Gathering RSS Feeds using Visual Studio and RSS.NET

04 May 2007
by John Papa

Integrating RSS feeds into applications has become very popular. There are several tools (such as Microsoft Outlook 2007) available to read blogs and news from RSS feeds. There are also some tools available (such as RSS.NET and the ASP.NET team's RSS Toolkit) that can aid you in the development of customizing feeds in your applications. This article will use the RSS.NET assembly, which can be freely downloaded and used from http://www.rssdotnet.com/.

Application overview

This article will explain how to create a custom Windows Service that retrieves posts from multiple RSS feeds and then stores them in a SQL Server database. Once you store the posts in the database you can then retrieve them and display them on a web site, so you can combine multiple feeds and their posts in a list on a web page, or perhaps in a WinForm application. When the Windows Service retrieves the posts from each RSS feed, it filters out any duplicate posts before saving them to the database.

NOTE:
All source code for the Windows Service is available to be downloaded as well as the SQL scripts to generate the database in SQL Server. See the CODE DOWNLOAD link in the header of this article.

Figure 1 shows a high level overview of the application. The Windows service will poll a list of RSS feeds using the RSS.NET toolkit at a defined interval. The list of RSS feeds and their URLs are stored in a local SQL Server DB. Using the RSS.NET library, each feed's posts are retrieved and examined. Duplicate posts are ignored, but new posts are inserted into a SQL Server database.

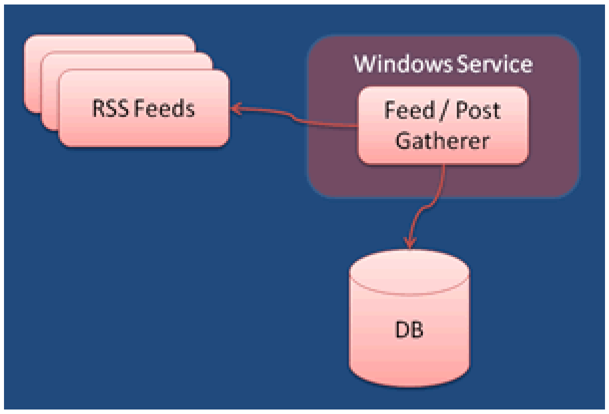


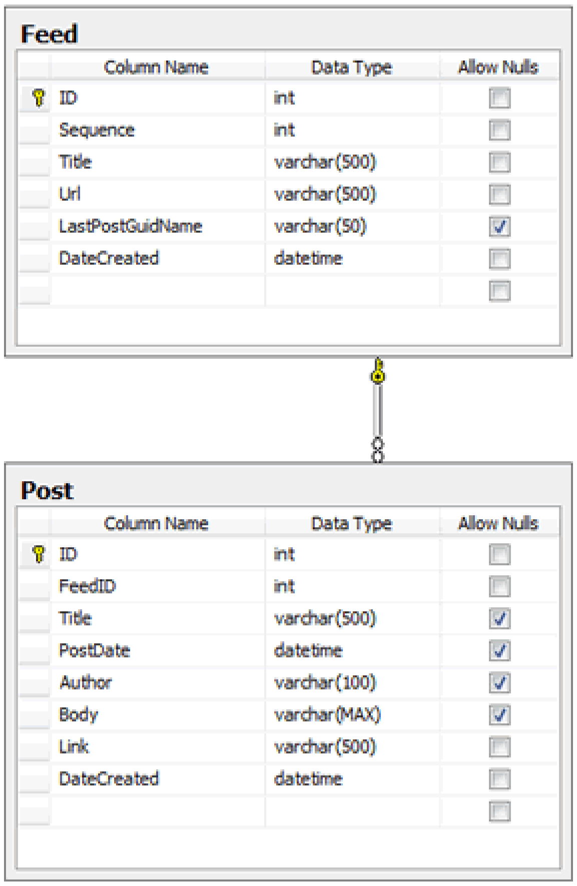
Figure 1 – Gathering posts from feeds

Setting up the database

You can execute the entire database setup script from the **DBScript.sql** file in the zipped code file attached to this article. The database is used to store two tables:

- Feed – a list of feeds that will be polled for their posts
- Post – a list of the posts for each feed

The tables and their basic schema are shown in Figure 2.



