayer backgrounds and CSS gradients (**Figure 5**), all with immediate

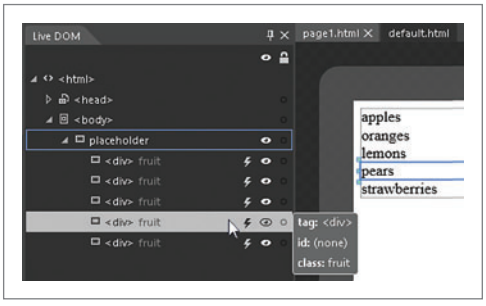


Figure 3 **The Live DOM Panel**

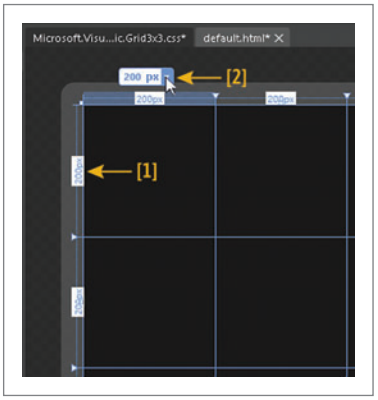


Figure 4 **Artboard Editing of CSS Grid**

Figure 5 **Color Editing Includes Visual Feedback**

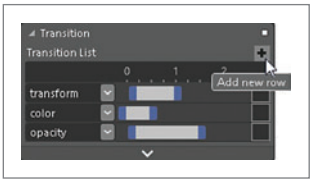


Figure 6 **Editing a CSS Transition**

visual feedback. In addition, you can easily edit CSS transitions in Blend. **Figure 6** shows the CSS transition editor with a staggered transition defined for three properties, including easing functions.

CSS Editing Styling with CSS is at the heart of visually authoring HTML-based UIs. Blend is built around a sophisticated set of tools for CSS styling. For example, the Style Rules panel (**Figure 7**) shows you all the style rules that are applied to the currently open HTML document.

In **Figure 7**, you can see how style rules are sorted by stylesheet and in declaration order [1]. Search is supported for larger collections of style rules. The Style Rules panel also shows media queries [2] and the style rules defined within each query. You also see which media query, if any, is currently active. Because Windows Store apps make use of media queries to handle various view states, this is very important for app styling.

Style rules can be created, edited or deleted in the Style Rules panel (although Blend has many productivity shortcuts for style rule creation). During the creation of CSS selectors, Blend helps with IntelliSense for CSS selectors. As you type a selector, Blend provides a list of completion options in the context of the current document, as you can see in **Figure 8**.

IntelliSense for CSS selectors also highlights the elements targeted by the selector while you're typing. This feature is called the rule scope adorner. The rule scope adorner is not only displayed by IntelliSense; it also appears any time a style is selected directly or indirectly, which makes it easier to identify which elements are affected by the selected style rule. In **Figure 9**, the rule scope adorner is shown with green outlines. CSS selectors can become quite complex, so you'll find this to be a very useful utility.

You can always select style rules directly in the Style Rules panel and then edit the CSS properties for that style rule in the CSS Properties panel. In many cases, however, it's easier to select an element on the design surface or in the Live DOM panel and find the style rules that affect that element.

Whenever you select an element on the design surface, the CSS Properties panel (**Figure 10**) displays a list of style rules that

Figure 7 **Style Rules Panel**

Figure 8 **IntelliSense for CSS Selectors**

are applied to the element in order of precedence. This makes it much easier to find the style you need. By default, properties are shown in a categorized view, but an alphabetical view is also available. The CSS properties list is fairly long (currently more than 300 CSS properties are defined), so to locate a property, you can search the property list or filter it to see only properties that are currently being set in the rule.

In many cases, the properties that contribute to the style of an element come from many different style rules. This can make it difficult to understand why an element looks the way it does. In these cases, a roll-up view of all "winning" properties helps determine which styles are being applied (**Figure 11**).

The Winning Properties view shows only the properties that rise to the surface of the CSS cascade, sorted by the style rule of origin. This view is extremely useful for diagnostics, but also for the quick tweaking of an existing property value.

The Computed Values view shows properties with the values that the browser sees for rendering after processing the CSS declarations. This can sometimes be a great help when you're trying to resolve problems. Blend also provides access to the CSS cascade, another tool for diagnosing problems in your display.

