Semantic Logging Application Block Hands-On Lab for Enterprise Library

patterns & practices
proven practices for predictable results

This walkthrough should act as your guide for learning about the Enterprise Library Semantic Logging Application Block and will allow you to practice employing its capabilities in various application contexts.

The term *semantic logging* refers specifically to the use of strongly typed events and the consistent structure of the log messages in the Semantic Logging Application Block.

With the Semantic Logging Application Block you are simply reporting the fact that some event occurred that you might want to record in a log. You do not need to specify an event ID, a priority, or the format of the message when you create the event to log in your application code. This approach of using strongly typed events in your logging process provides the following benefits:

- You can be sure that you format and structure your log messages in a consistent way because there is no chance of making a coding error when you write the log message. For example, an event with a particular ID will always have the same verbosity, extra information, and payload structure.
- It is easier to query and analyze your log files because the log messages are formatted and structured in a consistent manner.
- You can more easily parse your log data using some kind of automation. This is especially true if you are using one of the Semantic Logging Application Block sinks that preserves the structure of the log messages; for example, the database sinks preserve the structure of your log messages. However, even if you are using flat files, you can be sure that the format of log messages with a particular ID will be consistent.
- You can more easily consume the log data from another application. For example, in an application that automates activities in response to events that are recorded in a log file or a Windows Azure Table.
- It is easier to correlate log entries from multiple sources.

After completing this lab, you will be able to do the following:

- You will be able to create and use an Event Source class.
- You will be able to use the Enterprise Library Semantic Logging Application Block to implement logging in an application.
- You will be able to use the Enterprise Library Semantic Logging Application Block to write logs to multiple destinations, including Windows Azure Storage.

This hands-on lab includes the following labs:

- Lab 1: Creating and Using an Event Source
- Lab 2: Sending Event Messages to Windows Azure Storage
- Lab 3: Testing an Event Source

The estimated completion for this lab is **30 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Program Management: Grigori Melnik and Madalyn Parker (Microsoft Corporation)
- Development/Testing: Madalyn Parker, Julian Dominguez , Grigori Melnik, Mike Sampson and Tyler Ohlson (Microsoft Corporation), Fernando Simonazzi (Clarius Consulting), Mariano Grande (Digit Factory), and Naveen Pitipornvivat (Adecco)
- Documentation: Madalyn Parker, Fernando Simonazzi (Clarius Consulting), Grigori Melnik, Nelly Delgado and RoAnn Corbisier (Microsoft Corporation)

Lab 1: Creating and Using an Event Source

Introduction

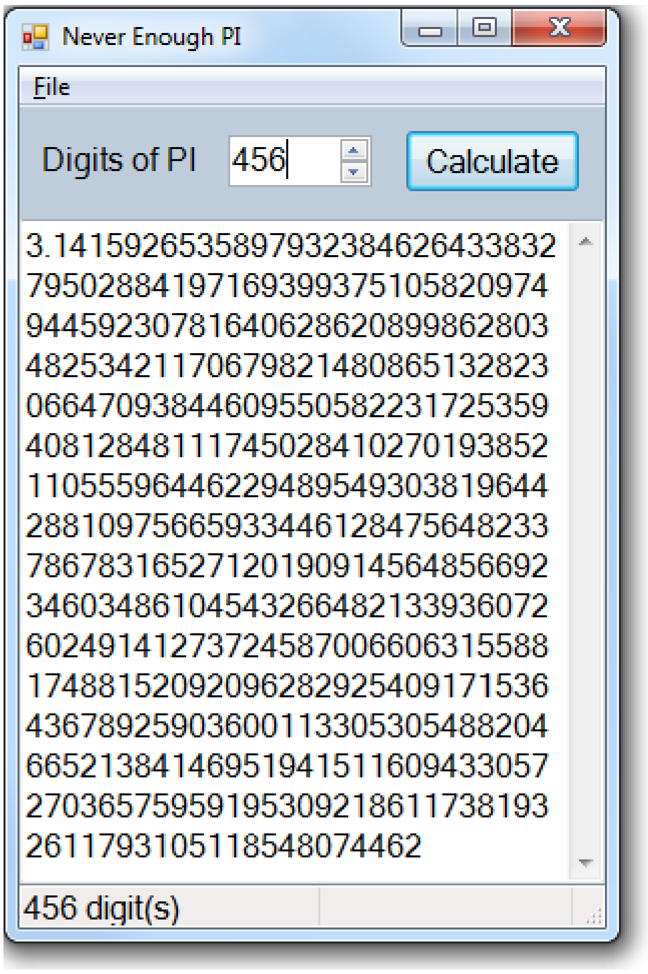
In this lab, you will create a custom class that extends the **EventSource** class in the **System.Diagnostics.Tracing** namespace. Then you will use your **CalculatorEventSource** to easily incorporate semantic logging into your application.

