DATA POINTS

Syndicated Data And Isolated Storage In Silverlight

John Papa

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

Contents

Silverlight is ideal for building a syndicated news reader application. It can read RSS and AtomPub syndicated formats, communicate with Web services via HTTP requests, and handle cross-domain policies. Once the syndicated data has been received, it can be read into a class structure, parsed with LINQ, and presented to the user via XAML-based data binding.

In this column I will demonstrate how to utilize these unique features to build a syndicated news reader. I will also show you how to debug problems with Web service communications from Silverlight and how to store data locally using isolated storage. All of the examples are available in the MSDN code download.

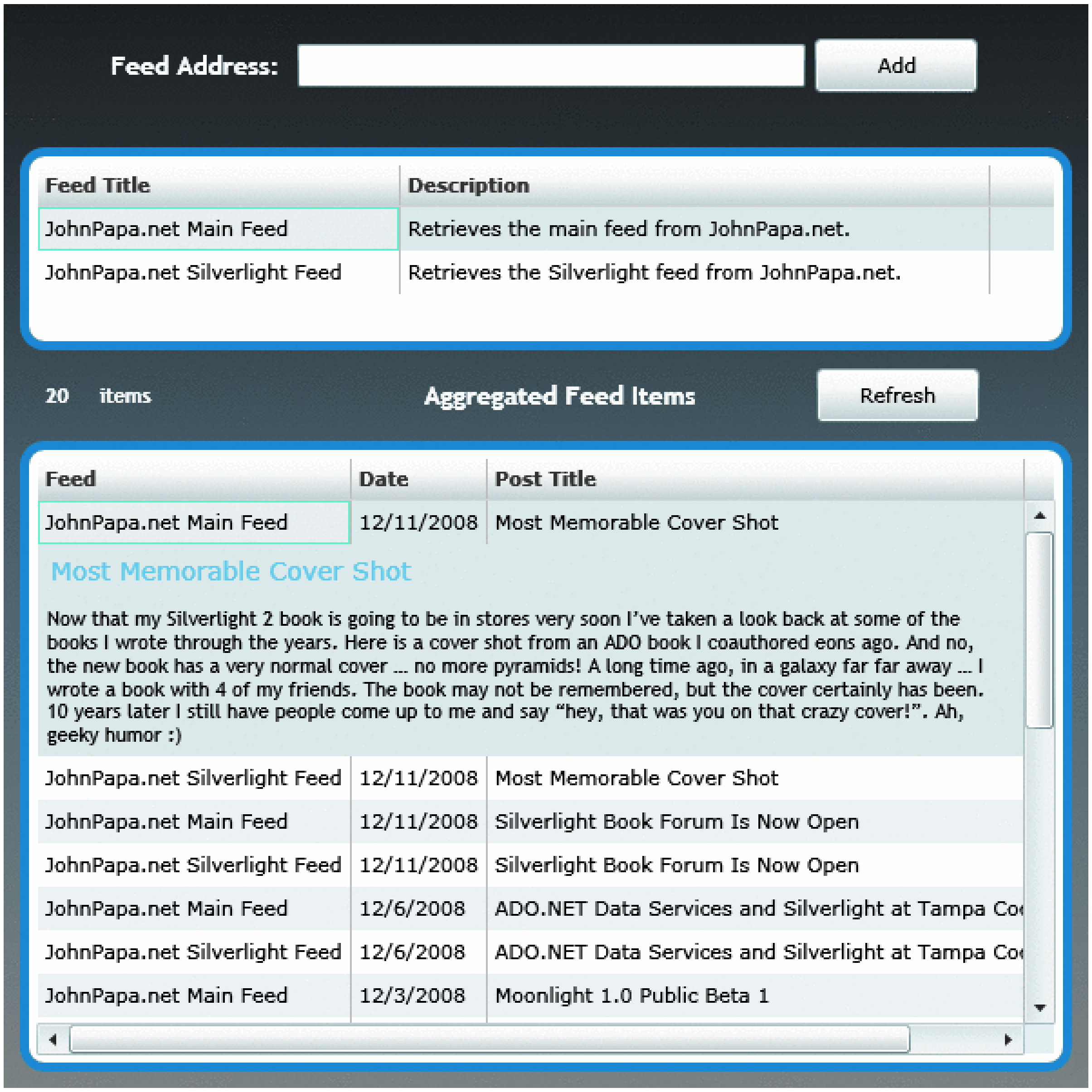
Getting Finished

Feed syndication provides access to the RSS and AtomPub formats through Web services. Each feed is accessed through a Uri that returns XML containing the feed items using the RSS or AtomPub formats. The sample application provided with this column demonstrates techniques that read syndicated data using Web service requests into a Silverlight application. Before diving into the details, it might be helpful if I show you the finished version of the application. Then I will discuss the logic and various other aspects of the application's code.