For every property, you can view the CSS cascade to see the values from all the rules that affect a given property. For example,

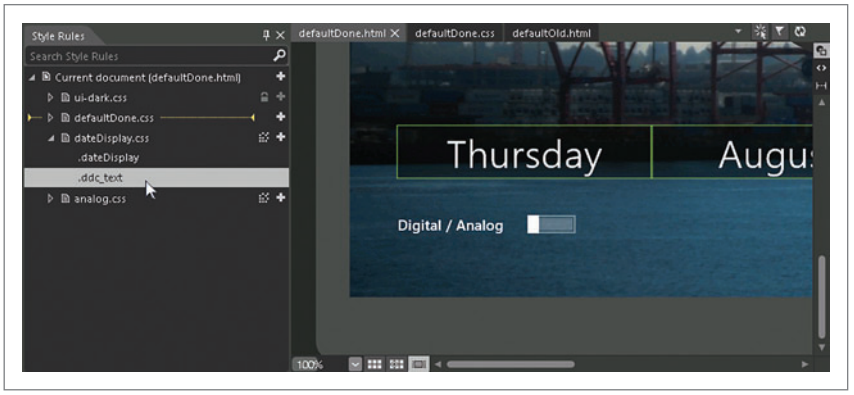


Figure 9 **An Example of the Rule Scope Adorner**

Figure 10 **CSS Properties Panel**

Figure 11 **View Winning Properties**

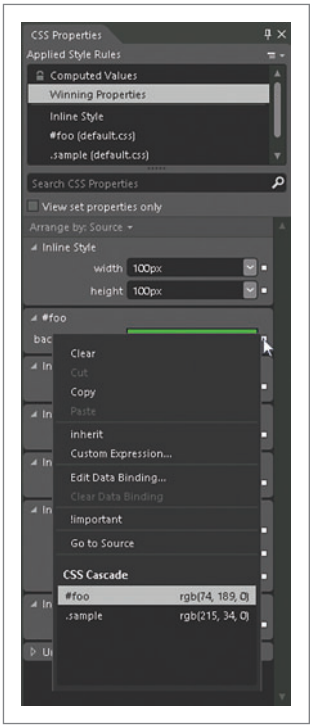


Figure 12 **A View of the CSS Cascade**

you might have a background-color property that's defined by two different rules, with the one on top being the one that "wins" for that property. The CSS cascade display (**Figure 12**) also lets you navigate quickly to competing rules.

Blend offers many shortcuts to quickly create style rules for selected elements. You can create a rule that targets the ID of the selected element (#foo, if the element has an ID of foo), or create rules matching any of the classes in the className attribute. You can also add and remove classes quickly, or add a new class and create a rule for it, in one single step (see **Figure 13**). These in-context productivity gestures make styling with Blend fast, smooth and efficient.

Last but not least, because everybody makes mistakes and changes his mind, Blend provides some refactoring functions. With these, you can quickly cut and copy entire style rules, all property values from a style rule or just selected property values. You can then paste the copied rules or properties into a new or existing style rule.

WinJS Controls In the Windows Library for JavaScript (WinJS), Windows provides a set of HTML-based controls for Windows Store apps. This includes simple controls such as a toggle switch, as well as sophisticated controls such as ListView or FlipView. WinJS controls are similar to controls in other UI toolkits, such as JQuery. One difference is that WinJS controls can be used from code (and also declaratively from markup) by using standard data attributes to apply the control metadata. This mechanism allows Blend to provide a first-class authoring experience for the controls.

WinJS controls are created in Blend by using the Assets panel. **Figure 14** shows the Assets panel with the category of the most common controls displayed. You can insert controls or HTML tags into the page by dragging or by double-clicking the asset. You can also search for and filter results in the Assets panel.

After you've added a control to the design surface, you can configure it by using the HTML Attributes panel. The configuration options for the selected control are shown in the Windows App Controls category in the panel. For a toggle switch, for example, you can set the labelOff and labelOn attributes or modify the title.

When the page loads, the WinJS toolkit finds elements with data-win-control attributes, and it then creates the backing control, with the options set in data-win-options. The control implementation, as with HTML controls in other toolkits, creates the elements that display the control dynamically. Because Blend runs code on the design surface, the control will be displayed accurately in edit mode. You'll also see the dynamic elements created by the control. The JavaScript-generated elements are identified by a lightning-bolt icon.