Feed

	Column Name	Data Type	Allow Nulls
🔑	ID	int	☐
	Sequence	int	☐
	Title	varchar(500)	☐
	Url	varchar(500)	☐
	LastPostGuidName	varchar(50)	✓
	DateCreated	datetime	☐
			☐

Post

	Column Name	Data Type	Allow Nulls
🔑	ID	int	☐
	FeedID	int	☐
	Title	varchar(500)	✓
	PostDate	datetime	✓
	Author	varchar(100)	✓
	Body	varchar(MAX)	✓
	Link	varchar(500)	☐
	DateCreated	datetime	☐
			☐

Figure 2 – The BlogRoll Database

The DBScript.sql file first sets up the SQL Server database and names it **BlogRoll**. You will want to adjust the location of the data and log files on your PC to a place besides C:\. Once the **BlogRoll** database is created, the **Feed** and **Post** tables are created. The **Feed** table stores the title of the feed, the sequence in which the feed should be polled in regard to other feeds, as well as the URL of the feed.

The **Feed** table also stores a **LastPostGuidName** column. This column represents a unique identifier for the last post that was retrieved for this feed. When the feed is polled again, this column's value is compared against each post that is retrieved. When a match is found, the Windows Service will know that the rest of the posts have already been retrieved so it will ignore the remaining duplicate posts. This column helps keep only unique posts in the **Post** table. This column allows NULL values because the first time a feed is polled, there are no posts already in the database.

I started the **Post** table with a list of two RSS feeds which point to my blog (http://codebetter.com/blogs/john.papa/rss.aspx) and the ADO.NET team's blog (http://blogs.msdn.com/adonet/rss.xml). The DBScrpt.sql file inserts these two feeds as samples that you can start with. You can add more RSS feeds to this list by adding records to the **Post** table.

The **Post** table stores a reference to the **Feed** table, so the posts can be associated with the feed from which they came. It also stores the title of the post, the date the post was published, its author, the body of the post and a link directly to the post.

Finally, the DBScript.sql file creates a SQL Server login and user named **BlogRollUser**. It gives the **BlogRollUser** permission to access the **BlogRoll** database and grants it permission to perform CRUD (Create, Read, Update, Delete) operations on the **Feed** and **Post** tables. Of course, you can set up your application to use any user that you want if you do not want to use the **BlogRollUser** that the script generates for you.

Gathering the feed list

The next step is to create the Windows Service that will poll the RSS feeds. I started by creating a Windows Service project through Visual Studio.NET and naming it **FeedGatheringService**. I then referenced the **System.Transactions** assembly as well as the RSS.NET assembly.

I added a Windows Service with the same name as the project and created a **System.Timers.Timer**, named **feedGathererTimer**, in it. In the Service's constructor I initialize the timer to run every 60 seconds. I also add an event handler for the timer's **Elapsed** event. This event will be used to grab the list of feeds and to poll them for their new posts.

```
public FeedGatheringService()
        {
            InitializeComponent();
            feedGathererTimer = new System.Timers.Timer();
            feedGathererTimer.Interval = 30000; //900000;
            feedGathererTimer.Enabled = false;
            feedGathererTimer.Elapsed += new ElapsedEventHandler(feedGathererTimer_Elapsed);
        }
```

The **feedGathererTimer_Elapsed** event grabs a list of feeds from the SQL Server database. I created a **Feed** class which has a static method called **GetList**, which queries the database's **Feed** table for all of the feeds. Each feed row that is retrieved is used to populate a **List<Feed>**. The following listing shows how a connection is opened to the database using ADO.NET. The feed list is grabbed and each row is used to create a **Feed** entity through the **Feed** entity's constructor. Each entity is added to the **List<Feed>** and finally the list of returned so the service can poll them.

```
public static List<Feed> GetList()
        {
            List<Feed> feedList = new List<Feed>();
            using (SqlConnection cn = new SqlConnection
(Settings.default.BlogRollConnectionString))
            {
                cn.Open();
                string sql

= "SELECT ID, Title, Url, LastPostGuidName, DateCreated FROM Feed ORDER BY Sequence";
                using (SqlCommand cmd = new SqlCommand(sql, cn))
                {
                    cmd.CommandType = CommandType.Text;
                    SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
                    while (rdr.Read())
                    {
                        feedList.Add(new Feed(Convert.ToInt32(rdr["Id"]),
                            Convert.ToDateTime(rdr["DateCreated"].ToString()),
                            rdr["Title"].ToString(), rdr["Url"].ToString(),
                            rdr["LastPostGuidName"].ToString()));
                    }
                }
            }
            return feedList;
        }
```

