relatia software corporation

its not in the manual

A Simple .NET wrapper for GoldMine's Stored Procedures Library makes GoldMine a snap.

The Back-story:

I've done a lot of work with GoldMine's Stored Procedure API (GMSPROCS). I find it a very easy tool to use for on-the-fly data import or database manipulation. It's perfect for SQL Jobs or SQL trigger work due to the obvious advantage of being entirely available from within Microsoft SQL Server's DBMS which means you can manipulate GoldMine's database tables without having to fiddle with writing an application infrastructure capable of calling both the SQL data collection and GoldMine's traditional API's simultaneously. This makes it realistic to quickly code solutions to data-centric problems that don't need a user interface. GMSPROCS also provides an iron-clad option for GoldMine integration with web applications due to the fact that there's no BDE to load so the problems normally associated with loading and unloading the BDE from the IIS sandbox go away entirely. To make the SPROCS even simpler, I rely on a custom "black-box" ASP component I wrote a long time ago that takes all the API calls out of the inline ASP code and isolates them in a single include file. When Microsoft's ASP.Net platform came on the scene I decided to extend this concept to a custom data class that wraps the SPROCS API so that adding GoldMine records could be done quickly and easily in either ASP.Net or .Net forms applications.

The API wrapper

Getting Started:

First, a quick primer on how the GMSPROCS work, if you've spent any time with them, you could probably skip this section, but for those of you who are new to the concepts this might be helpful. In a nutshell, GMSPROCS basically mimics the business logic methods found in the dll or COM api by allowing you to create a "name/value" container (N/V container) which ends up containing a list of changes you want to make to a GoldMine record, or, creates a new one. On the inside, the typical name/value pair container might look something like this after you've created and populated it:

"Company" "Bob's Bait and Tackle"
"Contact" "Bob Smith"
"Phone1" "8005555555"
"Fax" "7703334455"
"Key1" "Bait Shop"
"UMYFIELD" "Sadsack bait products"
and so on….

After you get the container built (instantiated and populated), you call one of the business logic functions, feeding it the handle to the N/V container. Kind of like this:

Exec GMW_WriteContact "1003003"

Where 1003003 is the handle to the N/V container you've created and populated. Each of the business logic methods (i.e. write contact, write schedule, write history and so forth) have their own list of valid name/value pairs for that function as well as some simple rules and the return values you can expect.

In the GMSPROCS API, the N/V Containers are created in the tempdb in the format: ##GMVN1003003.

If I've managed to make this sound complicated somehow, it isn't. The few examples provided in the GMSPROCS docs are more than sufficient to get you started.

The C# part:

I wanted to write a wrapper that was very straight-forward and simple to use in a variety of situations and therefore decided to off-load the data grooming and pre-processing to the calling class. You'll have to understand the GMSPROCS API pretty well before using this wrapper as it's up to the caller to be sure to follow the rules. For my situation, this is perfect because I just wanted to offload the typing, not the thinking.

I'm going to assume you have some experience with C# and ADO.Net so I'm not going to go into a lot of detail here about what's happening in the code, except where it's relevant to the GMSPROCS interface.

This example was created for use with my company's website so I created a stand-alone class file which makes it portable by simply changing the namespace. The headers…

```
using System;
using System.ComponentModel;
using System.Collections;
using System.Diagnostics;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;


namespace www.yourdomain.com
```

I implemented everything in a single class with two methods which results in a logic flow something like:

1. Instantiate the class
2. Send a bunch of name/value pairs
3. Call one of the business logic methods (which run against the n/v pairs).

This way, you can use the wrapper to write any kind of GoldMine record, as long as there's a business logic method supplied in the GMSPROCS API.

```
{
	public class writeGMRecord
	{
		private string _user;
		private string _gmnv_handle;
		private ArrayList _nvpairs = new ArrayList();
		private string _field;
		private string _value;
		private string[] _fieldValue= new string[2];
```

the constructor:

```
public writeGMRecord(string user)
		{

			this._user=user;
			this.adnvPair("USER",user);

		}
```

pretty much every business logic method requires a "user" name/value pair (which GMSPROCS sets as the record creator in GoldMine's DB tables) to exist in the N/V

container or it will fail so I made it required by placing it in the constructor, then immediately writing it to the name/value pair container(which will become clear in a minute).

Now, the first public method in the class:

```
public int adnvPair(string gmfield, string gmvalue)
{
	_nvpairs.Add(gmfield + "|" + gmvalue);
	return _nvpairs.Count;

}// end adnvPair method
```

As you can see, this is pretty straight forward. It just adds a name/value pair to the Array List I'm using to temporarily store the name/value pair collection. The C# ArrayList collection object does not allow multi-dimensional arrays, so I'm just using the pipe symbol as a delimiter so that each name/value pair occupies a single index position in the array list. Later, when I pull them back out, I can split each pair back apart.