To begin this exercise, open the EnoughPI.sln file located in the Lab01\begin folder.

To learn about the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.

   The **EnoughPI** application calculates the digits of pi (π, the ratio of the circumference of a circle to its diameter). Enter your desired precision via the **NumericUpDown** control and click the **Calculate** button. Be prepared to wait if you want more than 500 digits of precision.



```
3.1415926535897932384626433832
79502884197169399375105820974
94459230781640628620899862803
48253421170679821480865132823
06647093844609550582231725359
40812848111745028410270193852
11055596446229489549303819644
28810975665933446128475648233
78678316527120190914564856692
34603486104543266482133936072
60249141273724587006606315588
17488152092096282925409171536
43678925903600113305305488204
66521384146951941511609433057
27036575959195309218611738193
2611793105118548074462
```

456 digit(s)

Task 1: Creating an Event Source

Creating an Event Source

You can specify the log messages you will write by extending the **EventSource** class in the **System.Diagnostics.Tracing** namespace in the .NET Framework 4.5. The events generated by this **EventSource** class have no dependency on the Semantic Logging Application Block. However, when you utilize the Semantic Logging Application Block to handle to them, the custom **EventSource** enables you to quickly create clear, easily-consumable logs based on specific events.

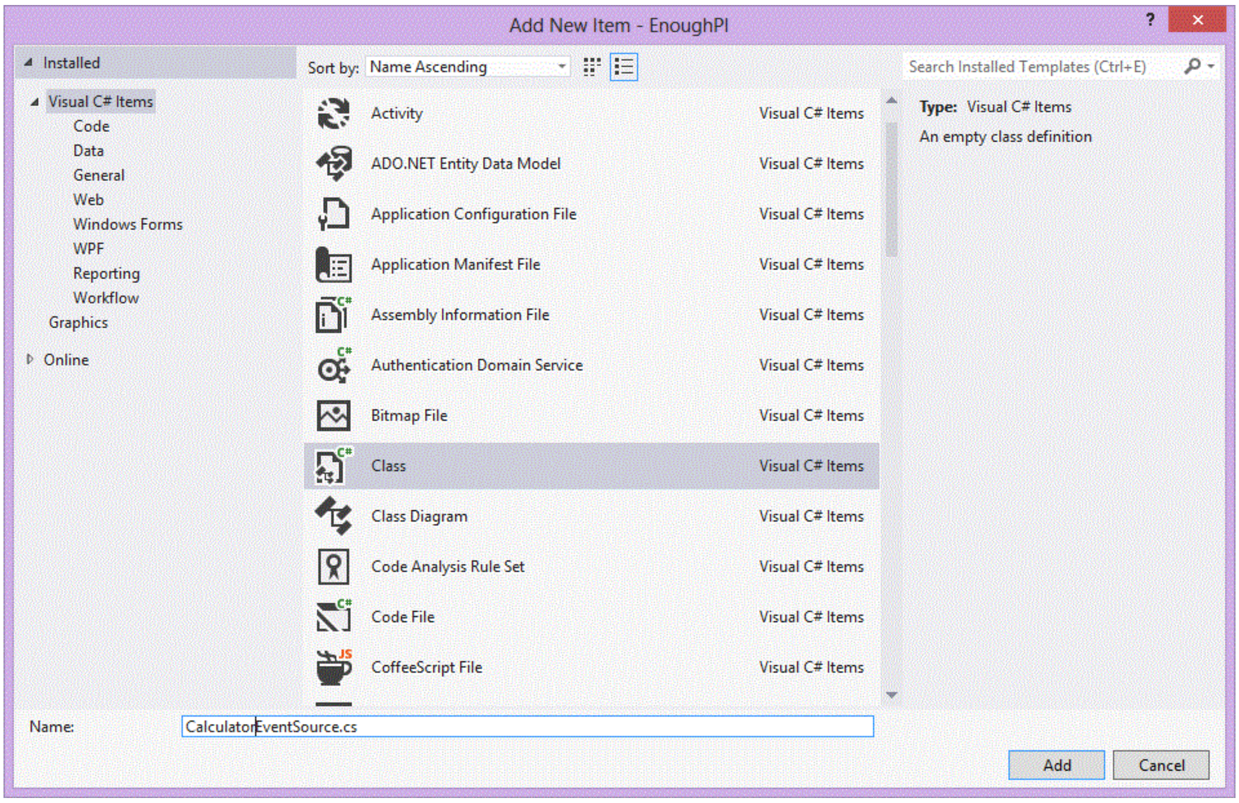
The **EventSource** class uses custom event methods to write events. Each custom event method determines the information that is written to the log, and specifies a log level and keywords for the event type.