Data Template Editing and Fragments Even though a ListView is a much more complex control than a toggle switch, it's configured using attributes in a similar way. Beyond this, Blend has deep support for data-template editing with a ListView. A data template is a snippet of HTML that's used to render every single data item in a list. Live data-template editing in Blend helps you design and style data templates in place, within the ListView, with the updates reflected accurately.

You can create an empty default data template right from the attributes editor and then use the Blend authoring tools to edit the template. **Figure 15** shows the markup for the simple ListView with a data template that's shown in **Figure 16**.

One div defines the ListView, and another div, marked as a template control, provides the root for the data template. The content of this div (in this case, the div with the class cityItem) is instantiated for each data item the ListView renders. The template

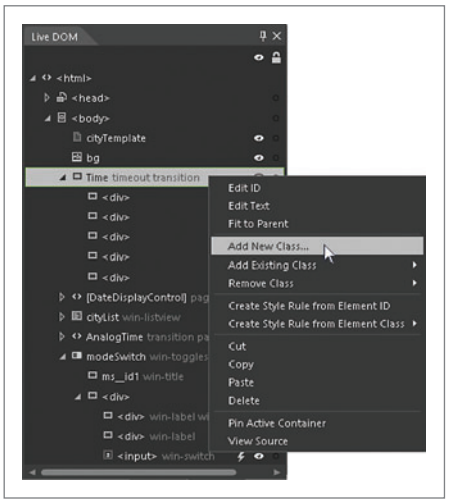


Figure 13 **Add a Class and Create a Style Rule in a Single Step**

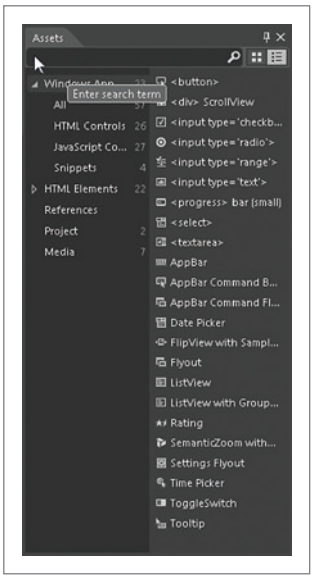


Figure 14 **Blend Assets Panel**

Figure 15 **ListView with a Data Template**

```
<div id="cityTemplate" data-win-control="WinJS.Binding.Template">
  <div class="cityItem">
    <img class="cityImage" data-win-bind="src:image">
    <div class="cityLabel" data-win-bind="textContent:name"></div>
  </div>
</div>

<div id="cityList"
  data-win-control="WinJS.UI.ListView"
  data-win-options="{
    itemDataSource:AppData.cities.dataSource,
    itemTemplate:select('#cityTemplate'),
    layout:{type:WinJS.UI.ListLayout}">
</div>
```

control attached to the template div also ensures that the template isn't actually visible on the page.

This poses a bit of a problem for visual editing because to edit a data template, you need to edit the content of the template. However, the data-template control hides the template content—to keep it out of the way of the "real" page content—so you can't see anything to edit. But even if the template content were visible, what you really want to see and edit is the data template in the context of the ListView, not outside of it.

Figure 17 illustrates what you need to do to edit a data template. Select an element in the ListView, directly on the design surface [1]. You can see in the Live DOM panel that the selected element (image) is a dynamically generated element inside the ListView [2]. The ListView is also displayed in the Live DOM panel [3].

In code view, the img tag inside the data template [4] is highlighted. At the top of the design surface, information tells you that the selected content originates from a data template [5].

When you select the element in the ListView, Blend traces the origins of the element displayed. Blend identifies this element as coming from a data template and displays this information. Now, as you edit the element, Blend automatically makes the relevant edits in the markup and styles of the data template, updating all the items displayed in the ListView as you go. No manual updates are required.

The ability to see through a ListView into a data template is one of my favorite Blend features. This also works with fragments. A fragment is an HTML page that's dynamically loaded into another page using a WinJS utility function or an HTMLControl (a special WinJS control). Just as with templates, Blend detects fragments on a page and provides the same level of in-place editing. You can see fragments in place and in context (**Figure 18**) the way they're meant to be used, and fully edit them, without restriction and without having to open another document.