As you'll see later, the GMSPROCS calls that are used to create and build the container are GMW_NV_Create (to create the container) and GMW_NV_SetValue to add a name/value pair to the container. You might be wondering why I don't just create the container and add the values as they come in from the calling code rather than bothering with the extra step of storing them in a local array list then having to drag them back out. The reason for this is that .NET's garbage collector will close the ADO objects we'll use to call the PROCS which will delete the ##temptable (the actual N/V Pair in SQL Server) before we get a chance to use it. In case you missed it, that would be bad. We'll avoid this problem by doing all the database work in the next method.

Okay, so we have our method to build the name/value pairs, now all we need is a way to actually write whatever kind of record the calling code wants to write to the database:

```
public int writeRecord (string gmfunction)
{
	if(_nvpairs.Count <= 1)
	{
		return 999; // Nothing to add!
	}
	else
	{
```

Now you'll need to instantiate 2 separate connection, datareader and command objects because the one you use to create the name/value pair container (using the actual call to GMW_NV_Create) HAS TO STAY OPEN or the temp table used to store the N/V pairs collection will be deleted as a result of the connection being closed.

```
//Connection String found in web.config
      SqlConnection myConnection = new
      SqlConnection(ConfigurationSettings.AppSettings["connectionString"
      ]);
      SqlConnection myConnection2 = new
      SqlConnection(ConfigurationSettings.AppSettings["connectionString"
      ]);
      SqlDataReader dr;
      SqlDataReader dr2;
      SqlCommand myCommand;
      SqlCommand myCommand2;
```

First we call GMW_NV_Create which will create the N/V Pairs container (the ##temptable in tempdb)

```
      myCommand=new SqlCommand();
      myCommand.Connection=myConnection;
      myCommand.CommandText="GMW_NV_Create";
      myCommand.CommandType = CommandType.StoredProcedure;

      // Add Parameters to SPROC

      SqlParameter pgmnv=new SqlParameter("@gmnv",SqlDbType.VarChar,20);
      pgmnv.Direction=ParameterDirection.Output;
      myCommand.Parameters.Add(pgmnv);


      // Return Value

      SqlParameter preturn=new
      SqlParameter("@returnval",SqlDbType.VarChar,20);
      preturn.Direction=ParameterDirection.ReturnValue;
      myCommand.Parameters.Add(preturn);

      // Open the database connection and execute the command

      myConnection.Open();
      dr = myCommand.ExecuteReader(CommandBehavior.Default);
      this._gmnv_handle=pgmnv.Value.ToString();
```

A couple of things to note. First, you'll notice that the only parameter (other than the return value) is the N/V Container handle that is retuned via the @gmnv parameter. I tuck this away in a private class variable because we'll need it for just about every call we make the GMSPROCS from here on out.

Now that we have created the N/V pair container, we need to populate it with our N/V Pairs. This is where we need our second connection, datareader and command objects because, the minute we close the one we used to call GMW_NV_Create, our N/V Pair container in the tempdb database will be deleted.

```
//Change to the 2nd Connection
myConnection2.Open();
myCommand2=new SqlCommand();
myCommand2.Connection=myConnection2;
myCommand2.CommandText="GMW_NV_SetValue";
myCommand2.CommandType=CommandType.StoredProcedure;
```

Now we just loop through our ArrayList object, extract and split the pairs back apart, adding each name value pair to the N/V Pair container as we go.

```
IEnumerator nvEnum = _nvpairs.GetEnumerator();
while (nvEnum.MoveNext())
{
      this._fieldValue=nvEnum.Current.ToString().Split('|');
      this._field=this._fieldValue[0].ToString();
      this._value=this._fieldValue[1].ToString();
      //Clear params
      myCommand2.Parameters.Clear();

      // Add Parameters to SPROC - requires three params - GMNV
      Handle(@gmnv), Fieldname(@name) and Value(@value)
      SqlParameter pgmnv2=new
      SqlParameter("@gmnv",SqlDbType.VarChar,20);
      pgmnv2.Direction=ParameterDirection.Input;
      pgmnv2.Value=this._gmnv_handle.ToString();
      myCommand2.Parameters.Add(pgmnv2);

      SqlParameter pfield=new
      SqlParameter("@fieldname",SqlDbType.VarChar, 20);
      pfield.Direction=ParameterDirection.Input;
      pfield.Value=this._field;
      myCommand2.Parameters.Add(pfield);

      SqlParameter pvalue=new
      SqlParameter("@value",SqlDbType.VarChar, 255);
      pvalue.Direction=ParameterDirection.Input;
      pvalue.Value=this._value;
      myCommand2.Parameters.Add(pvalue);

      // Return Value
      SqlParameter preturn2=new
      SqlParameter("@returnval",SqlDbType.VarChar,20);
      preturn2.Direction=ParameterDirection.ReturnValue;
      myCommand.Parameters.Add(preturn2);

      //Set_NV_Value returns:
      //-120 Returned if the gmnv passed is invalid
      //1 .. N     Returns the index number of the newly inserted
      or updated NV pair.

      // Open the database connection and execute the command
```

```
                dr2 = myCommand2.ExecuteReader(CommandBehavior.Default);
                if ((int)preturn.Value<0)
                        {
                                return 998; // Bad GMNV Handle
                        }// End if

                dr2.Close();


        }// End ArrayList while loop
```