The **EventSource** notifies any event listeners that are enabled to receive events that match certain filter criteria. An event listener can specify that it will receive events at or below a given log level, it can also specify that it will receive events that match certain keywords. The **ObservableEventListener,** which we will explore in the next lab, receives the event if it matches its filter criteria.

An event sink subscribes to the **ObservableEventListener**, formats the message using a formatter, and writes the message to output specific to that event sink.

To create an event source

1. Select the **EnoughPI** project in Solution Explorer. Select the **Project | Add Class** menu command. Call this class **CalculatorEventSource**.



Type: Visual C# Items

An empty class definition

Name: CalculatorEventSource.cs

2. Add the following namespace inclusions at the top of the file:

```
using System.Diagnostics.Tracing;
```

3. Make this class a **public** class derived from **EventSource**.

```
public class CalculatorEventSource : EventSource
{
}
```

4. Add the following members to your class.

```
private static readonly Lazy<CalculatorEventSource> Instance = new
Lazy<CalculatorEventSource>(() => new CalculatorEventSource());

private CalculatorEventSource() { }
public static CalculatorEventSource Log
{
    get { return Instance.Value; }
}
```

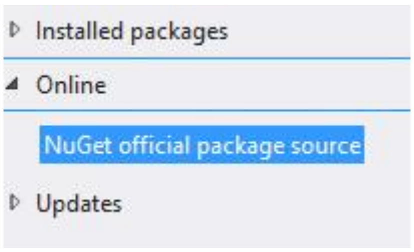
Here, you create a static field called **Instance** that provides access to an instance of the **CalculatorEventSource** class. Also, you create a static property called **Log** that returns the current value of the **Instance** field of the event source. This value is determined by the custom event methods called in your **Calculator** application.

Task 2: Using the Event Source to log Events

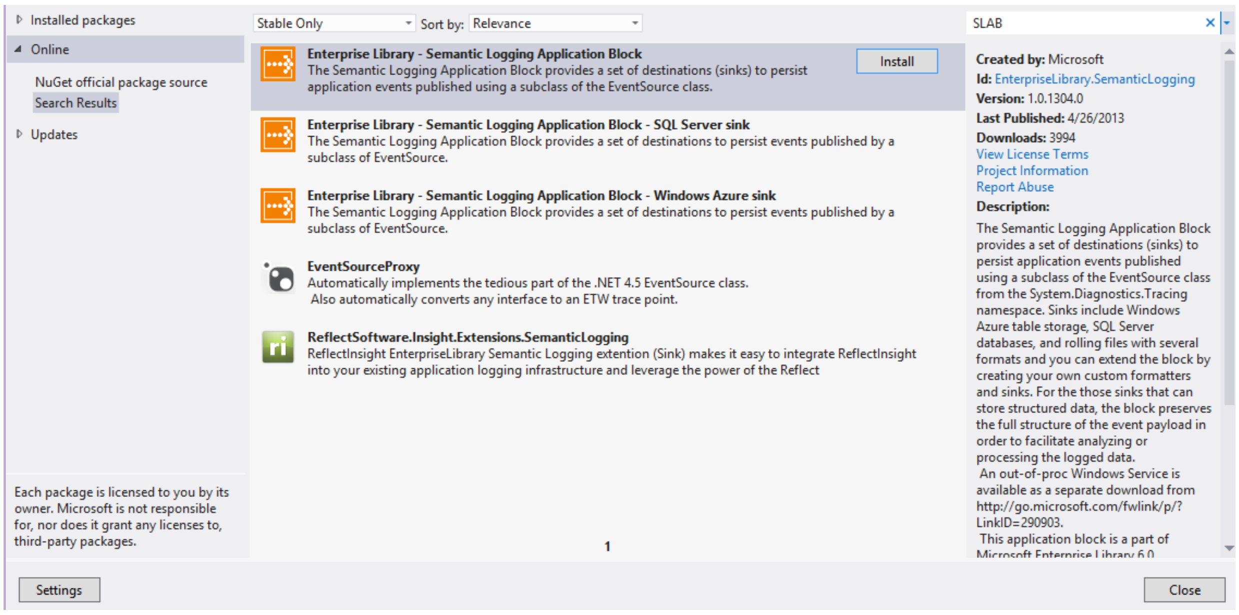
Adding References to the Required Assemblies

