Client-Side Logging in Silverlight

Many of us have implemented logging in our ASP.NET, Windows Forms and WPF applications, so why shouldn't you do the same in your Silverlight applications? Well, you should. In this blog post I will show you one approach on how you might perform this logging. The class I will use is called PDSALoggingManager. This class has a method named Log() you use to publish data into a log file in your Silverlight application. A method named LogException() is also available for logging information about any exceptions that happen on the client-side of your Silverlight application. Let's take a look at the usage of the PDSALoggingManager class.

Logging Data

The simplest way to log information using the PDSALoggingManager class is to call the Log() method with some string data as shown below:

```
PDSALoggingManager.Instance.Log("Some data to log");
```

This will add the string passed to the Log() method to an internal StringBuilder object that contains the log information followed by a NewLine character. The Log() method also writes the string to a file located in isolated storage. What is written for each piece of data passed to the Log() method is shown here:

```
'Informational' log entry written on 5/22/2012 5:51:48 AM,
from class: 'SL_Log.MainPage'
   Some Data To Log
```

If you set the LogSystemInfo property on the PDSALoggingManager class prior to calling Log(), then system information is written to the log at the same time as the log data. Below is a sample of the log data with the system information appended to the end.

```
----------------------------------------------------------
'Informational' log entry written on 5/22/2012 5:51:48 AM,
from class: 'SL_Log.MainPage'
   Some Data To Log
System Information
   DateTime=5/22/2012 5:51:48 AM
   Current URL=file:///D:/MyStuff/BlogEntries/2012/
     Samples/SL-Log/SL-Log/Bin/Debug/SL_LogTestPage.html
   OSVersion=Microsoft Windows NT 6.1.7601 Service Pack 1
   OSName=Windows 7
   CurrentAssemblyName=PDSA.Silverlight, Version=5.0.0.0,
       Culture=neutral, PublicKeyToken=null
   MainAssemblyName=SL-Log, Version=1.0.0.0, Culture=neutral,
       PublicKeyToken=null
   AppDomainName=Silverlight AppDomain
   UserLanguage=en-US
   CompanyName=PDSA, Inc.
   ProductName=Silverlight Logging
   Description=Silverlight Logging
   Title=Silverlight Logging
   Copyright=Copyright © 2012 by PDSA, Inc.
   ApplicationVersion=1.0.0.0
   Stack Trace={LogInfoSample,btnLogInfo_Click}
------------------------------------------------------------
```

The system information added to the end of the log comes from another class called PDSASystemInfo. Note that I blogged about this class earlier, so check out my previous blog entry at http://weblogs.asp.net/psheriff/archive/2012/05/20/retrieve-system-information-in-silverlight.aspx for information on this class and how that data was gathered.

Passing Extra Data to Log

You have an additional overload on the Log() method that takes a generic Dictionary<string, string> object you load with key/value pairs of data. Call this version of Log() like the following:

```
Dictionary<string, string> extra =
   new Dictionary<string, string>();

extra.Add("CustomerId", "1");
extra.Add("StateCode", "CA");

PDSALoggingManager.Instance.Log("Some data to log", extra);
```

Passing a key/value pair passed into the Log() method you can add any amount of extra data you want to your log very simply. When this log entry is written, you end up with an entry that looks like the following:

```
'Informational' log entry written on 5/22/2012 5:53:47 AM,
from class: 'SL_Log.MainPage'
   Some data to log
Extra Values Passed In From Application
   CustomerId=1
   StateCode=CA
```

The Log Method

Let’s now take a look at the Log() method in the PDSALoggingManager class. When you pass in a single string value, the following version of the Log() method is called.

```
public void Log(string value)
{
  this.Log(value, "Informational", null);
}
```

This method calls another overload of the Log() method to which you can pass in the “type” of log entry you are writing and a null in place of the Dictionary object. By default, the Log() method uses “Informational” as the log type. You can pass in whatever value you wish for this type.