Now you just call whatever business logic method was passed in the method call (and hope the author of the calling code has read the GMSPROCS manual)

```
                myCommand2.CommandText=gmfunction;
                myCommand2.Parameters.Clear();

                // Add Parameters to SPROC - requires one - GMNV Handle(@gmnv)
                SqlParameter pgmnv3=new SqlParameter();
                pgmnv3.ParameterName="@gmnv";
                pgmnv3.SqlDbType=System.Data.SqlDbType.VarChar;
                pgmnv3.Size=20;
                pgmnv3.Direction=ParameterDirection.Input;
                pgmnv3.Value=this._gmnv_handle.ToString();
                myCommand2.Parameters.Add(pgmnv3);

                // Return Value
                SqlParameter preturn3=new SqlParameter();
                preturn3.ParameterName="@retval";
                preturn3.SqlDbType=SqlDbType.Int;
                preturn3.Direction=ParameterDirection.ReturnValue;
                myCommand2.Parameters.Add(preturn3);



                dr2 = myCommand2.ExecuteReader(CommandBehavior.Default);

                //Cleanup
                dr.Close();
                dr2.Close();
                myConnection.Close();
                myConnection2.Close();

                return (int) preturn.Value;



                }// End if then

        }// end writeRecord method
```

The return value (an Integer) is just whatever the SPROCS are sending back to the wrapper, in other words, errors thrown by the GMSPROCS API Business Logic calls are just passed back to the original caller, therefore they need to be handled in the calling code.

Clearly, this is not something you'd want to rollout in an environment where you will have programmers calling your class who don't understand the GMSPROCS API because all of the thinking and planning still needs to be done before this wrapper is used. It doesn't do any data validation or check for errors in the N/V Pairs collection, that's left up to the caller, however, it's perfect in situations where you're working alone, or with someone else who is willing to read the GMSPROCS manual as it eliminates all the busywork.

Calling the Wrapper.

Calling the wrapper is pretty straight forward. Here's the simplest example I can think of:

```
writeGMRecord thisrecord=new writeGMRecord("WEBSITE");
returnval=thisrecord.adnvPair("Company",”Bob’s Bait and Tackle”);
returnval=thisrecord.adnvPair("Contact",”Bob Smith”);
returnval=thisrecord.adnvPair("Phone1",”7705555555”);
returnval=thisrecord.adnvPair("email",Bob@bobsbt.com);
returnval=thisrecord.adnvPair("Source","Website");
returnval=thisrecord.adnvPair("Key5","Palm Giveaway");
returnval=thisrecord.writeRecord("GMW_WriteContact");
```

And that's it. Of course, since we're not doing anything fancy, the wrapper supports all the functionality of the GMSPROCS. If you send an accountno or RECID, it will update the contact with the N/V Pairs rather than create a new one.

An example using the wrapper to update a record.

```
writeGMRecord thisrecord=new writeGMRecord("WEBSITE");
returnval=thisrecord.adnvPair("RECID",”1234##5ER(0?)(3””);
returnval=thisrecord.adnvPair("Key5","Rods and Reels");
returnval=thisrecord.adnvPair("UMYFIELD","Swan Lake");
returnval=thisrecord.writeRecord("GMW_WriteContact");
```

Writing an Activity:

```
writeGMRecord thisrecord=new writeGMRecord("WEBSITE");
returnval=thisrecord.adnvPair("Ref",”Call New Prospect!”);
returnval=thisrecord.adnvPair("OnDate",”09/25/2003”);
returnval=thisrecord.adnvPair("userid",”Linda”);
returnval=thisrecord.adnvPair("OnTime",”17:30:00”);
returnval=thisrecord.adnvPair("RECTYPE","C");
returnval=thisrecord.adnvPair("Notes","Inquire about worms");
returnval=thisrecord.writeRecord("GMW_WriteSchedule");
```

And so on.

The Wrap-up:

I find this little wrapper very handy when creating ad-hoc forms and things on our company website. It makes adding or changing contact records in GoldMine a snap as long as I don't break any of the GMSPROC rules. You could easily extend this to include more robust data validation routines in addition to better error handling and logging, however, for me, simpler is better as I'm familiar enough with the base API to make it pretty foolproof.

You can download the original class file from our download center which I recommend due to the funky formatting changes Microsoft Word makes to the text etc.

Good Luck!

Russell Smallwood is CEO of Relatia Software Corporation, authors of the popular Relatia Time and Billing GoldMine add-on. Relatia also provides custom programming services to GoldMine VARs across the US.