

XML RSS -- News You Can Use

by Kenn Scribner

It was an unusually chilly February around here. I pulled my coat up over the back of my neck to keep the frosty wind at bay, crossed the street, and headed for the cafe. The CodeCafe is a regular haunt of mine--it's a great place to grab a drink (like most developers, I love my caffeine), and quite often I learn something there. After dodging a few icy patches in the street, I grabbed the door handle and walked inside.

Johan, the owner, is always baking something, and today it was some sort of cinnamon roll with pecans. I try to watch my calories, but given the temperature outside, I figured that maybe today I could munch one of the cinnamon rolls while I sucked down some Diet Dew and reviewed some code. The CodeCafe is well known for its menu of interesting code projects, and I was looking forward to today's special.

"Hi, Kenn...grab a seat and I'll be right with you. The usual?" Angie, the cafe's waitress, hailed me from across the dining room. "The usual" for me is a cold Diet Mountain Dew and a wireless connection, but since I had cinnamon on the brain, I figured I'd grab a booth in back so I could indulge my sweet tooth in relative peace.

"Sure," I called back, "but I'll have one of Johan's cinnamon things too!" I smiled. Angie knows I can't help myself when it comes to confectionery.

I headed toward the booths in the back and took a seat. It wasn't long before Angie sidled up with my Diet Dew and cinnamon roll. Happily, it seemed to be one of the larger ones Johan had baked. So much for counting calories. Ah, the perks of being a regular customer! "What's the special today, Angie?" I asked.

"Good one today," she replied. "We're doing RSS."

"RSS?" I asked. "What's that?" If there's one thing we do well in this business, it's creating opaque acronyms. Anyone remember RDO, RPC, or PID? Half the battle is remembering what the acronyms stand for!

"Oh...well, RSS is a simple XML vocabulary that's used for syndicating news feeds. Basically it's blogging, but it can also be used for truly newsworthy information," she said. Personally, I'm not sure I have anything to say interesting enough that the whole of the Internet needs to read it, but there are those who do. It's a wonderful technology, though, that allows the everyday man to speak his mind to the entire world. I decided that today's special was for me.

"Hey, sounds cool," I said. "How does it work?" Angie began describing a very simple XML vocabulary for presenting news items. It wasn't long before I got the basic idea, but something nagged at me.

"You're right," I said. "RSS really isn't too difficult to grasp." Now I was going to hit her with my big question. "But XML provides content...what displays the XML, and how do we keep from requesting RSS information too often and flooding the various servers with requests?"

Angie smiled again. This smile, though, was the "Ah, Grasshopper, you have much to learn" kind of smile. She then started in talking about asynchronous .NET Web programming, isolated storage, and even DHTML (another acronym). I got the idea, but my head was spinning. For such a simple XML vocabulary, the news feed viewer application was actually rather complex! As I usually do, I took copious notes so I could pass the information on to you.

XML RSS

The acronym RSS is reputed to stand for "Really Simple Syndication," but nobody really knows for sure except perhaps the author, Dave Winer (of SOAP fame). And since Dave originally proposed the vocabulary, there are now three popular versions: 0.91, 0.92, and 2.0. (I can't explain the versioning, and neither could Angie.) Although some of the details differ between versions, the required elements have remained constant, so any viewer you might write should be able to display all three versions without too much difficulty. Angie gave me the URL of the RSS 2.0 "specification."

```
http://blogs.law.harvard.edu/tech/rss
```

I popped open my tablet PC and typed in the URL. When the page came up, I could see that the XML would very easily be deserialized into C# objects, so I made a mental note to look into using the .NET Framework's XML Serializer to make my programming life easier.

RSS, as an XML vocabulary, has a document element named "RSS":

```
<rss version="2.0"/>
```

Within the document element you'll find a collection of channel elements.

```
<channel />
```

While clearly there might be more than one channel in any given RSS document, so far Angie had never seen more than one included at any given time. I'm sure there is no rule stating that only a single channel can ever be present, but I figured a sample viewer could skip additional channels if any existed. You could add multiple channel support easily enough.

Channels have three required elements, several optional elements, and contain a collection of news item elements. The news item elements are not strictly required, but without them, what news would the feed be presenting? Anyway, the required elements include the channel title, a link back to the sender, and a description:

```
<title />  
<link />  
<description />
```

The news items themselves are contained within an item element.

```
<item />
```

The item elements have the same three required RSS elements that the channel incorporates: the title of the news item, the link to the HTML article itself, and a description.

```
<title />  
<link />  
<description />
```

Putting it all together, and showing only the required RSS elements for the moment, a very small RSS document might look like the XML you see in Figure 1. This is actual RSS information from the **MSDN Web site**.

```
<rss version="2.0">  
  <channel>  
    <title>MSDN: Visual C#</title>  
    <link>http://msdn.microsoft.com/vcsharp/</link>  
    <description>The latest information for developers on Visual  
      C#.</description>  
    <item>  
      <title>MSDE 2000 Walkthrough: Build a Data-Driven Website  
        Using  
          Visual C# .NET and Visual Studio .NET 2003</title>
```

```
<description>Use this walkthrough to create a data-driven
website
with MSDE and ASP.NET 1.1, and Visual C# .NET code
written in Visual Studio .NET 2003.</description>
<link>http://msdn.microsoft.com/data/default.aspx?
pull=/library/en-
us/dnmsde2kwrk/html/mypicscsvs.asp</link>
</item>
</channel>
</rss>
```

Figure 1: Sample MSDN RSS XML feed document

If there were more channels, you would see additional `<channel />` elements, and the same holds true for items within the channel you do see. More items could be and usually are present.