Interactive Mode

Interactive mode is one of the best and most useful features in Blend. As I said before, Blend always runs your code on the design surface. In normal edit mode, it prevents you from interacting with the app, so you can only select, manipulate and edit elements that are visible on the artboard. Interactive mode takes away this protective layer so that you can interact with the app as it's running.

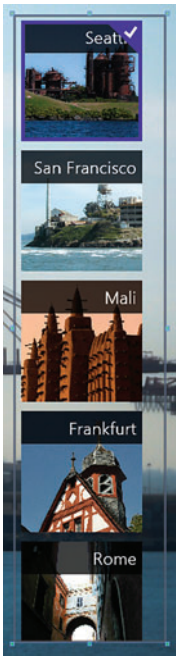


Figure 16 **ListView Using Data Template**

So why not just run the app? The most important reason is the ability to accumulate state. When you run the app in Windows or in the Simulator, the running instance is completely detached from what you have on the design surface. Any interaction you have with the running app changes only the state of the running app. The moment you stop this instance, the accumulated state is gone.

As you switch between interactive mode and edit mode, the application state is preserved. If you do something as simple as change the state of a toggle button, bring in a fly-out, change options or even create a drawing on a canvas, when you return to edit mode, your state is preserved. This enables you to edit and tweak your app in states you would normally never see on the design surface. This is incredibly