Polling the feeds for posts

Once the **List<Feed>** has been retrieved, the Service iterates through each of them and polls the feeds for their posts. This is where a library, such as RSS.NET or the ASP.NET team's RSS toolkit, comes in handy. These types of tools make interacting with RSS feeds much simpler than if you hit them directly and parse the results. There is a considerable amount of exception handling and data cleanup that is handled by these libraries, and they are both freely available to use.

RSS.NET has a **RssFeed** class that exposes a public method called **Read**, which accepts a URL of a feed. It requests a list of posts from the feed and returns them to the application. The posts are contained within the **RssFeed** class's **Channels** collection's **Items** collection.

```
rssFeed.Channels[0].Items
```

The following code listing demonstrates how the RSS.NET **RssFeed** class polls the feed's URL and returns the posts. Each post (represented by the **rssItem** variable) is then examined to see if its **Guid.Name** matches the last post that was retrieved for this feed (in previous polling attempts). The first time this code is executed for a feed the **LastPostGuidName** value will be NULL. However once a post has been inserted into the **Post** table, the **LastPostGuidName** value is updated for the **Feed** table. If a match is found then the processing for this feed stops as the rest of the posts already exist in the **Post** table.

```
public static List<Post> GetNewRssPostsFromUrl(Feed feed)
        {
            string mostRecentPostGuidName = null;
            List<Post> postList = new List<Post>();
            RssFeed rssFeed = RssFeed.Read(feed.Url);
            if (rssFeed.Channels.Count == 0)
                return postList;

            // Get Channel 0
            RssChannel channel = rssFeed.Channels[0];

            foreach (RssItem rssItem in channel.Items)
            {
                // If we already have this post, exit. This means the rest of posts are old.
                if (feed.LastPostGuidName == rssItem.Guid.Name)
                    break;

                // Grab the first Post's Guid
                if (mostRecentPostGuidName == null)
                            mostRecentPostGuidName = rssItem.Guid.Name;
                Post post = new Post();
                post.FeedID = feed.Id;
                post.Author = rssItem.Author;
                post.Link = rssItem.Link.ToString();
                post.PostDate = rssItem.PubDate;
                post.Body = rssItem.Description;
                post.Title = rssItem.Title;
                postList.Add(post);
            }

            // Update the Feed's last post setting
            if (mostRecentPostGuidName != null)
                feed.LastPostGuidName = mostRecentPostGuidName;

            return postList;
        }
```

The code then shows that once the post is determined not to be a duplicate that a **Post** entity is created. The **Post** entity is created from the values retrieved from the **rssItem** and then added to a **List<Post>**.

The final step before returning the **List<Post>** is to update the **LastPostGuidName** property. This is set so that the next time the polling process occurs that it will know what the identifier is for the last post that was retrieved. Notice that the posts are not yet inserted into the database. The entire process merely gathers the posts and sets the **LastPostGuidName**.

Inserting the posts

Once the **List<Post>** has been returned to the service, they must be inserted into the database's **Post** table. The following code in 5 demonstrates how the posts are inserted. First a **TransactionScope** is instantiated using the **System.Transactions** library. I create a transaction since I want all of the posts for a given feed to be inserted together. If any post fails to insert then I want all of the posts to rollback. Using the **System.Transactions TransactionScope** class I can maintain a 2-phase commit that will implement this atomic transaction. Since this uses a 2-phase commit, it requires that the MS Distributed Transaction Coordinator service is running. (Of course, if you wish to eliminate the transactions or write them a different way, you can simply replace the **System.Transactions** code with your own transactional code.)

```
using (TransactionScope ts = new TransactionScope())
                {
                    // if the Feed and its Guid changed, update the feed
                    if (feed.LastPostGuidNameChanged)
                        feed.Update();

                    // If it has posts, add them
                    if (postList.Count > 0)
                        foreach (Post post in postList)
                            post.Add();

                    ts.Complete();
                }
```