The optional elements differ slightly between the various RSS versions. For 2.0 (the latest version), optional channel elements include such things as the publishing date, language, copyright information, managing editor, "ttl" (time to live, or how long the information should be considered valid from the publishing date), and so forth. Most feeds I've seen ignore these optional fields, with the occasional exception of the publishing date and ttl values. The RSS 2.0 specification has a good description of these elements, and you'll see them in code when I write my RSS viewer a bit later in this article.

Item elements also have optional informational elements. The information contained here includes an author element, the publishing date (of the item, not the channel--they could differ), a unique identifier, additional comments, and so forth. I looked at a lot of RSS feeds, and sadly, none of them, including those from Microsoft, used the unique identifier element. Later, you'll see that when my sample viewer needs to decide whether any given news item has already been read (for channel update purposes), it has to examine the title of the item or its associated link instead of simply comparing unique identifier values. It's a shame that the sample app has to use a less efficient method. Check out the RSS 2.0 specification for the semantics and syntax of these optional elements--they're very well explained and there are truly very few of them.

I'm omitting a detailed discussion of the optional RSS elements, because in the dozens of RSS documents I examined while testing my viewer, no feed implemented more than a couple optional elements, and those generally were limited to publishing dates and ttl values.

You can see that there truly isn't much to the RSS XML vocabulary. Figure 1 is a full, complete, and perfectly valid RSS XML feed document, and it is all your RSS viewer would need to process to display the feed information. Ah, but as Angie pointed out to me, as she has so many times in the past, it's the details that will get you. I'll highlight those details as I describe my sample RSS viewer.

Before I get to the .NET code, though, you might wonder, "How do I get the RSS feed document?" Or even "How do I know what RSS feed documents are available?" There is no single repository for RSS feed information (akin to **UDDI** for Web Services, although you could use UDDI for that purpose). You, as an RSS information consumer, must discover these on your own. Once you discover an RSS feed, you obtain the RSS XML by downloading the XML from the feed's Web site in the same fashion as you'd download HTML. You simply use an HTTP GET verb to retrieve the XML, and, once obtained, you parse the XML and shred out the channels and items of interest. I'll show both of these in my sample code.

Since my goal here is to present a simple RSS viewer and touch upon some tricky related programming issues, my viewer is pretty bare bones. It's complete, mind you, but it isn't necessarily feature-rich. Several high-quality RSS news viewers are available, including **RssBandit**, that come with source code.

WinGator

RSS as an XML technology is very simple and frankly not terribly interesting. All you really have is a collection of hyperlinks with a title and description for each. But the application that would show you the RSS information can be quite interesting! Angie mentioned some things you should consider when writing applications in general that rely upon Internet-based information.

For one thing, all of your communication with the remote site should, generally speaking, be asynchronous. That is, you'll force a separate thread to do the actual communication with the remote site and then have it notify your main application when the information has arrived. This prevents the thread handling the user interface from also handling the communications, which in turn prevents the user interface from becoming unresponsive. Users hate unresponsive user interfaces. So do programmers such as me, especially when I paid for the software. Of course, this also makes for more complex software.

Since you don't have some sort of notification process available with RSS, where you would be notified when updates are available, you'll need to resort to polling news feeds. While in general this isn't that big of a problem, you should limit how often users can poll. Polling too often can resort in a choked server or perhaps having your network address blocked as a potential security threat. There isn't necessarily any hurry to update your feed information. Depending upon the news feed, you could even poll once per week for updates! (Angie admitted she hadn't updated her RSS information in weeks.) Other feeds should be updated more often, such as CNN's. My sample application limits the polling interval from 20 minutes to an hour, with the default being an hour.

But enough preliminaries; let's get to the application itself. The basic user interface is shown in Figure 2.

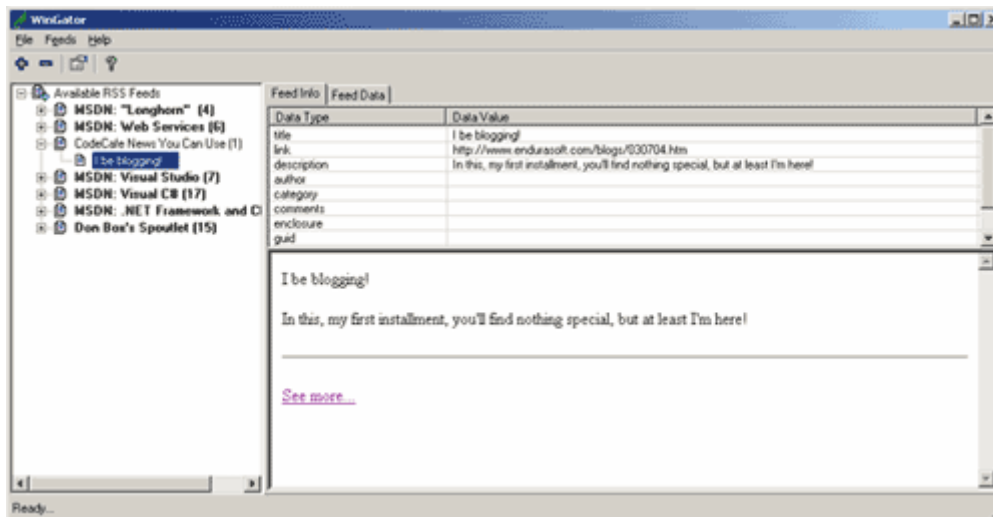


Figure 2. Basic WinGator user interface

The tree control on the left displays your known RSS feeds along with the items associated with the first channel for each feed, when the tree nodes are expanded. The tab control shows you the contents of the news item's XML (top frame) and formats the basic information for display (bottom frame). Using DHTML, I generate the "See more..." hyperlink; when it's clicked, I intercept the event, cancel it (so I don't display the link in this frame), and force the second tab to navigate to the actual news link (see Figure 3).