Figure 1 shows the Silverlight application reading these two syndicated feeds:

Copy Code

```
 http://pipes.yahooapis.com/pipes/pipe.run?_id=957
    d9624940693fb9f9644d7b12fb0e9&_render=rss
http://pipes.yahooapis.com/pipes/pipe.run?_id=057559bac7aad6640b
    c17529f3421db0&_render=rss
```



Feed Title	Description
JohnPapa.net Main Feed	Retrieves the main feed from JohnPapa.net.
JohnPapa.net Silverlight Feed	Retrieves the Silverlight feed from JohnPapa.net.

Feed	Date	Post Title
JohnPapa.net Main Feed	12/11/2008	Most Memorable Cover Shot

Most Memorable Cover Shot

Now that my Silverlight 2 book is going to be in stores very soon I've taken a look back at some of the books I wrote through the years. Here is a cover shot from an ADO book I coauthored eons ago. And no, the new book has a very normal cover … no more pyramids! A long time ago, in a galaxy far far away … I wrote a book with 4 of my friends. The book may not be remembered, but the cover certainly has been. 10 years later I still have people come up to me and say "hey, that was you on that crazy cover!". Ah, geeky humor :)

JohnPapa.net Silverlight Feed	12/11/2008	Most Memorable Cover Shot
JohnPapa.net Main Feed	12/11/2008	Silverlight Book Forum Is Now Open
JohnPapa.net Silverlight Feed	12/11/2008	Silverlight Book Forum Is Now Open
JohnPapa.net Main Feed	12/6/2008	ADO.NET Data Services and Silverlight at Tampa Co
JohnPapa.net Silverlight Feed	12/6/2008	ADO.NET Data Services and Silverlight at Tampa Co
JohnPapa.net Main Feed	12/3/2008	Moonlight 1.0 Public Beta 1

Figure 1 **The SilverlightSyndication Application**

A Web request is made to gather the feed data when the user clicks the Add button. The feed's Uri is stored locally on the client computer, and the title of the feed service is displayed in the upper DataGrid control. Storing the feed's Uri on the client computer allows the Silverlight application to get the feed data for those feeds when the application starts. When the feed data is gathered, the items are placed into a list of objects and then parsed using LINQ. The results are then bound to the lower DataGrid control in sorted order.

HTTP Web Requests and Threading

Consuming syndicated feeds from Silverlight applications starts with being able to make Web requests. Both the WebClient and the HttpWebRequest classes can make HTTP Web requests from Silverlight applications. The first step in the process is deciding which technique to use to make HTTP Web requests. In most cases, the WebClient class is all that is needed since it is simpler to use (it also calls the HttpWebRequest under the covers). HttpWebRequest allows for more customization of the requests.

This code snippet demonstrates how to make a request through WebClient:

Copy Code

```
//feedUri is of type Uri
```

```
WebClient request = new WebClient();
request.DownloadStringCompleted += AddFeedCompleted;
request.DownloadStringAsync(feedUri);
```
The following code, on the other hand, shows how to make a similar call using HttpWebRequest:
Copy Code
```
//feedUri is of type Uri
WebRequest request = HttpWebRequest.Create(feedUri);
request.BeginGetResponse(new AsyncCallback(ReadCallback), request);
```
The WebClient class is easier to use because it wraps some of the functionality of the HttpWebRequest class. Since all HTTP Web requests from Silverlight are made asynchronously through both WebClient and HttpWebRequest, it is important to understand how to process the data when the calls return. When HttpWebRequest makes its asynchronous calls and it completes, the completed event handler is not guaranteed to be operating on the UI thread. If data were retrieved and meant to be displayed in a UI element, the call to the UI element must be made using a technique that transfers control back to the UI thread from the background thread. This can be done with the Dispatcher or through the SynchronizationContext class. The following code shows how to make a call using the UI thread with the Dispatcher class's BeginInvoke method:
Copy Code
```
Deployment.Current.Dispatcher.BeginInvoke(() =>
     {
         MyDataGrid.DataContext = productList;
     });
```
This code example takes the productList variable (which was filled from the data returned from a Web service call) and sets it to a UI element's DataContext. In this case, a DataGrid will now be bound to the list of products. The Dispatcher is not needed, however, if the call was made through the WebClient class. In that case, the code could simply set the product list directly to the UI element's DataContext.
To use the SynchronizationContext class, an instance of the SynchronizationContext class must be created in a place where the UI thread is known to be available. The constructor and the load event handler are good places to create the instance of a SynchronizationContext. This code sample shows the _syncContext field being initialized in the class constructor:
Copy Code
```
public Page() {
_syncContext = SynchronizationContext.Current;
}
```
And this code shows the SynchronizationContext instance using its Post method to make the call to the LoadProducts method. It makes sure that the LoadProducts method has access to the UI thread:
Copy Code
```
if (_syncContext != null) {
  _syncContext.Post(delegate(object state){ LoadProducts(products); }
  ,null);
}
```
Besides being easier to use, WebClient requests always come back on the UI thread. This means that any results from the WebClient's request can be easily bound to the UI elements without having to involve the Dispatcher (or, optionally, the SynchronizationContext class instead of the Dispatcher). For reading syndicated data, the WebClient class is adequate and will be used for this column's sample application.

Adding a Feed

In the sample application, when the user enters a feed address and clicks the Add button, the code shown in **Figure 2** executes. First the code attempts to create a Uri from the address, using the Uri.TryCreate method if possible. If it can create a Uri, the Uri is returned to the local variable feedUri. Otherwise, feedUri remains null and the code exits.
Figure 2 Adding a Feed
Copy Code
```
private void btnAdd_Click(object sender, RoutedEventArgs e)
{
    Uri feedUri;
    Uri.TryCreate(txtAddress.Text, UriKind.Absolute, out feedUri);
    if (feedUri == null)
        return;

    LoadFeed(feedUri);
}

public void LoadFeed(Uri feedUri)
{
    WebClient request = new WebClient();
    request.DownloadStringCompleted += AddFeedCompleted;
    request.DownloadStringAsync(feedUri);
}
```

Once a valid Uri is created, the LoadFeed method executes, making the HTTP request to gather the feed data using the WebClient class. The WebClient instance is created, and an event handler is assigned to the DownloadStringCompleted event. When the DownloadStringAsync method is executed and is ready to return its data, it needs to know what event handler to go to. That is why the event handler (in this case, AddFeedCompleted) must be assigned before the asynchronous event is executed.

Once the request has completed, the AddFeedCompleted event handler will execute (see **Figure 3**). The DownloadStringCompletedEventArgs parameter contains a Result property and an Error property, both of which are important to check after each Web request. The e.Error property will be null if there were no errors during the request. The e.Result property contains the results of the Web request. For the sample application, e.Result will contain the XML representing the feed data.

Figure 3 AddFeedCompleted

Copy Code

```
private void AddFeedCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    if (e.Error != null)
        return;
    string xml = e.Result;
    if (xml.Length == 0)
        return;
    StringReader stringReader = new StringReader(xml);
    XmlReader reader = XmlReader.Create(stringReader);
    SyndicationFeed feed = SyndicationFeed.Load(reader);
    if (_feeds.Where(f => f.Title.Text == feed.Title.Text).ToList().Count > 0)
        return;
    _feeds.Add(feed); // This also saves the feeds to isolated storage
    ReBindAggregatedItems();
    txtAddress.Text = string.Empty;
}
```

Once the feed data is gathered, it can be read into the System.ServiceModel.SyndicationFeed class using the SyndicationFeed class's Load method. Note that when retrieving feed data and using it in a read-only manner, using LINQ to XML to retrieve the feed and loading it in a custom object may be a better option than SyndicationFeed. SyndicationFeed has more features, but if they are not being used it may not be worth the additional size added to the XAP—SyndicationFeed adds about 150KB to the XAP while LINQ to XML adds about 40KB. With the additional power of SyndicationFeed you also have some cost in size.

