Manage BLOB Data Fields

ADO.NET and SQL Server let you retrieve random images for display in your WinForms apps.

by Fabio Claudio Ferracchiati and Juval Löwy

Technology Toolbox

- ☑ VB.NET
- ☑ C#
- ☑ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6
- ☑ Other: ADO.NET

Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these related resources.

Download

VS0308QA Download the code for this article, which includes applications that demonstrate subclassing with VB.NET and asynchronous event publishing.

Discuss

VS0308QA_D Discuss this article in the ADO.NET, VB.NET, and C# forums.

Read More

VS0308QA_T Read this article online. It includes Listings 2 and 3.

VS0304DE_T "Follow the Top 10 ADO.NET Tips" by Dino Esposito

VS0206BS_T "Subclass Controls in .NET" by Bill Storage

VS0204JL_T "Tame .NET Events" by Juval Löwy

Q: Manage BLOB Data Fields

I need to develop an application that displays random images stored in a Microsoft SQL Server database in a picture box. Is there an easy way to accomplish this functionality?

A:

Yes. ADO.NET and SQL Server give you all the tools for retrieving binary large objects (BLOBs) easily from a database. You can use a simple application I created as a model for implementing dynamic image changing in a WinForms form. Start by examining the database structure:

```
CREATE TABLE [dbo].[T_Image] (
   [ImageID] [int] IDENTITY (1, 1)
   NOT NULL , [ImageBinary] [image] NOT
   NULL ) ON [PRIMARY] TEXTIMAGE_ON
   [PRIMARY]
```

The table has two columns—an image identifier as primary key, and the image's binary code. As you can see, SQL Server provides an image data type that's useful for managing images within a database. This stored procedure retrieves the binary image data and provides an image identifier:

```
CREATE PROCEDURE dbo.RetrieveImage
   @id int
AS
SELECT ImageBinary FROM T_Image
WHERE ImageId = @id
```

This stored procedure inserts a new image into the database:

```
CREATE PROCEDURE dbo.InsertImage
   @i image
AS
INSERT INTO T_Image (ImageBinary)
VALUES (@i)
```

The code to execute a stored procedure is simple, thanks to the ADO.NET classes. You call the InsertImage stored procedure, which provides binary code from an image file (see Listing 1).

Calling a stored procedure from the code takes six steps. First, create a SqlConnection object that specifies the connection string of the database to connect. Second, create a SqlCommand object; use the new object to specify the StoredProcedure command type and the CommandText with the stored procedure name. Third, add a new Parameters collection item for each stored procedure parameter. Fourth, open the connection to the database. Fifth, use either

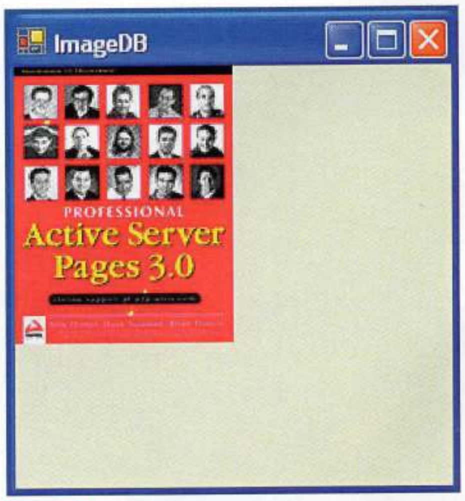

Figure 1 Retrieve an Image. You can retrieve an image from the database and assign it to a picture box. You change the image by passing the new image's identifier to the stored procedure.