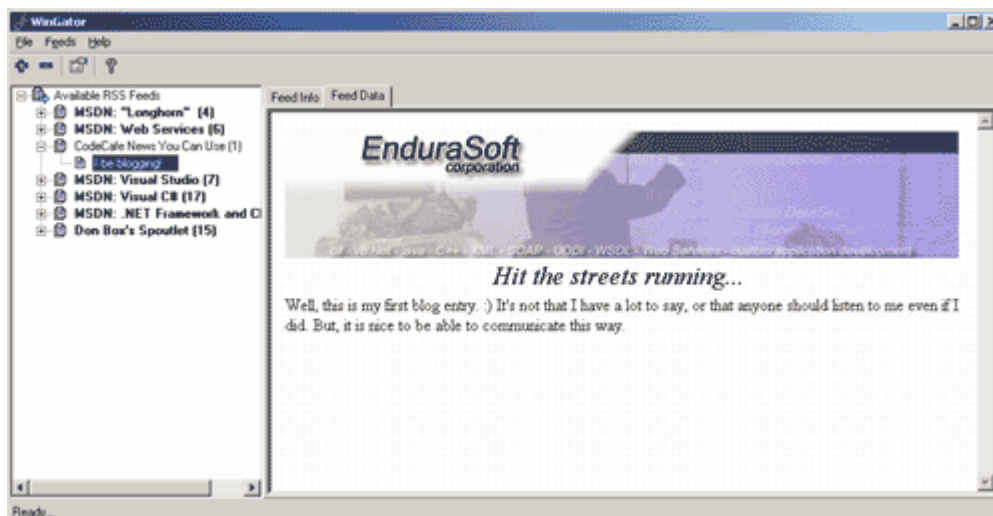


Figure 3. WinGator with displayed news feed information

As items are read, the tree node's font is adjusted, from bold (unread) to normal (read). If all of the available items within a channel are read, the feed's tree node font is similarly adjusted.

To ease the burdens on the news servers, I cache news information in files that I maintain in isolated storage. Isolated storage is an application-specific on-disk area you can use to store data. While you could stuff information into files located in the same directory as your executable file, or you could put information into the Registry, isolated storage is a more flexible and desirable option. The files in isolated storage can rove with the user in a networked environment and you don't fill the (limited) Registry resource with what could conceivably be very large cache files.

But caching data also means that I'll need to update the cache when the application initializes. Since updating many RSS feeds could take some time, especially if the user is on a dial-up Internet connection, I really don't have enough information to display a full user interface. Instead, I pop up a splash screen that provides some feedback as feed updates are completed (see Figure 4).



Figure 4. WinGator splash screen with update status.

Once all of the feeds are updated, or after a minimum display period has expired (whichever is longer), I'll tear down the splash screen and display the main user interface. As part of the update process, the screen location and state of the user interface is persisted, so that the next time it's run, it pops up in the same position and with the same state (minimized, maximized, and so on).

There is much I could discuss about this application, but I thought I'd hit three main areas of interest: RSS feed handling, isolated storage, and the DHTML work I do in the detail frame.

RSS Feed Handling

As soon as I saw the RSS XML I knew that the .NET XML Serializer could easily handle converting the raw RSS XML into nice managed objects for me. And in theory, this is true. In practice, I wound up building my own (very simple) RSS XML deserializer to supplement the .NET serializer. The reason for this was that the incoming RSS XML, even from Microsoft RSS feeds, all too frequently would contain invalid XML character data that would cause the .NET XML Serializer to throw exceptions. Roughly 8 out of 10 feeds I tried would not deserialize properly using the .NET XML Serializer, so I wrote my own that I employ if the feed confuses the Framework object. I actually tried nearly 30 different feeds, and most didn't deserialize as I'd hoped.

To keep the description of my process brief, since it generally applies elsewhere very effectively, I'll talk about using the .NET XML Serializer as if it were a perfect solution. When faced with this sort of XML programming task, I rely upon a .NET Framework SDK program called `xsd.exe`. `xsd.exe` is capable of reading an XML file and deriving an XML schema from the file. I can then run that same schema through `xsd.exe` yet again and have it automatically generate C# or VB.NET source files that can be used with the .NET XML Serializer.

In this particular case, since none of the feeds implemented all of the optional XML elements, I had to add the missing elements by hand just prior to running `xsd.exe` the first time. After all, even though they're not commonly used, I still want to have access to them.