To add the necessary NuGet packages

1. Select the **EnoughPI** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the “Online” option to view NuGet packages available online.



3. Search for **SLAB** in the search bar. Select **Enterprise Library – Semantic Logging Application Block** and click install.



Enterprise Library - Semantic Logging Application Block
The Semantic Logging Application Block provides a set of destinations (sinks) to persist application events published using a subclass of the EventSource class.

Enterprise Library - Semantic Logging Application Block - SQL Server sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.

Enterprise Library - Semantic Logging Application Block - Windows Azure sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.

EventSourceProxy
Automatically implements the tedious part of the .NET 4.5 EventSource class.
Also automatically converts any interface to an ETW trace point.

ReflectSoftware.Insight.Extensions.SemanticLogging
ReflectInsight EnterpriseLibrary Semantic Logging extention (Sink) makes it easy to integrate ReflectInsight into your existing application logging infrastructure and leverage the power of the Reflect

Created by: Microsoft
Id: EnterpriseLibrary.SemanticLogging
Version: 1.0.1304.0
Last Published: 4/26/2013
Downloads: 3994
View License Terms
Project Information
Report Abuse
Description:
The Semantic Logging Application Block provides a set of destinations (sinks) to persist application events published using a subclass of the EventSource class from the System.Diagnostics.Tracing namespace. Sinks include Windows Azure table storage, SQL Server databases, and rolling files with several formats and you can extend the block by creating your own custom formatters and sinks. For the those sinks that can store structured data, the block preserves the full structure of the event payload in order to facilitate analyzing or processing the logged data.
An out-of-proc Windows Service is available as a separate download from http://go.microsoft.com/fwlink/p/?LinkID=290903.
This application block is a part of Microsoft Enterprise Library 6.0

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

4. Click **Accept** on the License Acceptance window.

Adding an Event Listener and Logging to the Application

To add an event listener and logging

1. Open the **EntryPoint.cs** file and add the following namespace inclusions.

```
using Microsoft.Practices.EnterpriseLibrary.SemanticLogging;
using
Microsoft.Practices.EnterpriseLibrary.SemanticLogging.Formatters;
using System.Diagnostics.Tracing;
```

2. Add the highlighted code to the **Main** method to setup and enable the Event Listener.

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionEventHandler(
          Application_ThreadException);

    //TODO: Set up and enable the event listener
    using (var listener = new ObservableEventListener())
    {
        listener.LogToConsole();
        listener.EnableEvents(CalculatorEventSource.Log,
EventLevel.LogAlways, Keywords.All);

        Form entryForm = new MainForm();
        Application.EnableVisualStyles();
        Application.Run(entryForm);
    }
}
```

Typically, you create an **ObservableEventListener** instance to receive the events from your event source, and then you create any sinks you need to save the log messages. The built-in sinks have constructors that enable you to configure the sink as you create it. The block includes convenience extension methods, such as **LogToConsole** and **LogToSqlDatabase**, to set up the sinks and attach them to a listener. When you enable the listener, you specify the event source to use, the highest level of event to capture, and any keywords to filter on.

Note: Disposing an **EventListener** disables it for any **EventSource** it might have been enabled for, and in the case of the **ObservableEventListener** and the sinks in the Semantic Logging Application Block, it also causes any events that might have been buffered to be flushed to storage.

3. You want to log the calculation completion in **Calculator.cs** so add the following code to use the **CalculatorEventSource** to log the relevant information. Here you will pass the **Digits** argument, which tells how many digits of pi were calculated.