The 2nd overload of this method is the one that you pass in the Dictionary object to. This method calls the same Log() method as the previous one, but this time passes the extra values from the Dictionary object.

```
public void Log(string value,
   Dictionary<string, string> extraValues)
{
  this.Log(value, "Informational", extraValues);
}
```

Now, let’s look at the version of the Log() method that does the actual work of logging the string data, the type and optionally the extra dictionary values you might pass in. The first thing Log() does is to call a method named GetCallingClassName(). This method will be shown later, but it is used to retrieve the name of the method in your Silverlight application that called the Log() method. Next, it calls a method named Format() that will format the log data into what you saw earlier in this blog. Finally, the entry is written to isolated storage.

```
public void Log(string value, string logType,
  Dictionary<string, string> extraValues)
{
  IsolatedStorageFile file = null;
  string message = string.Empty;

  try
  {
    // Get the name of the calling method
    CallingClassName = GetCallingClassName();
    // Format the log entry
    message = Format(value, logType, extraValues);

    file = IsolatedStorageFile.GetUserStoreForSite();
    using (IsolatedStorageFileStream fs = new
       IsolatedStorageFileStream(LogFileName,
         FileMode.Append, file))
    {
      using (StreamWriter sw = new StreamWriter(fs))
      {
        sw.WriteLine(message);
      }
    }
  }
  catch (Exception ex)
  {
    TheLog.Append("Exception Occurred in Log() method" +
      Delimiter + ex.ToString());
  }
  finally
  {
    if (file != null)
      file.Dispose();
  }
}
```

Format Method

The Format() method is used to put the log entry into a readable format. A StringBuilder object, named sb, is what is used to format the data and concatenate all of the data together. Once all of the data is gathered up, this local StringBuilder object is appended to the 'TheLog' property. This property is kept in memory so you can retrieve the log during the running of your application without having to read the data from the isolated storage file. This property is also used to log any exception that occurs within the PDSALoggingManager class itself.

Notice there is another property called Delimiter that is used to separate each line of the log. This property is initialized in the constructor of the PDSAloggingManager class to Environment.NewLine.

```
protected virtual string Format(string value, string logType,
                Dictionary<string, string> extraValues)
{
  StringBuilder sb = new StringBuilder(512);

  if (string.IsNullOrEmpty(Delimiter))
    Delimiter = Environment.NewLine;

  if (LogSystemInfo)
    sb.Append(new string('-', 200) + Delimiter);

  sb.Append("'" + logType + "'");
  sb.Append(" log entry written on "
            + DateTime.Now.ToString());
  if (!string.IsNullOrEmpty(CallingClassName))
    sb.Append(", from class: '" + CallingClassName + "'");
  sb.Append(Delimiter);
  sb.Append("   " + value + Delimiter);
  // Add on Extra Values
  sb.Append(FormatKeyValuePairs(extraValues));
  if (LogSystemInfo)
  {
    PDSASystemInfo si = new PDSASystemInfo();

    sb.Append(si.GetAllSystemInfo(Delimiter));
    sb.Append(Delimiter);
    sb.Append(new string('-', 200) + Delimiter);
  }

  // Append to the main log property
  TheLog.Append(sb.ToString());

  return sb.ToString();
}
```

A property called LogSystemInfo is initialized to true in the constructor of this class. If set to true, then system information is gathered from the PDSASystemInfo class and appended to the log. If there are any extra values passed in the Dictionary object, those values are also appended to the log. These are formatted in the FormatKeyValuePairs() method. You can look up this method by downloading the code for this blog entry. See the end of this blog for instructions on how to get this article and the associated code sample.

GetCallingClassName Method

The PDSALoggingManager and PDSASystemInfo classes are located in a DLL named PDSA.Silverlight and is referenced from your Silverlight application. It is, of course, a best practice to put generic classes like this into a separate DLL. However, there is another benefit we derived from doing this. We want to retrieve the name of the method that called the Log() method so we can record where Log() was called from. The StackFrame object is used to retrieve each method in the stack trace. As we grab each method we can check the method to see if it is in the current assembly and if it is, we will ignore it. However, once we find a method that is in a different assembly, we can assume that this is the assembly and method that called the Log() method.

```
public string GetCallingClassName()
{
  int loop = 0;
  string ret = string.Empty;
  string currentName =
     Assembly.GetExecutingAssembly().FullName;

  try
  {
    StackFrame sf = new StackFrame(loop);
    while (sf.GetMethod() != null)
    {
      // Don't get any methods contained in this assembly.
      if (sf.GetMethod().DeclaringType.Assembly.FullName !=
          currentName)
      {
        ret = sf.GetMethod().DeclaringType.FullName;
        break;
      }

      loop++;
      sf = new System.Diagnostics.StackFrame(loop);
    }
  }
  catch
  {
    // Do nothing
  }

  return ret;
}
```