Once I did this, I was left with a C# source file called `rssdata.cs`, which I added to the WinGator solution. I've edited the file contents for brevity, but the basic file contents (required RSS elements) are shown in Figure 5.

```
//-----  
// <autogenerated>  
//   This code was generated by a tool.  
//   Runtime Version: 1.1.4322.573  
//  
//   Changes to this file may cause incorrect behavior and will be  
//   lost if the code is regenerated.  
// </autogenerated>  
//-----  
  
//  
// This source code was auto-generated by xsd,  
// Version=1.1.4322.573.  
//  
using System.Xml.Serialization;  
  
/// <remarks/>  
[System.Xml.Serialization.XmlRootAttribute(Namespace="",
```

```
        IsNullable=false)]
public class rss {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("channel",
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public rssChannel[] channel;

    /// <remarks/>
    [System.Xml.Serialization.XmlAttributeAttribute()]
    public string version;
}

/// <remarks/>
public class rssChannel {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public string title;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public string link;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public string description;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute("item",
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public rssChannelItem[] item;
}

/// <remarks/>
public class rssChannelItem {

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
    public string title;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(
        Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
```

```
public string link;

/// <remarks/>
[System.Xml.Serialization.XmlElementAttribute(
    Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
public string description;

// Added to support the UI...this won't affect incoming RSS
data,
// but will help tremendously when working with the UI.
public bool unread = true;
}
```

Figure 5. (Edited) rssdata.cs source file contents

From Figure 5 you see the required RSS XML elements manifested as C# source code (as public properties). I did edit this file (ignoring the warning at the beginning of the source file) and added a property I can use to quickly determine whether or not an item has been read. This won't affect deserialization, but will make the UI code very much simpler. Purists would tell me I probably should have derived an entirely new class from `rssChannelItem` and added my `unread` property to that, which would insulate me from future regenerations of the `rssdata.cs` file, but I don't see this file updating very often. Normally I'm with the purists, but in this case this works quite effectively for the moment.

Given this source file and resulting managed objects, once I obtain RSS XML, I can very easily turn that XML into the managed objects and work with the information as classes and collections instead of as raw XML. Assuming the XML is contained within the stream named `strm`, Figure 6 shows you how to deserialize the RSS XML.

```
XmlSerializer ser = new XmlSerializer(typeof(rss));
rss rssFeedData = (rss)ser.Deserialize(strm);
```

Figure 6. Deserializing RSS XML

And, of course, in Figure 6 you can see why you'd take the time to use `xsd.exe` to generate the C# (or VB.NET) source files and use the .NET XML Serializer. To accept, parse, convert, and consume an entire RSS feed's XML stream, I wrote 2 (yes, TWO) lines of code! Well, I'm omitting the one line of code I added to `rssdata.cs` for read status, but I think you probably see my point. This capability is hugely powerful, and I employ it whenever and wherever it makes sense to do so.

The real trick lies with obtaining the RSS XML feed data in the first place, and to do that, I use the Framework's Web capabilities. In `WinGator`, I poll for RSS feed data using both synchronous and asynchronous means. I use the synchronous style for making the initial

contact with the feed's provider, just after the user provides a new feed URL. In this specific case, I believe it's fine to lock the user interface and ascertain whether or not the URL provided is a valid RSS feed data URL. For all other cases, I use the asynchronous Framework Web calls. I wrapped both of these RSS feed data calls into a single worker class I named `FeedGrabber`.

The synchronous call simply hits the URL, as provided by the user, and retrieves the response stream to be deserialized. If the feed information were pristine, I could use the same two lines of code shown in Figure 6. But since the titles often contained invalid XML (even from Microsoft), and since I want to do my best to get the user the news information, I made things a bit more complex by loading the XML in the response stream into an `XmlDocument` instance for later use in case of deserialization failure. The synchronous `GetFeed()` method is shown in Figure 7.

```
public rss GetFeed()
{
    XmlDocument xdoc = new XmlDocument();
    rss rss = null;
    try
    {
        // Quick check
        if ( this.Url.Length == 0 ) throw new Exception("A "
            + "zero-length RSS URL was requested...this is an "
            + "invalid condition and is a program bug "
            + "(this means YOU!)");

        // Generate the request (note the feed URL is assumed
        // to be loaded into the our Url property...).
        HttpRequest request =
            (HttpRequest)WebRequest.Create(this.Url);

        // Pull the response
        WebResponse response = request.GetResponse();

        // Contain within an XML document (in case of error)
        xdoc.Load(response.GetResponseStream());
        MemoryStream mstrm = new MemoryStream();
        xdoc.Save(mstrm);

        // Deserialize
        XmlSerializer ser = new XmlSerializer(typeof(rss));
        mstrm.Position = 0;
        rss = (rss)ser.Deserialize(mstrm);
        if ( rss == null ) throw new NullReferenceException("No"
            + " RSS data");
    } // try
    catch (InvalidOperationException)
```

```
{
    // Deserialization error... This happens when there are
    // invalid
    // XML characters in the stream. The error usually looks
    // like:
    //
    // "There is an error in XML document (1, 339)."
    //
    // I actually saw this from a Microsoft feed:
    //
    // http://msdn.microsoft.com/vcsharp/rss.xml
    //
    // They'd put "Whidbey" in quotes, but instead of simple
    // quotes, they used Microsoft Word's "smart quotes!"
    // Blew up deserialization big-time. :( I've seen a lot
    // of other garbage characters in most of the XML
    // streams I tried to connect to, so deserialization by
    // hand (reluctantly) at this point is necessary.
    //
    rss = RssDeserializer.Deserialize(xdoc.DocumentElement);
} // catch
catch (Exception ex)
{
    // Some error...we'll simply display it
    MessageBox.Show(ex.Message,
        "WinGator RSS Feed Retrieval Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
} // catch

return rss;
}
```

Figure 7. Synchronous GetFeed() method

Obtaining the RSS XML really boils down to a handful of lines of code:

```
HttpWebRequest request =
    (HttpWebRequest)WebRequest.Create(this.Url);
WebResponse response = request.GetResponse();
(stream) = response.GetResponseStream();
```

Given the response stream, you can do anything with the incoming XML. Ultimately I hand it to the .NET XML Serializer, but you could do other things as well. It's entirely up to you.