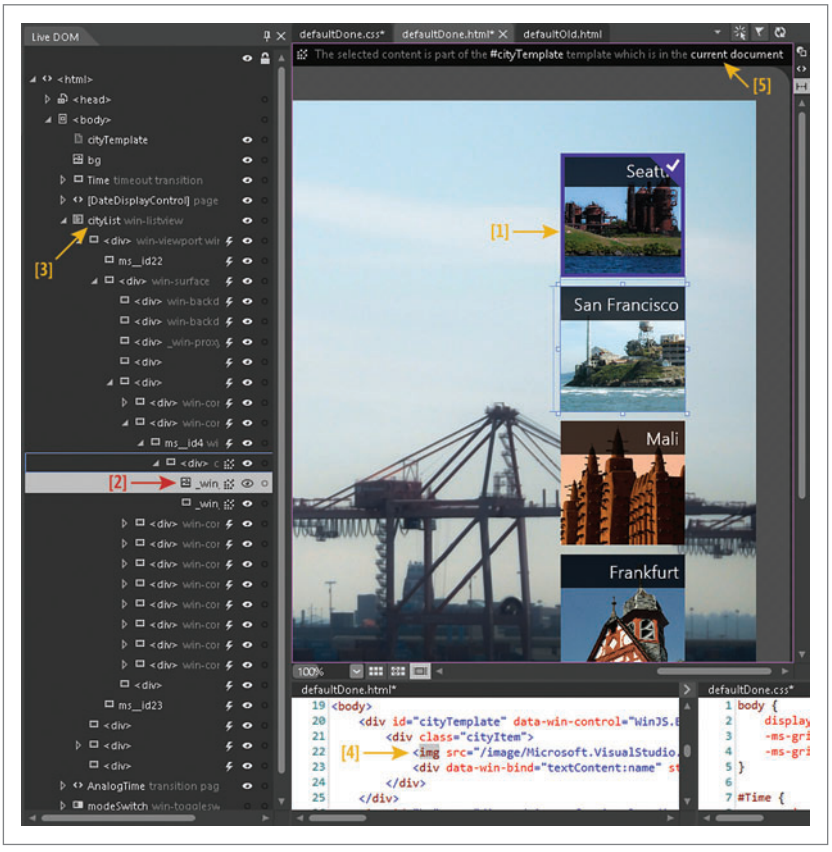


Figure 17 **Editing a ListView and Data Template**

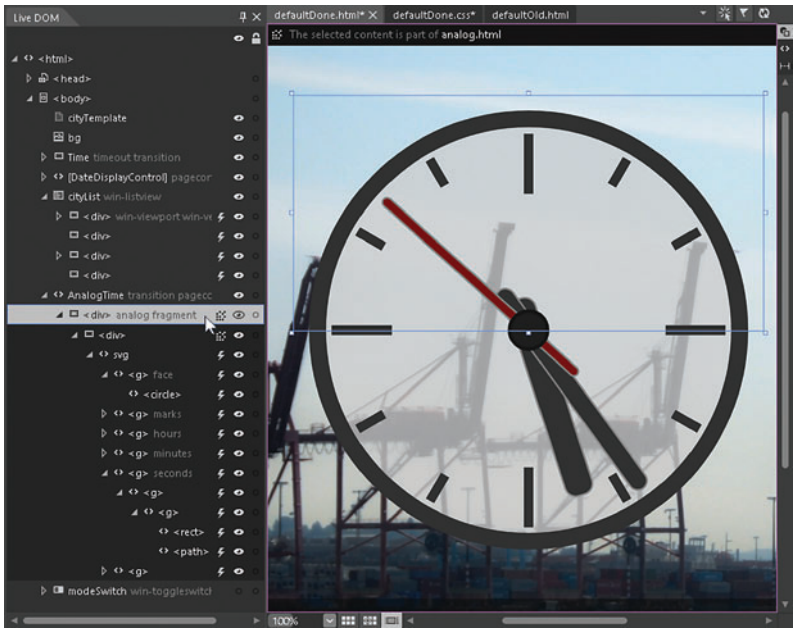


Figure 18 **Editing an HTML Fragment**

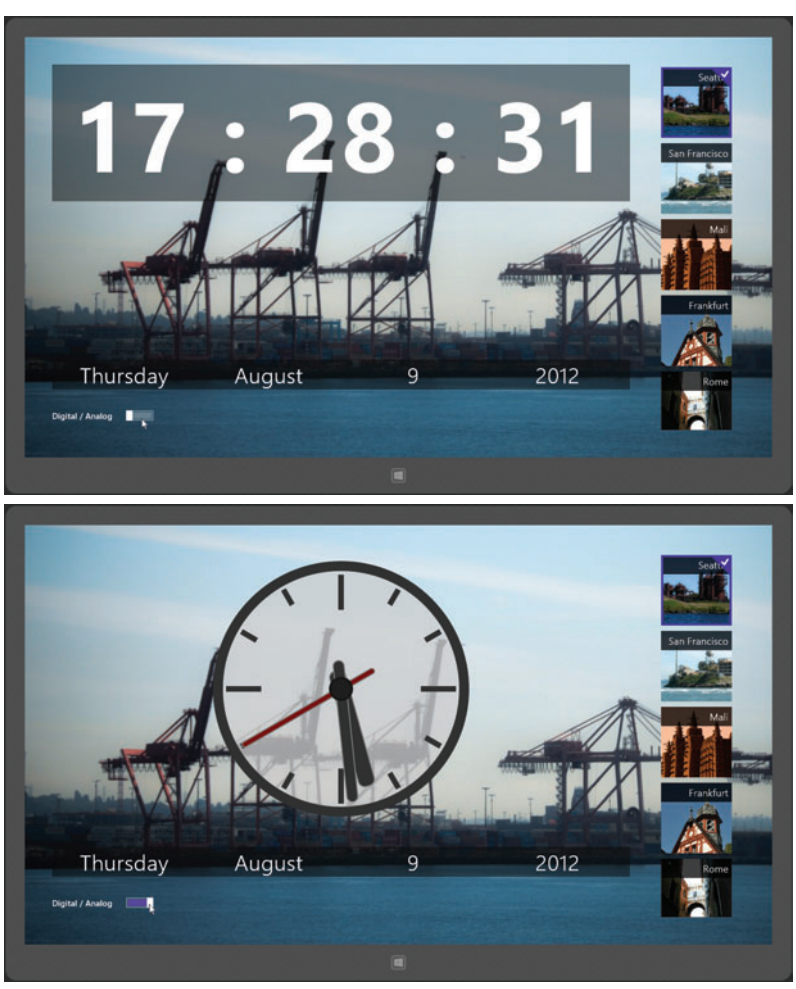


Figure 19 **Switching the Dynamic App State via Interactive Mode**

convenient, fast and liberating because you can now edit visually in states that can't be reached statically at all. In many ways, interactive mode is like the developer tools in a browser, but intimately tied to the actual source project and the design surface.