SyndicationFeed is a special class that knows how to represent feed data (both RSS and AtomPub) as an object. It has properties that describe the feed itself, such as Title and Description, as well as an Items property that contains an IEnumerable<SyndicationItem>. Each SyndicationItem class instance represents a feed item for the feed. For example, the feeds are represented by instances of the SyndicationFeed class, and their Items collections contain the individual posts from the feeds.

Once the SyndicationFeed class is loaded with the feed and its items, the code shown in **Figure 3** checks to see if the same feed has already been gathered. If so, the code exits immediately. Otherwise, the feed is added to the local ObservableCollection<SyndicationFeed> called _feeds. Through the ReBindAggregatedItems method, the feed items from all of the loaded feeds are then filtered, sorted, and bound to the lower DataGrid. Since the WebClient class made the HTTP Web request, the AddFeedCompleted event handler will have access to the UI thread. This is why the code inside the ReBindAggregatedItems method can bind the data to the DataGrid UI element without the Dispatcher's help.

Parsing Feeds

When the ReBindAggregatedItems method executes, the feed data is stored in a collection of SyndicatedFeed instances and their respective collections of SyndicatedItem instances. LINQ is ideal for querying the feed data since it is now in an object structure. The data did not need to be loaded into SyndicatedFeed objects. Instead it could have been kept in its native XML format (as RSS or AtomPub), and it could have been parsed using an XmlReader or LINQ to XML. However, the SyndicatedFeed class makes it easier to manage, and LINQ can still be used to query the data.

Displaying the feed items for several feeds requires that the feed items are all mashed together. The LINQ query shown in **Figure 4** demonstrates how to grab all of the feed items (SyndicationItem instances) for all of the feeds (SyndicationFeed instances) and sort them by their publication date.

Figure 4 Querying Feeds with LINQ

Copy Code

```
private void ReBindAggregatedItems()
{
    //Read the feed items and bind them to the lower list
    var query = from f in _feeds
                from i in f.Items
                orderby i.PublishDate descending
                select new SyndicationItemExtra
                        { FeedTitle = f.Title.Text, Item = i };

    var items = query.ToList();
    feedItemsGridLayout.DataContext = items;
}
```

Notice in **Figure 4** that the query returns a list of SyndicationItemExtra classes. The SyndicationItemExtra class is a custom class that has FeedTitle property of type string and an Item property of type SyndicationItem. The application displays the feed items in the DataGrid, and most of the data for this can be found in the SyndicationItem class.

However, since the application mashes together items from several feeds, displaying the title of the feed for each feed item makes it clear which feed each item is from. The title for the feed is not accessible from the SyndicationItem class, so the application uses a custom class called SyndicationItemExtra, which will store the SyndicationItem and the feed's title.

The feed items are then bound to the Grid panel feedItemsGridLayout in the Silverlight application. The Grid panel contains the DataGrid as well as other UI elements (such as the number of items displayed in a TextBlock) that are involved in data-binding operations to display information about the feed items.

Cross-Domain Requests for Feeds

Requests to gather feeds are HTTP Web requests that generally make requests to a different Web domain. Any Web request from Silverlight that communicates with a different domain than the one that hosts the Silverlight application must comply with the remote domain's cross-domain policy. The diagram in **Figure 5** demonstrates this situation.