The asynchronous call itself isn't terribly different than the synchronous call. The primary difference lies with the callback function. With a synchronous Web data call, when you finally get the response stream, it's the entire response stream. With the asynchronous Web data call, you have something that's more like a garden hose. You've unrolled the hose and connected it to the spigot, but no water is yet flowing. You have to turn on the water. Bringing this back to .NET terms, you have to initiate read actions against the response stream. If during your reads you find there is more data to be read, you'll need to read more data until you're finished. This definitely complicates things, and I must admit I lifted this code nearly verbatim from the MSDN help file for asynchronous web methods calls. (There is a very much more complete discussion of asynchronous calls there, although I'll touch on the main points here.)

The asynchronous call begins with the `BeginGetFeed()` method, as you see in Figure 8.

```
public void BeginGetFeed()
{
    try
    {
        // Quick check
        if ( this.Url.Length == 0 ) throw new Exception("A "
            + "zero-length RSS URL was requested...this is an "
            + "invalid condition and is a program bug "
            + "(this means YOU!)");

        // Generate the request (note the feed URL is assumed
        // to be loaded into the our Url property...).
        HttpRequest request =
            (HttpRequest)WebRequest.Create(this.Url);

        // Create a state object to be passed
        // between asynchronous calls
        RequestState rs = new RequestState();
        rs.Request = request;

        // Make the asynchronous call
        request.BeginGetResponse(
            new AsyncCallback(this.responseCallback),rs);
    } // try
    catch (Exception ex)
    {
        // Some error...we'll simply display it
        MessageBox.Show(ex.Message,
            "WinGator RSS Feed Retrieval Error",
            MessageBoxButtons.OK,MessageBoxIcon.Error);
    } // catch
}
```

Figure 8. Initiating an asynchronous RSS feed data call.

As with the synchronous Web call, I create an instance of the `HttpRequest` object, but instead of making a direct request, I also create a request state container object and hand that to the asynchronous call initiation, `HttpRequest.BeginGetResponse()`. The request state container is merely a class that contains the information derived between stream reads, assuming I didn't get all of the information in the first Web request.

When the initial chunk of information is returned from the remote Web site, the .NET Framework will notify me using the method I passed into the `BeginGetResponse()` method, which is a method named `responseCallback()`. This callback function is an intermediate stop on my data request journey. What I have when this method is executed is a valid connection with the remote host, and using that connection I can begin reading data. But I've not actually read any data...I've just made a connection. You can see this in Figure 9.

```
private void responseCallback(IAsyncResult ar)
{
    try
    {
        // Get the RequestState object from the async result.
        RequestState rs = (RequestState)ar.AsyncState;

        // Get the WebRequest from RequestState.
        WebRequest req = rs.Request;

        // Call EndGetResponse, which produces the WebResponse
        object
        // that came from the request issued above.
        WebResponse resp = req.EndGetResponse(ar);

        // Start reading data from the response stream.
        Stream ResponseStream = resp.GetResponseStream();

        // Store the response stream in RequestState to read
        // the stream asynchronously.
        rs.ResponseStream = ResponseStream;

        // Pass rs.BufferRead to BeginRead.
        // Read data into rs.BufferRead
        IAsyncResult iarRead =
            ResponseStream.BeginRead(rs.BufferRead, 0,
            RequestState.BufferSize,
            new AsyncCallback(readCallback), rs);
    } // try
    catch (Exception ex)
    {
```

```
// Some error...we'll simply display it
MessageBox.Show(ex.Message,
    "WinGator RSS Feed Retrieval Error",
    MessageBoxButtons.OK,MessageBoxIcon.Error);
} // catch
}
```