The first part of the transaction is to update the **LastPostGuidName** column in the **Feed** table. Then, each post is inserted into the **Post** table. Finally, if all goes well and no exception is thrown, the transaction is marked complete and it is committed to the database. A new transaction is created for each feed, so each feed is separate from the next feed.

Creating an installer

When creating the Windows Service, I set the timer's interval to 60 seconds so I could easily debug and watch the feeds populate. However, the reality is that I probably do not want this service running every minute since feeds do not usually change that often. I suggest that you change the timer's interval to something less frequent such as every 15 minutes (an interval of 900000).

If you want to debug the application, you can adjust the interval back down to 60 seconds for testing purposes.

Once the Windows service was created I needed a way to install the service on my machine to try it out (and to debug it). To do this I opened the **FeedGatheringService.cs** file's designer and right-clicked in open space to bring up the context menu shown in Figure 3. I selected the *Add Installer* option from this menu, which adds a **ProjectInstaller.cs** file to the project.

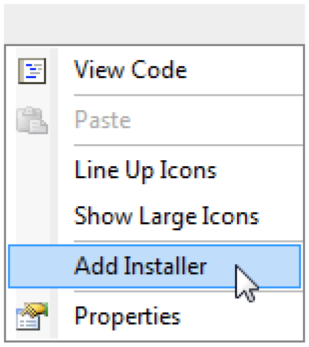


Figure 3 –Adding the Installer

The **ProjectInstaller** provides a component that will be used to install the service process and a component that will be used to install the service itself. You can examine these components by opening the P**rojectInstaller** in design view. I set the **serviceProcessInstaller1** component's **Account** property to **LocalSystem**. I could have specified a specific user, and in a production application I recommend doing just that. However, for demonstration purposes, I will use **LocalSystem** so it is easier to debug the application.

I also set the **serviceInstaller1** component's **ServiceName** property to **FeedGatheringService** and set its **StartType** to Automatic. This tells the service to start itself automatically when the computer is started. I set the description to "Gathers RSS Feeds" so the service will have some basic description when viewing it in the Services list.

Install the Windows Service

Once the project is compiled the service can be installed on the computer by following a few simple steps. Open a Visual Studio.NET command prompt and change to the **bin\Debug** directory of the **FeedGatheringService** project. (You will want to use the **bin\Release** directory when if you compile a release version.) Execute the following command to install the service on the computer:

```
InstallUtil.exe FeedGatheringService.exe
```