VB.NET • Add an Image

```
Try
   Dim dbConn As New SqlConnection()
   dbConn.ConnectionString = ". . ."

   Dim dbComm As New SqlCommand()
   dbComm.CommandText = "[InsertImage]"
   dbComm.CommandType = StoredProcedure
   dbComm.Connection = dbConn
   dbComm.Parameters.Add(New _
      SqlParameter("@id", _
      SqlDbType.Int, 4, _
      ParameterDirection.Input, _
      False, CType(10, Byte), _
      CType(0, Byte), "", _
      DataRowVersion.Current, _
      Nothing))
   dbConn.Open()
   Dim fs As FileStream
   fs = File.OpenRead(<filemame>)
   Dim buffer(fs.Length) As Byte
   fs.Read(buffer, 0, fs.Length)

   dbComm.Parameters("@i").Value = _
      buffer
   dbComm.ExecuteNonQuery()

Catch ex As Exception
   MessageBox.Show(ex.Message)
Finally
   If (dbConn.State = _
      ConnectionState.Open) Then
      dbConn.Close()
   End If
End Try
```

Listing 1 This code calls the InsertImage stored procedure in the SQL Server database to add an image as a binary large object (BLOB) item. When you use ADO.NET to call a stored procedure, you must specify each of its parameters in the Parameters collection. Finally, you must provide the values to insert in the database and call the ExecuteNonQuery method.

the ExecuteNonQuery method to execute a stored procedure that doesn't return a value, or the ExecuteReader method to execute a stored procedure that returns a reference to the DataReader object. Finally, close the connection.

As you can see in Listing 1, the FileStream object reads the contents of the image file. You create a Byte buffer for containing the image content, then pass it to the stored procedure you'll insert into the database.

You must execute the RetrieveImage stored procedure in order to retrieve an image from the database and provide the image identifier of the image you want to retrieve:

```
Try
   dbConn.Open()
   dbComm.Parameters("@id").Value = _
      <imageid>
   Dim buffer() As Byte
   buffer = dbComm.ExecuteScalar()
   Dim s As New MemoryStream(buffer)
   s.Write(buffer, 0, buffer.Length)
   pbImageDB.Image = _
      Image.FromStream(s)
Catch ex As Exception
   MessageBox.Show(ex.Message)
Finally
   If (dbConn.State = _
      ConnectionState.Open) Then
      dbConn.Close()
   End If
End Try
```

After the connection to the database is open, you can provide the image's identifier dynamically to retrieve the image from the database (see Figure 1). The SqlCommand class's ExecuteScalar method executes the SQL instruction in order to retrieve only the command's first parameter. This is useful when you execute a SQL command that returns only a value (for example, a newly inserted record's identifier or a specific image's bytes). Now that the buffer contains the stored procedure's result, the code uses a MemoryStream object to fill a memory space with the image bytes. This step is necessary because the Image class doesn't provide a method that reads image bytes. The Image class accepts either a filename or a handle to the bitmap, except for a Stream object. So, you change the picture box image dynamically by setting its Image object reference with the new Image object filled with the image bytes stored in a MemoryStream object. *—F.C.F.*

Q: Implement Subclassing in WinForms Forms

My application is basically a series of forms. Any number of the form's instances can be open at any time. I need to detect when another application has taken the focus through either mouse clicks or keystrokes, such as Alt+Tab. Is there an easy way to produce this effect? I want my program to shut itself down if another application takes focus.

A:

A WinForms form in the .NET Framework is subjected to the Activated event when the user selects it with either the mouse or keystrokes. However, this isn't a good event for your application, because it's raised when the form in a single document interface (SDI) application, or a child form in a multiple document interface (MDI) application, is activated—not when the application gets the focus from another application. Your only solution is to subclass the form (download the sample code from the *VSM* Web site; see the Go Online box for details).

Subclassing is an advanced technique for implementing non-standard Windows controls and features. I used it for the first time to create a listview control that displays a different color in each row. Subclassing lets you retrieve Windows system messages and change default behavior against these messages. Implementing subclassing in VB.NET is simple. You can override the Control class's WndProc methods directly in your form's code.

```
Protected Overrides Sub WndProc( _
    ByRef m As Message)
    MyBase.WndProc(m)
```

The first instruction you add within the method is the call to the base class's method. The Message parameter contains the Msg property, which indicates the Windows system message that the system fires. You can use a Select Case in the WndProc method to manage these messages, and you can find definitions of the messages' values in the Windows.h header file.