Figure 9. Response stream data read initiation

Looking at Figure 9, you can see that I first retrieve the request state object passed into the call initiation, and using that, pull the request and then the response objects. With the response stream in hand, I begin the asynchronous read using `Stream.BeginRead()`, but of course, this requires yet another callback method I've named `readCallback()` (which you see in Figure 10).

```
private void readCallback(IAsyncResult ar)
{
    Stream responseStream = null;
    try
    {
        // Get the RequestState object from AsyncResult.
        RequestState rs = (RequestState)ar.AsyncState;

        // Retrieve the ResponseStream that was set in
        // responseCallback.
        responseStream = rs.ResponseStream;

        // Read rs.BufferRead to verify that it contains data.
        UTF8Encoding enc = new UTF8Encoding();
        Int32 bytesRead = responseStream.EndRead(ar);
        if ( bytesRead > 0 )
        {
            // Prepare a char array buffer for converting to
            Unicode.
            char[] charBuffer = new
            char[RequestState.BufferSize];

            // Convert byte stream to Unicode char array
            rs.StreamDecode.GetChars(rs.BufferRead, 0,
            bytesRead,
            charBuffer, 0);

            // Append the recently read data to the
            // RequestData stringbuilder
            // object contained in RequestState.
            rs.RequestData.Append(enc.GetString(rs.BufferRead,
            0, bytesRead));
        }
    }
}
```

```
// Continue reading data until
// responseStream.EndRead returns -1.
responseStream.BeginRead(rs.BufferRead, 0,
    RequestState.BufferSize,
    new AsyncCallback(readCallback), rs);
} // if
else
{
    // Close down the response stream.
    responseStream.Close();

    // Put the received data into a stream for
    // XML deserialization...
    byte[] buffer =
enc.GetBytes(rs.RequestData.ToString());
    MemoryStream mstrm = new MemoryStream(buffer);

    // Deserialize
    rss rss = null;
    try
    {
        XmlSerializer ser = new XmlSerializer(typeof(rss));
        rss = (rss)ser.Deserialize(mstrm);

        // Fire event
        if ( RssResponse != null )
        {
            RssResponse(this,
                new RssResponseEventArgs(rss));
        } // if
    } // try
    catch ( InvalidOperationException )
    {
        // Deserialization error... This happens when there
are
        // invalid XML characters in the stream. The error
        // usually looks like:
        //
        // "There is an error in XML document (1, 339)."
        //
        // I actually saw this from a Microsoft feed:
        //
        // http://msdn.microsoft.com/vcsharp/rss.xml
        //
        // They'd put "Whidbey" in quotes, but instead of
        // simple quotes, they used Microsoft Word's
        // "smart quotes!" Blew up deserialization big-time.
```

```
// ( I've seen a lot of other garbage characters in
// most of the XML streams I tried to connect to, so
// deserialization by hand (reluctantly) at this
// point is necessary.
//
XmlDocument xdoc = new XmlDocument();
mstrm.Position = 0;
xdoc.Load(mstrm);
rss = RssDeserializer.Deserialize(
    xdoc.DocumentElement);

// Fire event
if ( RssResponse != null )
{
    RssResponse(this, new
RssResponseEventArgs(rss));
} // if
} // try
catch (Exception ex)
{
    // To conserve critical resources, we MUST close
    // the stream to release the TCP connection with
    // the remote host.
    if ( responseStream != null )
responseStream.Close();

    // Some error...we'll simply display it
    MessageBox.Show(ex.Message,
        "WinGator Feed Refresh Error",
        MessageBoxButtons.OK,MessageBoxIcon.Error);
} // catch
} // else
} // try
catch (Exception ex)
{
    // To conserve critical resources, we MUST close
    // the stream to release the TCP connection with
    // the remote host.
    if ( responseStream != null ) responseStream.Close();

    // Some error...we'll simply display it
    MessageBox.Show(ex.Message,
        "WinGator RSS Feed Retrieval Error",
        MessageBoxButtons.OK,MessageBoxIcon.Error);
} // catch
}
```

Figure 10. Reading the response stream data

The logic behind the read callback function is actually a bit simpler than it looks in Figure 10. The code is a bit more complicated because of the exception handling. You don't want to leave connections with the remote host left open, so there is code there to make sure the connection is in fact closed upon exception. Ignoring the exception handling for the moment, I'm following this simple logic:

- If there is data to be read, read it (up to a specific number of bytes)
- If there is no more data to be read, close the connection and parse the resultant XML
- If there is an XML Serializer exception while parsing the XML, try to deserialize by hand

If there is data to be read, I initiate another asynchronous read:

```
if ( { number of bytes we just read } > 0 )  
    responseStream.BeginRead(...);
```

To do this, I'll pass into the asynchronous call the same request state object I've been using. Inside this object is a text buffer (a byte array) that I'm filling with data each time I read from the incoming response stream. The byte array is then copied into a `StringBuilder` object for later deserialization.

If I tried to read some data but found no more, I close the connection, take the read data and place it into a stream (for deserialization), and then initiate the XML deserialization:

```
responseStream.Close();  
  
byte[] buffer = enc.GetBytes(rs.RequestData.ToString());  
MemoryStream mstrm = new MemoryStream(buffer);  
  
XmlSerializer ser = new XmlSerializer(typeof(rss));  
rss = (rss)ser.Deserialize(mstrm);
```

If the XML Serializer determines there are invalid characters in the response (it did this quite often in my tests), I'll deserialize by hand using a simple parser I created for the job. I could skip the .NET XML Serializer altogether at this point, but I left it in for demonstration purposes. Only rarely does the .NET Framework let me down, but this is one of those times. You can't predict what information might be embedded within the XML stream you're deserializing, so it would have been nice had Microsoft written in a cancellable invalid character event that would allow you to remove/replace the invalid XML character and continue the (de)serialization process. Sadly, no, so I catch the exception that the XML Serializer object throws when it reads a character it cannot interpret, and read the entire stream over again using a somewhat looser interpretation of the rules of XML formatting.

In any case, with luck, I now have a valid managed rss object that contains the channels and items available from the selected RSS data feed. In this case, I'll need to notify the main application of the pending information, and I created an event just for this, FeedGrabber.RssResponse. The main application hooks into this event and takes actions based upon new or updated incoming feed data.

Isolated Storage

With most applications of any complexity, you often find you'll need to store persistent information between application invocations, and WinGator is no different. I persist the application's previous screen location information, so that the next time it's executed, the user interface is displayed onscreen in the same location as when it was last executed. Of more interest is the RSS data feed caching that must take place.