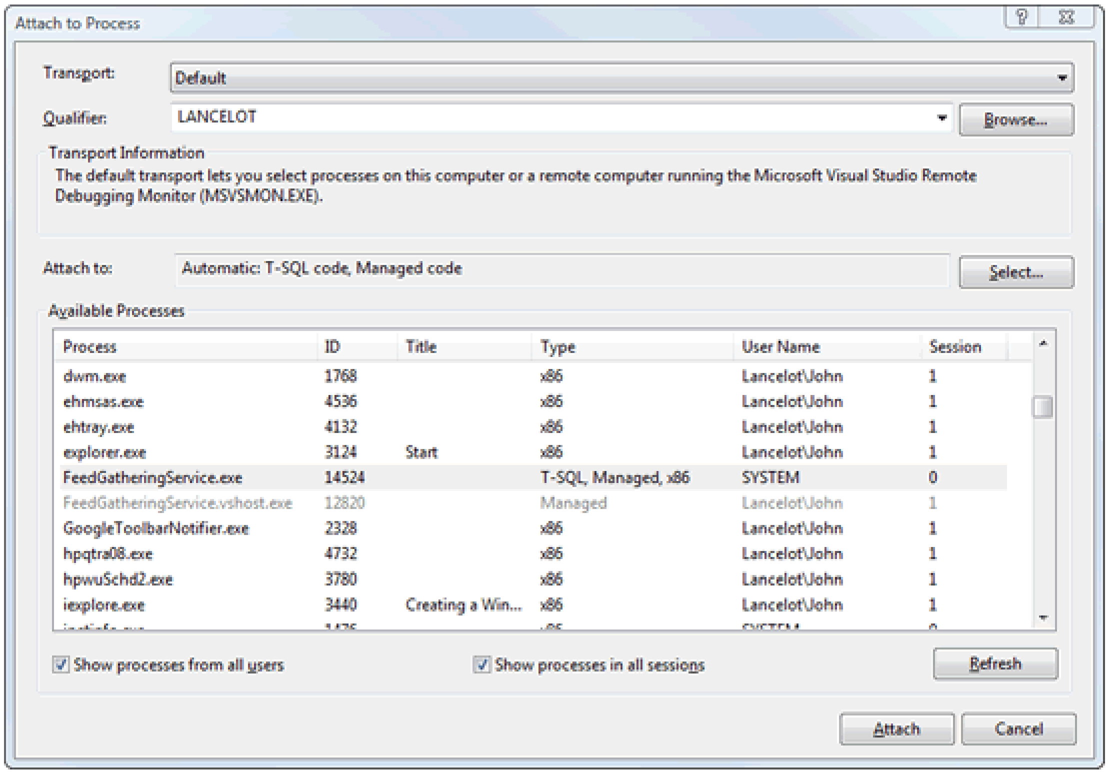
The **InstallUtil** command will install the service and create the appropriate registry entries for the service. (If you want to uninstall the service you can execute the **InstallUtil** command again with the /u argument.) It will not start the service immediately. Keep in mind that the next time the computer is restarted that the service will start automatically since I set the **StartType** property to Automatic in the **ServiceInstaller** component. Execute the following command in the Visual Studio.Net command window to start the service:

```
net start FeedGatheringService
```

Another option to start the service is to right-click on My Computer, select Manage from the context menu, open the Services node in the tree, locate the **FeedGatheringService** in the list of services and start it from there.

Debugging the Service

Debugging a Windows service is not difficult, but is does require a few additional steps beyond what is involved with a WinForm or ASP.NET application. First, set a breakpoint in the **feedGathererTimer_Elapsed** event handler so you can watch the code execute. Then, go to the Debug menu and select Attach to Process. Make sure that the checkboxes are selected so you can see all processes. Next, select the **FeedGatheringService** from the list of available processes and click the Attach button (shown in Figure 4).



Transport: Default

Qualifier: LANCELOT

Transport Information

The default transport lets you select processes on this computer or a remote computer running the Microsoft Visual Studio Remote Debugging Monitor (MSVSMON.EXE).

Attach to: Automatic: T-SQL code, Managed code

Available Processes

Process	ID	Title	Type	User Name	Session
dwm.exe	1768		x86	Lancelot\John	1
ehmsas.exe	4536		x86	Lancelot\John	1
ehtray.exe	4132		x86	Lancelot\John	1
explorer.exe	3124	Start	x86	Lancelot\John	1
FeedGatheringService.exe	14524		T-SQL, Managed, x86	SYSTEM	0
FeedGatheringService.vshost.exe	12820		Managed	Lancelot\John	1
GoogleToolbarNotifier.exe	2328		x86	Lancelot\John	1
hpqtra08.exe	4732		x86	Lancelot\John	1
hpwuSchd2.exe	3780		x86	Lancelot\John	1
iexplore.exe	3440	Creating a Win...	x86	Lancelot\John	1

[✓] Show processes from all users [✓] Show processes in all sessions

Figure 4 - Attaching to the Windows Service

If you set the breakpoint in the **feedGathererTimer_Elapsed** event handler then the code will hit the breakpoint and you will be able to walk through the code as it executes. The breakpoint will be hit once the timer's interval has been reached.

Wrapping Up

At this point you now have a Windows service that will gather posts from a list of RSS feeds and store them in a database. The code included with this article points to a specific database server instance of (local)\SQL2005. Remember to change this to reflect the name of your SQL Server instance. The posts can now be read using standard ADO.NET data access objects and displayed in any of your applications.