```
protected void OnCalculated(CalculatedEventArgs args)
{
     // TODO: Log final result
     CalculatorEventSource.Log.Calculated(args.Digits);

     if (Calculated != null)
        Calculated(this, args);
}
```

This will create an error since there is no **Calculated** method in the **CalculatorEventSource** yet.

4. Right-click on **Calculated**, and select **Generate | Method Stub**

This will create a method with a default implementation in the CalculatorEventSource class.

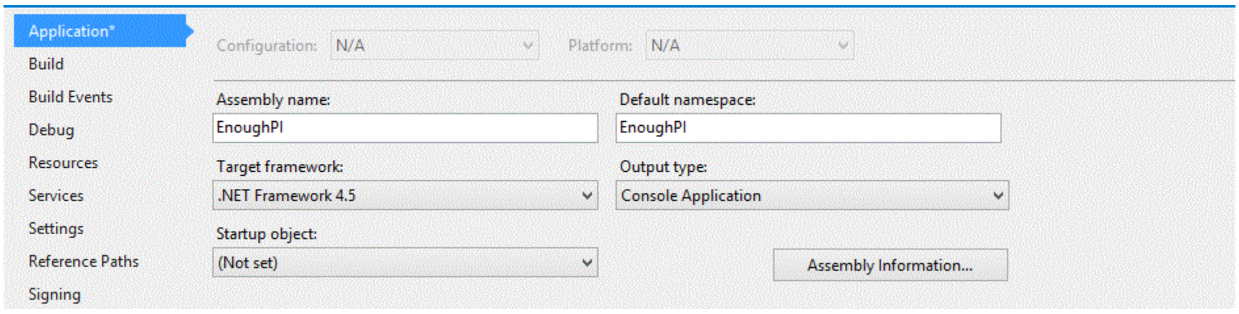
5. Update the **Calculated** method as follows in **CalculatorEventSource.cs**.

```
[Event(100)]
internal void Calculated(int digits)
{
    this.WriteEvent(100, digits);
}
```

Note: This is the simplest form of the event method. After this small amount of work, you can again focus on the business logic and not worry about formatting, log consistency, categories, etc. Later in this exercise you will specialize the event methods to provide more custom logs that are readable and easily consumable.

To run the application

1. Select the **EnoughPI** project. Select the **Project | EnoughPI Properties…** menu command, select the **Application** tab, and set **Output type** to **Console Application**. This way you can view the logs sent to the Console.


Application*
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing

Configuration: N/A   Platform: N/A

Assembly name: EnoughPI
Default namespace: EnoughPI
Target framework: .NET Framework 4.5
Output type: Console Application
Startup object: (Not set)

2. Select the **File | Save All** menu command.
3. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button.
4. View the application console log for messages from the **EnoughPI** source. There will be one log detailing that the calculation of pi has finished.
5. Exit the application.

Adding Logs for the Other Events

To add a log

1. Log the calculation progress by adding the following code to the **OnCalculating** method in the **Calculator.cs** file. Here, you would like the log to include the digit of pi you are calculating, the current value of pi, and whether the calculator was canceled.

```
protected void OnCalculating(CalculatingEventArgs args)
{
  // TODO: Log progress
  CalculatorEventSource.Log.Calculating(args.StartingAt, args.Pi,
      args.Cancel);

  if (Calculating != null)
    Calculating(this, args);

  if (args.Cancel == true)
  {
    // TODO: Log cancellation

  }
}
```

2. Right-click **Calculating** and select **Generate | Method Stub**.
3. In **CalculatorEventSource.cs** update the **Calculating** method to include the parameters used in the call made in **Calculator.cs**. Also, assign it a different event id than used in the **Calculated** event method.

```
[Event(101)]
internal void Calculating(int digit, string pi, bool cancelled)
{
    this.WriteEvent(101, digit, pi, cancelled);
}
```

Each event type in your application is represented by a method in your **EventSource** implementation. These event methods take parameters that define the payload of the event and are decorated with attributes that provide additional metadata such as the event ID or verbosity level. The payload can be nothing at all or can have detailed information. An event's message is a human readable version of the payload information whose format you can control using the **Event** attribute's **Message** parameter.

4. Each event is defined using a method that wraps a call to the **WriteEvent** method in the **EventSource** class. The first parameter of the **WriteEvent** method is an event id that must be unique to that event, and different overloaded versions of this method enable you to write additional information to the log. Attributes on each event method further define the characteristics of the event. Notice that the **Event** attribute includes the same value as its first parameter. Log the cancellation of calculations.