While studying some of the available RSS readers, it was clear that they cached the feed information in memory. Frankly, it's easier to do so, and given that most people have gobs of memory on their systems, it isn't necessarily a bad way to go. But some of the readers were consuming upwards of 50MB of RAM for their cache information, which seems excessive. When I design applications such as this, I'll choose a RAM-based cache if I need speedy access. Here, though, this type of data access speed isn't necessarily required--certainly not in a demonstration application and most likely not at all. So I elected to be different and implement caching on disk instead of in memory. It'll take more time to pull the cache for a given feed from the cache file, but since updating the cache is an Internet-based activity anyway (translation: SLOW as compared to local memory access), there was no reason to consume huge amounts of the user's available memory. The user interface effect of this when simply accessing the cache is totally unnoticeable, at least to my eye.

Whether you agree with that reasoning or not, I still had to decide where to store the cache information. Perhaps I could stuff it into the application's executable directory? Or how about storing the cache information in the temporary files directory on the user's machine? I could even create my own cache directory and go from there. But I chose to store the files in the user's isolated storage.

What's isolated storage? I actually got the suggestion from Angie, and this is what she said. "Isolated storage? Oh, well, that's where user applications store application-specific information. You know, My Documents and such? Only in this case it's usually in a special directory under the user's Documents and Settings subdirectory." Oh, okay. So .NET provides a location-agnostic place to store application-specific information. Well, that would mean that .NET must provide an API to access this information as well, and in fact .NET does.

Isolated storage is the preferred location for storing application-specific information. There are two primary reasons for this. First, this is the area system administrators routinely backup, so if the user loses a hard drive or something, they can normally retrieve their application information. Second, if the user takes advantage of roving account information, as in an environment where they might log into one of several machines, isolated storage information is available to any of those machines, making the application information viewable (and usable) over a wide variety of network configurations. If you stored the information in the application's directory, you'd certainly lose the roving capability and you'd probably also lose the backup benefits as well. Storing the cache information in the temporary folder simply doesn't apply since the cache is presumably not intended to be temporary.

Of course, the cache could conceivably grow to be quite large if the user opens and interprets many data feeds, so storing the actual cached information in the Registry should also be avoided. While portions of the Registry do travel with the user in a roving situation and the Registry (as a file) is certainly backed up by system administrators, the size of the Registry is often capped. Should the cached information grow to such a size that the Registry limits are met or exceeded, your user is in a world of hurt.

The most appropriate and logical place to store the feed cache is in the user's isolated storage. While isolated storage could be anywhere, you'll typically find it in the Documents and Settings subdirectory for your user. On my system, using WinGator, this would be:

In this oddly named directory, I have several files: UIState.config, FeedCache.config, and several files with GUIDs for a filename with the extension .xml. The UIState.config file maintains the persisted state of the user interface--screen location, size, minimized/maximized state, and so forth. When the application is terminated and then re-executed, WinGator reads this file and sizes/places its user interface accordingly.

FeedCache.config is the main RSS feed cache file, and inside this file you'll find XML nodes for each known feed that associates several pieces of information (as seen in Figure 11).

```
<?xml version="1.0" encoding="utf-8"?>
<FeedCache refreshRate="60">
  <Feed>
    <feedName>MSDN: Visual Studio</feedName>
    <feedPath>79b05f61-34de-4028-8e50-
e204feee9fd1.xml</feedPath>
    <feedUrl>http://msdn.microsoft.com/vstudio/rss.xml</feedUrl>
  </Feed>
  {More/other feeds here...}
</FeedCache>
```

Figure 11. WinGator RSS feed cache information

Basically, in the main cache file I store the feed name (for quick lookup and search), the name of the actual RSS XML information (with a Guid as a filename), and the URL WinGator should poll for additional/new RSS data. You'd think I would consult this file often, but in fact the same information is also cached in memory. This file is simply where the feed information is stored when WinGator isn't active.

If isolated storage is in different locations for different users on different operating systems, how do you know where it is found, and how do you read/write to it? The answer to the first question is you don't need to know this--Windows and .NET manage this for you. They manage it through the second question's response, which is through a special family of classes you'll find in System.IO.IsolatedStorage.

Using isolated storage is actually quite easy. The basic pattern is:

1. Open the root isolated storage for the user
2. Open an isolated storage stream for reading/writing data
3. Use the stream (basically read/write to a file)
4. Close the stream

When the root isolated storage file is open, you are free to create and delete files and/or subdirectories as necessary. You can iterate the known files and/or directories easily as well. Accessing the feed cache file looks much like the code you see in Figure 12.

```
IsolatedStorageFileStream isoStream = null;
try
{
    // Open the isolated storage stream for the feed cache file
    IsolatedStorageFile isoStore =
        IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly |
        IsolatedStorageScope.Domain, null, null);
    isoStream = new
    IsolatedStorageFileStream("FeedCache.config",
        FileMode.Open, isoStore);

    // Load the contents of the cache
    . . . (do something with the file...)
} // try
catch (Exception ex)
{
    // Exception!
    . . . (take any action necessary...)
} // catch
finally
{
    // Close the stream
```

```
if ( isoStream != null ) isoStream.Close();  
} // finally
```

Figure 12. Accessing WinGator RSS feed cache information

The secret is to call `IsolatedStorageFile.GetStore()` with the desired scope and use the same scope throughout your application. To support this, WinGator actually declares an `IsolatedStorageScope` variable and uses that throughout the application, but the gist is the same as you see in Figure 12. The scope determines how the isolated storage will be accessed, and even indirectly establishes the location on the user's local hard drive. In this case, I'm telling .NET to open an isolated storage file for this user, this assembly, and this application domain (an option I didn't include was the roaming option, but this is easily added). What's nice about this is that if the version of the application changes, the assembly identification changes, and you access a different isolated storage location. This allows you to be essentially unconcerned about reading earlier application version data files. Of course, during development, this is something to watch for as the version numbers of your assemblies are probably changing (due to auto-incrementing build numbers if nothing else).

