**Leveraging MSMQ in ASP.NET Applications**

Posted by [**Quin Street**](http://www.codeguru.com/member.php/Quin+Street/) on **January 10th, 2008**

Top of Form

* 0**By Greg Huber**

Have you ever developed a Web application that requires extensive processing? Ever had long running Web pages that often time out in the browser? Most Web developers run into these kinds of issues at some point. As Web application requirements evolve, processing and business logic tend to get more complex. Consequently, it is no surprise that Web applications can lead to frustrating Web experiences.

On the other hand, there may also be processes that simply take time no matter how simple or complex the application. At times, there is processing that typically does not lend itself to occur in a Web application.

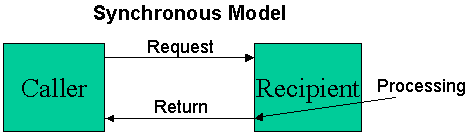
The above scenarios require an effective way to initiate long running processes so that inherit problems due to the synchronous nature of the Web and long running processes are minimized or avoided all together.

**Requirements**

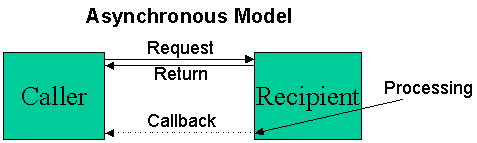
.NET Framework 1.1 (Visual Studio .NET helpful but not required)  
Basic familiarity with ASP.NET, .NET Framework  
Windows 2000 Professional/Server or Windows XP or 2003 Server

**Synchronous vs. Asynchronous Programming Concepts**

Before diving into the code, it's important to understand a few key concepts regarding programming models. Let's start out with the Synchronous model. In its simplest form, this programming model means that the caller waits until all processing has completed and control is returned.



Alternatively, asynchronous programming means that the caller makes a call and proceeds immediately. A "callback" located in the callers process space may be invoked by the recipient if desired (and available).



The asynchronous concept not only applies to programming models, but also to architectures. Similar to the programming model, a Web application can be architected in such a way that an asynchronous model is achieved. Hence, many of the limitations of the synchronous nature of the Web can be dealt with. In this article, MSMQ will be leveraged to show how to implement such a model.

**Overview of MSMQ**

MSMQ is the abbreviation for "Microsoft Message Queue". A queue is simply a message store that is accessible via an API. In .NET, the System.Messaging library is used to interface with an MSMQ.

The primary purpose of MSMQ is to allow for implementing reliable, scalable, high performance distributed applications. Microsoft first released MSMQ with Windows NT 4.0, implemented via COM components. This initial release was fairly complex to deal with in a programmatic fashion. Nevertheless, it introduced many important concepts and benefits, which have been improved upon and carried forward to the current version of MSMQ. Considerable performance gains have also come about since MSMQ 1.0.

For more information, previous version and current features, and support information on MSMQ, etc. please visit the [Microsoft MSMQ Center](http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp). Also visit the article titled "[Accessing Message Queues](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetasync1.asp)" for information on reading messages.

**Getting Started**

The remainder of this article will pose a common scenario, a solution with MSMQ, and walk through the necessary steps to implement the solution.

**A Common Scenario**

Over time, business requirements evolve, and applications need to be robust enough that they can keep up. Consider the case where a complex calculation evolves to the point where a Web application can no longer process in "Web-time". In fact, as requirements evolve, data for such a calculation may even come from another system all together, which certainly complicates things.

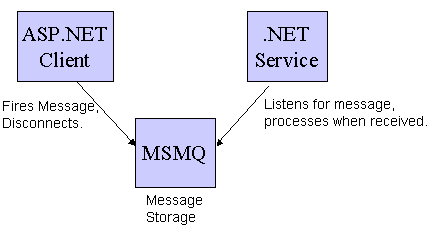
In other cases, an application you are planning on developing may be a great fit for the Web, with the exception of one or two back-end pieces -- for example, generating a report, moving large amounts of data from a data source, etc. These are often activities that Web applications need at some point in time, but due to the lengthy processing time, the Web doesn't lend itself well.

Unfortunately, there may be little you can do to increase performance / response times in these kinds of scenarios. Often, developers resort to creating a thick-client Windows application that can reside on a desktop-which allows for more control over timeouts. However, many complexities are also introduced, i.e. processing moves from the server to the client and numerous issues may arise including scalability, security, deployment, etc.

One great solution is implementing an asynchronous programming model using MSMQ.

**The Proposed Solution**: "Disconnected Processing" with MSMQ

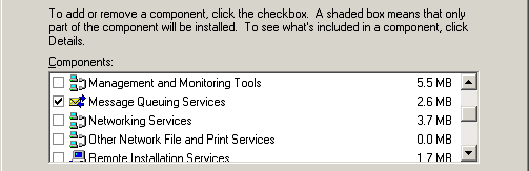
At a high level, there are three major logical components. These components may exist on the same machine or could be on completely separate machines.