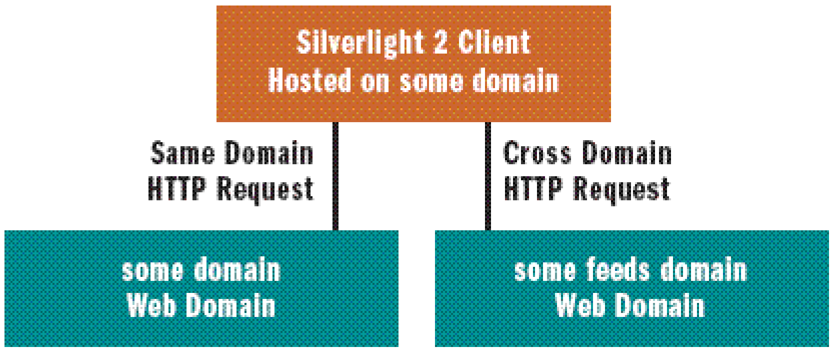
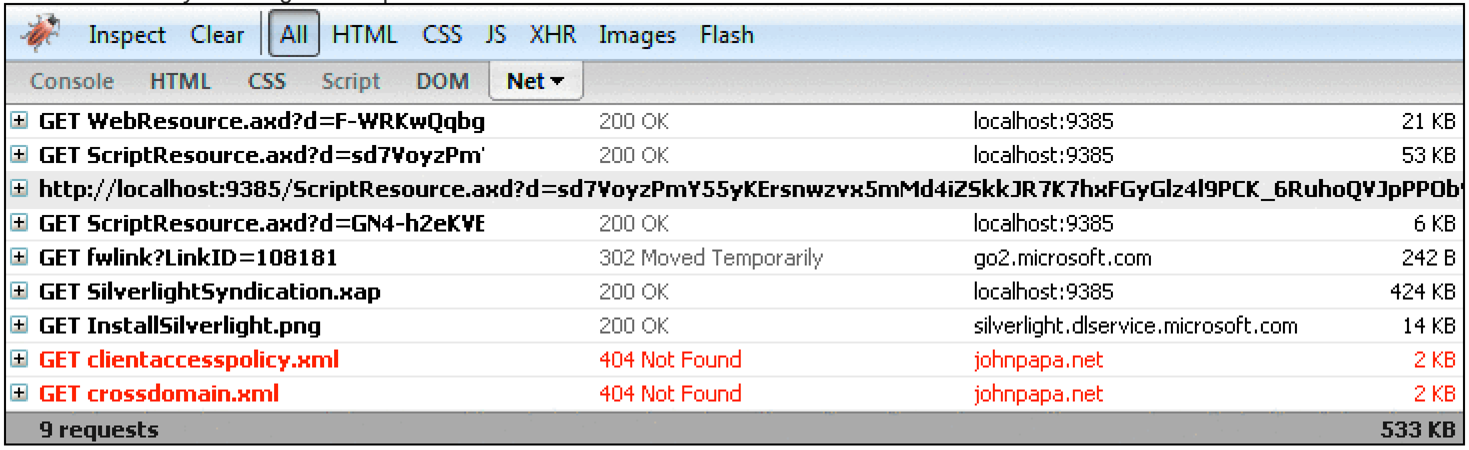


Figure 5 **Cross-Domain Call for a Feed**

For more on cross-domain policies, please refer to my September 2008 Data Points column. In that column I discuss the file formats and how the policies work.

When an HTTP Web request is made across domains, Silverlight preempts the request by first requesting the cross-domain policy file from the remote Web server. Silverlight first looks for the clientaccesspolicy.xml file (the Silverlight cross-domain policy file), and if it is not found, it then looks for the crossdomain.xml file (the Flash cross-domain policy file). If neither file is found, the request fails and an error is thrown. This error can be caught in the DownloadStringCompleted event handler and presented to the user, if desired.

For example, if the Uri http://johnpapa.net/feed/default.aspx is entered into the sample application, Silverlight will first look for one of the cross-domain policy files on the johnpapa.net Web server's root. If neither of the files are found, then an error is returned to the application, at which point the application can notify the user if desired. **Figure 6** shows the FireBug plug-in, which is tracking all requests from the browser. It shows the browser looking for the cross-domain policy files, not finding them, and returning without actually making the request for the RSS feed.



```
⊞ GET WebResource.axd?d=F-WRKwQqbg      200 OK                  localhost:9385                     21 KB
⊞ GET ScriptResource.axd?d=sd7VoyzPm'   200 OK                  localhost:9385                     53 KB
⊞ http://localhost:9385/ScriptResource.axd?d=sd7VoyzPmY55yKErsnwzvx5mMd4iZSkkJR7K7hxFGyGlz4l9PCK_6RuhoQVJpPPOb
⊞ GET ScriptResource.axd?d=GN4-h2eKVE   200 OK                  localhost:9385                      6 KB
⊞ GET fwlink?LinkID=108181              302 Moved Temporarily   go2.microsoft.com                  242 B
⊞ GET SilverlightSyndication.xap        200 OK                  localhost:9385                     424 KB
⊞ GET InstallSilverlight.png            200 OK                  silverlight.dlservice.microsoft.com 14 KB
⊞ GET clientaccesspolicy.xml            404 Not Found           johnpapa.net                        2 KB
⊞ GET crossdomain.xml                   404 Not Found           johnpapa.net                        2 KB
  9 requests                                                                                      533 KB
```

Figure 6 **Debugging Cross-Domain Feed Calls**

FireBug is a great tool for watching HTTP requests in Firefox, and Web Development Helper is a great tool when using Internet Explorer. Another option is Fiddler2, which is a standalone application that can watch all traffic on your computer.