Depending upon what WinGator needs to do, the isolated storage files/streams are opened read-only, create, or open with truncate. In every case, though, you'll find a `finally` block inserted into the code to be sure that the isolated storage is properly closed once you're through reading/writing to the individual file.

I won't go over the work WinGator does with the feed information since the data is essentially stuffed into a cache file and persisted on disk. Angie needs the booth, we only have an hour for lunch, and there's a bit more to cover. Remember, though, that in the `FeedCache.cs` file you'll find routines that scrub incoming feed data against the cache and update the information accordingly, search for feed information (presumably in response to a user input of some kind), and load/save the main cache file.

Dynamic HTML

The last interesting area I'd like to talk about is dynamic HTML, or DHTML, and how WinGator uses it to generate a browser interface on the fly. Since I'm using .NET, I'm assured that the user has Internet Explorer 6.0, and since the incoming RSS data includes an URL I'll need to access and display, it makes sense to employ the browser control in the RSS application.

In WinGator's case, I use the browser control in two places: the main news URL display (see Figure 3) and for displaying "rollup" or summary information about the news item (see Figure 2). The main news URL is a simple navigation activity and therefore not worthy of DHTML consideration. Once the user selects a news feed from the tree control, you know the URL for the news item, so you simply hand that to the browser and tell it to navigate to that URL.

The rollup information is more interesting from a DHTML standpoint. I should point out that using the browser in this case is but one way to solve the problem (the "problem" being how to generate a user interface that clearly tells the user how to access the individual news item). I could have used a link button and a separate Windows Forms panel. Once the user clicked the link button, I could accept that event and do the very same thing I'm doing with WinGator now, which is to activate the second tab and display the news item. But I elected to use the browser control and insert some HTML instead, if for no other reason than to show how it could be done.

To use DHTML in your Windows Forms applications, you'll need to do a couple of things. Clearly you'll need to have an instance of the browser control embedded somewhere in your application. But the nonobvious thing is you'll need to reference `Microsoft.mshtml`. The basic browser control allows you to gain access to the raw HTML document, but it doesn't expose the DHTML interfaces so that you can do anything with the document. `Microsoft.mshtml` provides those interfaces and will give you the necessary access to the HTML document you'll require to insert or modify the contained HTML.

With an instance of the browser embedded in your application and `Microsoft.mshtml` referenced, you can insert new HTML using the code you see in Figure 13.

```
IHTMLDocument2 doc2 =  
(IHTMLDocument2)mybrowser.Document;  
IHTMLElement body = doc2.body;  
body.innerHTML = "<html><body>Hello,  
World!</body></html>";
```

Figure 13. Providing the browser control DHTML information

The HTML WinGator uses is slightly more complex, but not greatly so. One difference is WinGator inserts an anchor element the user can click to display the news item. The anchor's URL is the actual URL of the news item, which brings up an interesting situation. If the user clicks the anchor, the browser will navigate to the URL specified. But from a user interface perspective, it's the wrong browser instance! Rather, you must intercept the navigation request, cancel that, and force the second browser instance to navigate to the news item information (taking time to activate the second tab in the tab control along the way). If you didn't do this, the second browser (the one intended for viewing news items) would never navigate anywhere--you'd be reading the news item in the (very much) smaller details view.

You cannot place new HTML into the browser's body object without first having navigated to somewhere first. If you try to modify the `IHTMLElement.innerHTML` property without first navigating, the browser will throw an exception. Therefore, when the application is initialized, WinGator navigates to an internal, well-known URL: `about:blank`. The code that performs this initialization is shown in Figure 14.

```
// Navigate the browsers...
object flags = new object();
object frame = new object();
object data = new object();
object headers = new object();
brsItemDetail.Navigate("about:blank",ref flags,ref frame,ref data
,ref headers);
brsItemData.Navigate("about:blank",ref flags,ref frame,ref data
,ref headers);
```

Figure 14. Pre-navigating (initializing) the browser control

Final Thoughts

After hearing Angie talk about RSS, and after seeing how simple the XML really is, I was amazed by how complex WinGator turned out. Don't let the relative lack of data complexity lure you into believing that the associated application must also be simple. There was a lot to consider with WinGator--asynchronous Web data access, local data caching, DHTML and browser navigation, and so forth. It wasn't necessarily a simple application to write. But Angie was impressed, which is always gratifying. [Click here](#) to download the source code for WinGator.

Angie hates getting spammed, so I don't give out her e-mail. If you have questions or comments about RSS or WinGator, I'm always happy to address them. Just drop me a note at kenns@wintellect.com and we'll see what we can't come up with. Until next time, happy coding from the CodeCafe!