The ASP.NET client sends a message to the message queue and moves on its merry way. The MSMQ component is a simple message broker; it allows the reading and writing of messages. When a message is read, it is removed from the queue. The .NET service monitors the MSMQ and processes as soon as a message is received. The messages will pile up in the queue until the .NET service is ready to process the next message.

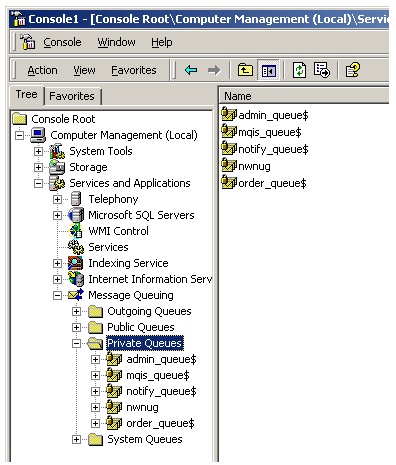
**Pre Requisite: Setting up MSMQ**

Ensure that you have your Windows 2000/XP CD handy. Simply open the control panel, "add/remove" Windows components, and ensure the Message Queuing Services checkbox is checked. Message queuing must be installed on both the sending (client) and receiving machine (server).



After this step is completed, you will be able to create a private message queue. A private queue is for "workgroup installations", meaning the computer is not integrated in the Active Directory. There are some limitations that exist in private queues-they are not published whereas public queues are.

Open up the Microsoft Management Console, add/remove the "Computer Management" snap-in, browse to "Services and Applications", and expand and select the "Message Queuing" node. Right click on "Private Queues" and create a new queue.



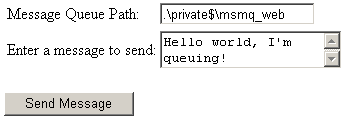
**Code Walkthrough: Writing a message from the Web**

Now that you have set up a private message queue to send and receive messages, you can easily write to it with a Web application. There are a few simple steps to get started. The code for these steps have been included as well-simply unzip to a virtual directory and modify the web.config with your settings.

**Important Steps for setting using Messaging in an ASP.NET page**

|  |  |
| --- | --- |
| **Visual Studio.NET Method** | **Framework Method** |
| Add a reference in your Solution to System.Messaging: http://www.codeguru.com/images/article/19563/031203_06.gif | Explicitly add the assembly to your web.config under the :  http://www.codeguru.com/images/article/19563/031203_07.gif  \*\* Note, you can verify the assembly information by using "gacutil" tool. |
| Add the line to the top of the class:  using System.Messaging; | Add the line to the top of the page:  <%@ Import Namespace="System.Messaging" %> |

After completing the above steps, your code will be ready to interact with the MSMQ. For demonstration purposes, you can easily create an ASP.NET page with a simple user interface to send a message to a message queue:



This interface shows the two important elements that are necessary for sending messages to an MSMQ-the message queue path and the message itself.

**Message Queue Path**  
A message queue path string must be properly constructed. The syntax of the path can vary widely, depending on the type of queue you are talking to, and where it is located. In the above example, the MSMQ resides on the same machine as the Web server. However, in realistic situations, the MSMQ will reside on another machine. Examples of queue paths:

|  |  |  |
| --- | --- | --- |
| **Queue Type** | **Location message is sent from** | **Example** |
| Public Queue (must belong to ActiveDirectory) | Anywhere | ServerName\QueueName |
| Private Queue | Same server MSMQ resides on | .\private$\QueueName |
| Private Queue | Anyremote location | FormatName:DIRECT=OS:machinename\private$\queuename |

**Message**  
When using MSMQ with .NET, a message can be **any** serializable object. When serializing a message to MSMQ, you can use different formatters (including Binary, XML, or even ActiveX which allows for sending COM objects-useful if you want to tap into an existing MSMQ aware application based on COM). You can specify the formatting of your message through the message object that you are sending. For more information on message options, please see the MSDN article on [Accessing Message Queues](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetasync1.asp).

In the examples in this article, the default formatter (XML serialization) is used to send a message to the queue, and the message is simply a string object. As mentioned previously, this message could be anything serializable, such as a business object that could then be consumed directly by the listener process.

**Code Snippet: Sending a Message**

1. MessageQueue MyMessageQ;
2. Message MyMessage;
3. MyMessageQ = new MessageQueue(txtQueuePath.Value);
4. MyMessage = new Message(txtMessageToSend.Value);
5. MyMessageQ.Send(MyMessage);
6. divMessage.InnerHtml= "<b>Message sent
7. successfully!</b>";

In this scenario, one important reason for using MSMQ is to provide the capability to separate out lengthy processing from the user. Consequently, after sending a message, the Web page has completed processing and control is returned immediately to the user.

**Code Walkthrough: Receiving & Processing a Message**

To receive a message with MSMQ in the given scenario, a separate process must be running that is watching the queue. A great way to implement this process is to create a simple .NET service that listens to the queue and processes as soon as a message is sent to it.