Logging Exceptions

In addition to logging string data, you might also wish to log exception data. To facilitate this I added a LogException() method. This method will accept an Exception object. By default, this method simply takes the result of the ToString() method on the exception object and passes it to the Log() method. However, you could modify this to retrieve any additional information from the Exception object that you want and pass that to the Log() method.

```
try
{
  decimal ret = 10;

  ret = ret / 0;
}
catch (Exception ex)
{
  PDSALoggingManager.Instance.LogException(ex);
}
```

A second overload of the LogException() method allows you to pass in additional information as a generic Dictionary object just like the original Log() method allows.

```
decimal ret = 10;

try
{
  ret = ret / 0;
}
catch (Exception ex)
{
  Dictionary<string, string> extra =
     new Dictionary<string, string>();

  extra.Add("StackTraceFromException", ex.StackTrace);
  extra.Add("ret variable", ret.ToString());

  PDSALoggingManager.Instance.LogException(ex, extra);
}
```

Here is the output from logging the exception:

```
'Exception' log entry written on 5/22/2012 4:10:15 PM,
  from class: 'SL_Log.MainPage'
   System.DivideByZeroException: Attempted to divide by zero.
   at System.Decimal.FCallDivide(Decimal& d1, Decimal& d2)
   at System.Decimal.op_Division(Decimal d1, Decimal d2)
   at SL_Log.MainPage.btnLogException_Click(Object sender,
     RoutedEventArgs e)
```

Getting the Log from your User

Once you have logged the data, you might want to get that information. The obvious choice would be to pass the complete log information to a WCF service. I am not presenting that choice in this sample due to space. However, one thing you need to consider is what if your user cannot access the WCF service for some reason? In this case you should have a backup plan. Two options you might consider are one, give the user a button they can click on that will copy the log to the clipboard, and two, another button that they can use to save the log data to a file on their computer. First, here is the code you would write to copy the log to the clipboard.

```
try
{
  Clipboard.SetText(PDSALoggingManager.Instance.ReadLog());
}
catch
{
  MessageBox.Show("Can't copy to the Clipboard.");
}
```

The ReadLog() method returns the data from the TheLog property, or if that is empty, will read the data from the isolated storage file. Next, you could read the log data and pass that to a method that will prompt the user to enter the name and location on their hard drive where to store the log data.

```
private void btnSaveToFile_Click(object sender,
 RoutedEventArgs e)
{
  SaveToFile(PDSALoggingManager.Instance.ReadLog());
}

private void SaveToFile(string contents)
{
  SaveFileDialog sfd = null;

  try
  {
    sfd = new SaveFileDialog();
    sfd.DefaultExt = "txt";
    sfd.Filter = "Log Files (*.log)|*.log|
                  All Files (*.*)|*.*";
    sfd.FilterIndex = 1;

    bool? result = sfd.ShowDialog();

    if (result.HasValue && result == true)
    {
      using (StreamWriter sw =
              new StreamWriter(sfd.OpenFile()))
      {
        sw.Write(contents);
        sw.Close();
      }
    }
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

After storing this file on their hard drive, they could then attach that file to an email and send the log data to you.

Summary

Implementing a logging system in your Silverlight application is a great way to keep track of what you user does in your application. It is also extremely useful for tracking down errors. You must still be able to get the log file from the user, but that is fairly easy using either the clipboard or saving to a file and having your user email you the log. Hopefully this simple little class will give you a head-start on creating your own logging system.

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Tips & Tricks”, then select “Client-Side Logging in Silverlight” from the drop down list.