Your application must use the WM_ACTIVEAPP message that the OS fires when the user selects your application with either a mouse click or a keystroke. The WM_ACTIVEAPP Windows message fires with the word *param* set to zero when your application becomes inactive. You can check this value in order to close the application:

```
Select Case m.Msg
   Case WM_ACTIVATEAPP
      If m.WParam.ToInt32 = 0 Then
         Application.Exit()
      End If
End Select
```

You force the application to quit by calling the shared Exit method that the Application class provides. *—F.C.F.*

Q: Automate Asynchronous Event Publishing

You show how to publish events asynchronously in your "Tame .NET Events" article (*VSM* April 2002; see the Go Online box). I need to publish many event types asynchronously. Is there a way to automate the process instead of duplicating the code for every publisher and delegate?

A:

The technique I demonstrated in that article addresses the problem of publishing events asynchronously. .NET lets you use the BeginInvoke() delegate method to invoke the delegate target asynchronously on a thread from the thread pool. The only limitation to using BeginInvoke() is that the delegate must have only a single target in it; otherwise, an exception is thrown. Although it's normal to have a single target when you use a delegate dedicated to asynchronous invocation, you usually end up with multiple subscribers to events. The previous article's solution was to iterate manually over the delegate's internal invocation list and publish to every delegate in that list asynchronously (download Listing 2).

The problem with Listing 2's code is that it isn't generic, and you must repeat such code in every case in which you want to publish events asynchronously. Fortunately, you can write a generic helper class to automate asynchronous event publishing (download Listing 3 and the sample code).

You use the param modifier to pass in any collection of arguments, as well as the delegate containing the subscribers list. The FireAsync() method iterates over the internal collection of the passed-in delegate. For each delegate in the list, it uses another delegate of type AsyncFire to call the private helper method InvokeDelegate() asynchronously. InvokeDelegate() simply uses the Delegate type's DynamicInvoke() method to invoke the delegate.

Using EventsHelper to publish events asynchronously is easy, compared to Listing 2:

```
public delegate void
   NumberChangedEvent(int num);

public class MySource
{
   public event NumberChangedEvent
      NumberChanged;

   public void FireEventAsync(int num)
   {
   EventsHelper.FireAsync(
      NumberChanged,num);
   }
}
```

You also decorate EventsHelper's InvokeDelegate method with the OneWay attribute, which is defined in the System.Runtime.-Remoting.Messaging namespace. .NET doesn't keep track of the method invocation when it invokes a one-way method asynchronously, and it doesn't manage any completion callbacks or record exceptions. As a result, dispatching the call asynchronously involves no overhead. One-way methods mean semantically that the caller shouldn't care what happens after calling the methods. This is clearly the case with FireAsync(), because the EventsHelper's client doesn't care about the result of publishing the event asynchronously to all the subscribers. *—J.L.*

Fabio Claudio Ferracchiati has 10 years of experience using Microsoft technologies. He's been focusing attention recently on the new .NET Framework architecture and languages and has written books for Wrox Press about this technology. He works in Rome for the CPI Progetti SpA company (www.cpiprogetti.it). Contact him by e-mail at ferracchiati@rocketmail.com.

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and .NET migration. Juval is Microsoft's regional director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. His portion of this article derives from his latest book, *Programming .NET Components* (O'Reilly & Associates). Juval speaks frequently at software-development conferences. Contact him at www.idesign.net.

Additional Resources

- *Professional ADO.NET* by Fabio Claudio Ferracchiati et al. [Wrox Press, 2001, ISBN: 186100527X]
- *Programming Microsoft Visual Basic .NET (Core Reference)* by Francesco Balena [Microsoft Press, 2002, ISBN: 0735613753]
- *Programming .NET Components* by Juval Löwy [O'Reilly & Associates, 2003, ISBN: 0596003471]