One solution to this problem is to ask the Web administrator for the feed to put a clientaccesspolicy.xml file in the Web server's root. This may not be realistic, since you most likely do not control the remote Web server nor do you know who does. Another option is to see if the feed uses an intermediary service such as Yahoo Pipes. For example, the main feed at johnpapa.net can be retrieved through Yahoo Pipes using the Uri http://pipes.yahooapis.com/pipes/pipe.run?_id=057559bac7aad6640bc17529f3421db0&_render=rss. Since there is a cross-domain policy file located at http://pipes.yahooapis.com/clientaccesspolicy.xml that allows open access, this is a good alternative.

A third option is to use a service such as Popfly or FeedBurner to aggregate the feeds, effectively relaying them through a service that also has an open cross-domain policy. Finally, a fourth option would be to write your own custom Web service that gathers the feeds and then relays them to the Silverlight application. Using a service such as Popfly or Yahoo Pipes offers the simplest solutions.

Basic Isolated Storage

The sample app lets a user add several feeds and view all of the items for each of those feeds. If a user enters 10 feeds and decides that he needs to close the app and come back later to read them, he would likely expect that the feeds would be remembered by the application. Otherwise, he'd have to enter the Uri for each feed every time the app is opened. Since these feeds are specific to a user, they can be stored either on the server using some token to identify the user that entered them or on the user's computer. Silverlight allows data to be stored to a protected area of the user's computer using the classes in the System.IO.IsolatedStorage namespace. Silverlight Isolated Storage is like cookies on steroids: it allows you to store simple scalar values or even store serialized complex object graphs on the client computer. The simplest way to save to isolated storage is to create an ApplicationSettings entry and stuff your data in it, as shown here:

Copy Code

```
private void SaveFeedsToStorage_UsingSettings()
{
    string data = GetFeedsFromStorage_UsingSettings() + FEED_DELIMITER +
        txtAddress.Text;
    if (IsolatedStorageSettings.ApplicationSettings.Contains(FEED_DATA))
        IsolatedStorageSettings.ApplicationSettings[FEED_DATA] = data;
    else
        IsolatedStorageSettings.ApplicationSettings.Add(FEED_DATA, data);
}
```

This can be called every time a SyndicationFeed is added or removed from the Observ-ableCollection<SyndicationFeed> field instance called _feeds. Since the ObservableCollection exposes a CollectionChanged event, a handler can be assigned to the event that performs the save, as shown here:

Copy Code

```
_feeds.CollectionChanged += ((sender, e) => {
                               SaveFeedsToStorage_UsingSettings(); });
```

When the SaveFeedsToStorage_UsingSettings method is executed, it first calls the GetFeedsFromStorage_UsingSettings method, which grabs the addresses of all of the feeds from isolated storage and puts them in a single string delimited by a special character.

When the app first starts, the LoadFeedsFromStorage_UsingSettings method retrieves the feeds from isolated storage:

Copy Code

```
private void LoadFeedsFromStorage_UsingSettings()
{
    string data = LoadFeedsFromStorage_UsingSettings();
    string[] feedList = data.Split(new string[1] { FEED_DELIMITER },
      StringSplitOptions.RemoveEmptyEntries);
    foreach (var address in feedList)
        LoadFeed(new Uri(address));
}
```

The code first reads the list of Uri addresses for each feed from isolated storage. Then it iterates through the addresses and loads each individual feed one at a time, using the LoadFeed method.

Organized Isolated Storage

This feature allows the application to remember the feed addresses for the user and load them when the user runs the application. Packing Uri addresses into a delimited string is simple but neither elegant nor expansive. For example, if you wanted to store more than just a single scalar value, this would get complicated using this technique.

Another way to store data in isolated storage is to use the IsolatedStorageFile and IsolatedStorageFileStream classes, which let you store more complex data structures, including serialized objects, per user. The data can even be segmented into different files and folders in isolated storage. This is

ideal for organizing data that will be saved in isolated storage. For example, a folder could be created for all static lists of data, and a separate file for each list could be created. So within a folder in isolated storage, a file might exist for name prefixes, another for gender, and yet another for U.S. states.
The sample application could create a file in isolated storage to contain the list of Uri addresses. The data must first be serialized and then sent to the file in isolated storage (as shown in **Figure 7**). First, an instance of the IsolatedStorageFile class for the current user is created using the GetUserStoreForApplication method. Then a file stream is created so the application can write the Uri address. The data is then serialized and written to the IsolatedStorageFileStream instance. The example for this application serializes a string, but any serializable object can be written to isolated storage as well.