Figure 19 shows two views of a simple world clock app. To see the second view, some code needs to run, which is triggered by the mode switch toggle. In edit mode, there's no way to make this code execute, but as soon as I switch to interactive mode I can simply slide the Digital/Analog toggle and the view changes, ready for edit. I can't imagine working without interactive mode anymore.

Designing for Devices

Windows Store apps run on a wide variety of devices, from small tablets to large desktop monitors. These devices come in a wide variety of resolutions and display pixel densities. In addition, Windows Store apps can be in different view modes (landscape, portrait, snapped and filled mode).

When authoring adaptable applications, it's important to be able to see and edit the app in a variety of different view states. Blend lets you do this, accurately displaying the different scaling modes on the design surface, controlled by the Device panel.

Wrapping Up

Blend for Visual Studio 2012 provides visual authoring for Windows Store apps, with support for both XAML and HTML. The XAML functionality is similar to previous versions of Blend, so if you've used previous versions you'll feel right at home. In addition to rich support for Windows Store app development, Blend for Visual Studio 2012 also supports enhanced compatibility with the XAML Designer in Blend.

Blend support for HTML represents a new and innovative kind of authoring environment for HTML. Blend can handle not only HTML and CSS markup, but also the rather frequent pattern of JavaScript-generated content. Most important, Blend for HTML makes visual authoring of HTML, CSS and WinJS productive, fast and fun. ■

CHRISTIAN SCHORMANN *is a partner program manager on the Blend team. His passion is building visual authoring tools for designers, artists and developers.*

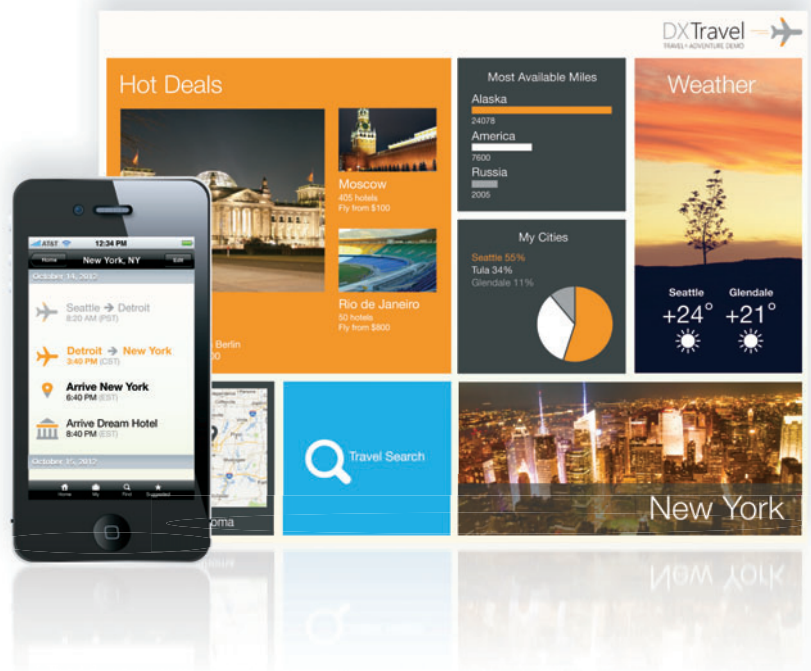
THANKS to the following technical experts for reviewing this article: *Joanna Mason, Unni Ravindranathan, Josh Pepper and Erik Saltwell*

DevExpress

Let's see what develops.

discover DXTREME.

Delight your users by creating apps that feel as though they were designed expressly for the device. With **DXTREME**, multi-channel means building applications that span devices and optimize the best parts of each platform. And with HTML5/JS visualization built in your dynamic charts and graphs will be both powerful and beautiful.



Easy does it.

Are you ready to go cross-platform?

Download the **DXTREME** Preview to experience the future.

www.DevExpress.com

DXv2

The next generation of inspiring tools. **Today.**

u dxt dxp asp win wpf sl 8

> ComponentOne offers the most well-rounded offering of tools and components covering all the major platforms.
>
> *Randy Hearn*
> *Senior Research Analyst*

- *Endless controls, both UI & data*
- *Modern themes for Windows 8*
- *Flexible .NET API*
- *OLAP tools for advanced data analysis*
- *Support for all .NET platforms*

ComponentOne®
Ultimate™

ComponentOne®
a division of GrapeCity®

Download your free trial @
componentone.com/ult