**Code Snippet: Receiving a message in the .NET service**:

MessageQueue MyMessageQ;

Message MyMessage;

MyMessageQ = new MessageQueue(\_QueuePath);

MyMessage = MyMessageQ.Receive;

WriteStatus("Message Received!");

DoSomeLongRunningProcess();

WriteStatus("Processing Finished!");

The "DoSomeLongRunningProcess" method contains code that performs a process that would otherwise cause a timeout if a Web page were running it. Throughout the processing of this method, a database table will be updated (shown later) with status information so at any given point in time, the status of the processing can be determined.

In order to install the service included in the source code, you can simply run the following command after shelling out to the command line:

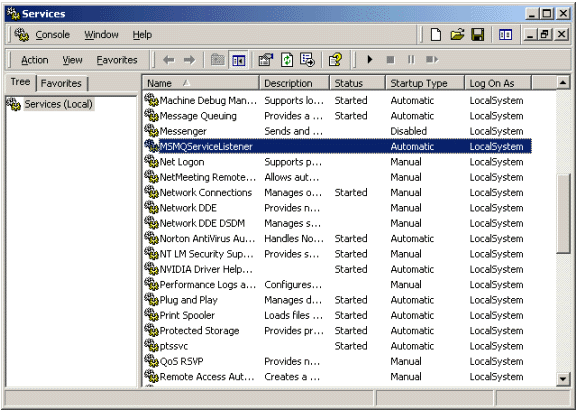
***C:\WINNT\Microsoft.NET\Framework\v1.1.4322\installutil msmqService\_listener.exe***

Upon successful installation, you will see some informational text scroll by and a confirmation message:

*"The Commit phase completed successfully.*

*The transacted install has completed."*

After installing the listener service, you can open the administrative control panel and browse the current Windows services running on your machine. You should see the service you just installed as pictured below.



After unzipping the service, you will want to ensure that the paths in the MSMQService\_Listener.exe.config are correct (instructions are in this file- which is located in the bin directory).

You can now start your service by right clicking "start" and be on your merry way!

**Other Considerations**

The message queue **Receive** method waits indefinitely until a message is received on the queue. Depending on the type of processing you are doing, you may want to queue up some worker threads so that multiple messages can be received and processed at once. Consequently, this type of processing can be very robust and scalable.

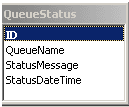
There are a variety of other methods on the message queue object certainly worth investigating. The following are a few that are noteworthy:

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| Peek | Performs a non-destructive read a message, meaning the message is not deleted after "peeking". (Read purges the message) |
| BeginRead / EndRead, BeginPeak/ End Peak | Allows for asynchronous handling of the message. Code will continue running while a message is being received; an event handler will execute when the message has been fully received. |
| Delete | Deletes a message out of the queue. |

**Code Walkthrough: Checking Status Information**

As hinted above, in the scenario where a Web page has triggered a long running process, a user may want status information. One simple technique for providing status information would be to write information (from the .NET service that is doing the processing, as mentioned above) to a database table. The Web application can then check this database table at the request of the user to determine current status.

**Database Table**



The table above is a simple table that keeps track of a Queue Name and current status. In a real world scenario, you would most likely have more information that you may want to keep track of (such as errors that occurred, information about the message received, etc.). When the listener service receives a message, this status table is updated throughout the processing that occurs.

**Code Snippet: Viewing the Table**

string dbConnString =

ConfigurationSettings.AppSettings["AccessDBConnectionString"];

//Create connection object

OleDbConnection oCon = new OleDbConnection(dbConnString);

string sSQL = "Select \* FROM QueueStatus ORDER By ID Desc";

OleDbCommand oCmd= new OleDbCommand(sSQL, oCon);

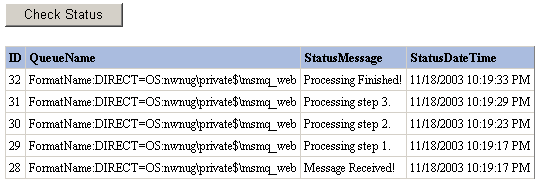
oCon.Open();

dgStatus.DataSource = oCmd.ExecuteReader();

dgStatus.DataBind();

In the source code provided, a Web-based interface reads this table and simply displays the current status. To get the latest status, the user must hit the "show status" button.

Here is a sample view of seeing how the status has progressed:



**Conclusion**

This article demonstrated a simple technique for handling long running Web processes through the Web via Microsoft MSMQ and the System.Messaging framework. MSMQ is a very useful product that developers should consider leveraging in Web applications.

**About the Author**

Greg Huber is the President and Founder of the Northwest Ohio .NET Users group (http://www.nwnug.com). He is active in the .NET community-speaking at area user groups, serving on the INETA (http://www.ineta.org) user group relations committee, and spearheading community-oriented development initiatives, including Quizzard, a shared-source .NET quiz game. He has been developing and architecting software on the Microsoft platform since 1998, and is currently a lead developer at NFO Worldgroup.