Figure 7 Saving Serialized Data to an Isolated Storage File

Copy Code
```
private void SaveFeedsToStorage_UsingFile() {
    using (var isoStore = IsolatedStorageFile.GetUserStoreForApplication())
{
        List<string> data = GetFeedsFromStorage_UsingFile();
        if (data == null)
            if (txtAddress.Text.Length == 0)
                return;
            else
                data = new List<string>();
         using (var isoStoreFileStream =
                new IsolatedStorageFileStream(FEED_FILENAME,
                   FileMode.Create, isoStore)) {
            data.Add(txtAddress.Text);
            byte[] bytes = Serialize(data);
            isoStoreFileStream.Write(bytes, 0, bytes.Length);
        }
    }
}
```
Reading the serialized data out of a file from isolated storage is a little more involved than the previous example. **Figure 8** shows that first you must get an instance of the IsolatedStorageFile class for the user, and then check if the file exists before you read it. If the file exists, the file is opened for read access, which allows the data to be read via a stream of type IsolatedStorageFileStream. The data is read from the stream, put together, and then deserialized so it can be used to load the syndicated feeds.

Figure 8 Reading Serialized Data from an Isolated Storage File

Copy Code
```
private List<string> GetFeedsFromStorage_UsingFile() {
    byte[] feedBytes;
    var ms = new MemoryStream();
    using (var isoStore =
      solatedStorageFile.GetUserStoreForApplication())
    {
        if (!isoStore.FileExists(FEED_FILENAME)) return null;
        using (var stream = isoStore.OpenFile(FEED_FILENAME,
          FileMode.Open, FileAccess.Read))  {
            while (true) {
                byte[] tempBytes = new byte[1024];
                int read = stream.Read(tempBytes, 0, tempBytes.Length);
                if (read <= 0) {
                    //feedBytes = ms.ToArray();
                    break;
                }
                ms.Write(tempBytes, 0, read);
            }
        }
        feedBytes = ms.ToArray();
        List<string> feedList = Deserialize(typeof(List<string>),
            feedBytes) as List<string>;
        return feedList;
    }
}

private void LoadFeedsFromStorage_UsingFile() {
    var feedList = GetFeedsFromStorage_UsingFile();
    foreach (var address in feedList) {
        Uri feedUri;
        Uri.TryCreate(address, UriKind.Absolute, out feedUri);
        if (feedUri != null)
            LoadFeed(feedUri);
    }
}
```

For simpler data structures, using serialized objects and files in isolated storage may not be necessary. However, when isolated storage is used for several types of local storage, it can help organize the data and provide easy access to read and write to it. Isolated storage should be used wisely to store data that should be cached locally.

Also note that users can ultimately clear the storage at any time since they have full control over their settings. This means that data stored in isolated storage should not be considered guaranteed persistent storage. Another good example is storing a list of U.S. states in isolated storage so a Web request (and database hit) does not have to be made every time a combobox needs to be filled with a list of U.S. states.

Wrap-Up

The sample application demonstrates how easy it is to load RSS and AtomPub feeds into a Silverlight application. Silverlight makes it possible to make a Web request, accept its results, handle cross-domain policy calls, load feed data into SyndicationFeed classes, query them with LINQ, bind them to UI elements, and store feed data in isolated storage.

Both last month and this month, Hanu Kommalapati covered building a line-of-business application with Silverlight, which you can read about in his articles "Silverlight: Build Line-Of-Business Enterprise Apps with Silverlight, Part 1" and "Silverlight: Build Line-Of-Business Enterprise Apps with Silverlight, Part 2."

Send your questions and comments for John to mmdata@microsoft.com.

John Papa (johnpapa.net) is a Senior Consultant with ASPSOFT and a baseball fan who spends summer nights rooting for the Yankees with his family. John, a C# MVP and INETA speaker, has authored several books and is now working on his latest, titled *Data-Driven Services with Silverlight 2*. He often speaks at conferences such as DevConnections and VSLive.