```
protected void OnCalculating(CalculatingEventArgs args)
{
     // TODO: Log progress
     CalculatorEventSource.Log.Calculating(
       args.StartingAt,
       args.Pi,
       args.Cancel);

     if (Calculating != null)
        Calculating(this, args);

    if (args.Cancel == true)
    {
      // TODO: Log cancellation
      CalculatorEventSource.Log.CalculationCanceled();
    }
}
```

5. Again, right click the event method name and select **Generate | Method Stub**. Add the following code to the **CalculationCanceled** to the **CalculatorEventSource** class.

```
[Event(102)]
internal void CalculationCanceled()
{
    this.WriteEvent(102);
}
```

6. Log calculation exceptions by adding the following code to the **OnCalculatorException** method in the **Calculator.cs** file.

```
protected void OnCalculatorException(CalculatorExceptionEventArgs
args)
{
  // TODO: Log exception
  CalculatorEventSource.Log.CalculatorException(
    args.Exception.ToString());

  if (CalculatorException != null)
      CalculatorException(this, args);
}
```

You are passing the exception as a string because **Exception** is not a type supported for event methods by the **EventSource** class.

7. Generate a method stub from this and update the **CalculatorException** method with the highlighted code below.

```
[Event(103)]
internal void CalculatorException(string exception)
{
    this.WriteEvent(103, exception);
}
```

Note: Passing the Exception as a string is not ideal, as you have no control over the format or information passed. It is much better to write a method in your **EventSource** class to handle the formatting of Exceptions so they are consistent and as verbose as you would like.

To format exceptions

1. Create a new method in your **CalculatorEventSource** class to format exceptions. Have it take the Exception as a parameter, and return a string that includes the Inner Exception and Message. This should be **public** so it can be called from your **Calculator** class.

```
public static string FormatException(Exception exception)
{
    return exception.GetType().ToString() + Environment.NewLine +
exception.Message;
}
```

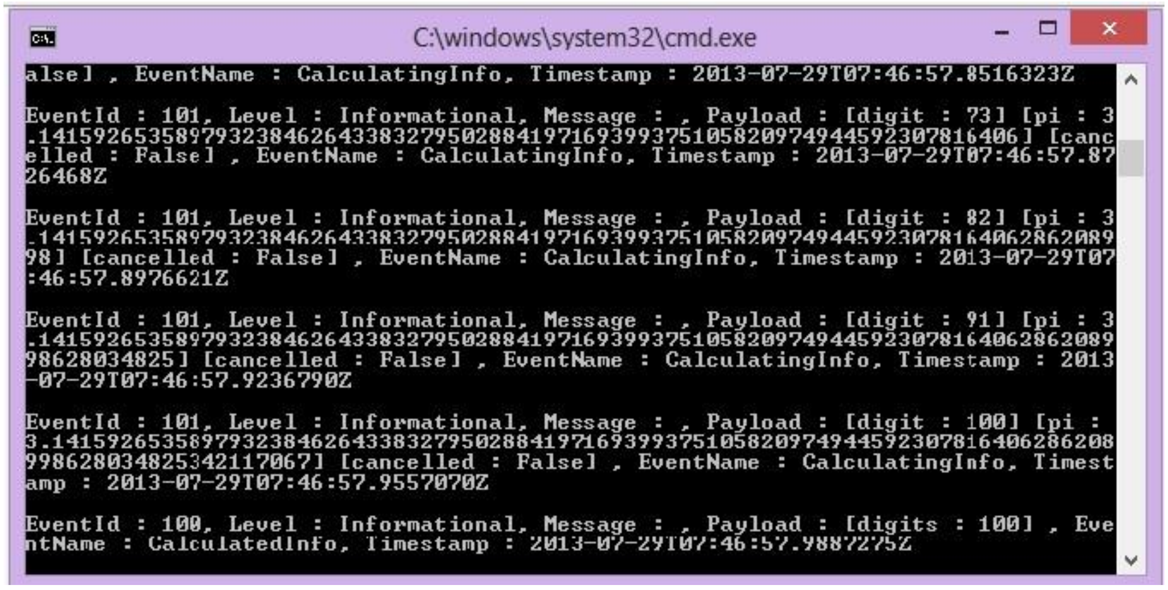
2. Call this in **Calculator.cs** and pass the formatted exception to the **CalculatorException** method in **CalculatorEventSource.**

```
protected void OnCalculatorException(
    CalculatorExceptionEventArgs args)
{
    // TODO: Log exception
    CalculatorEventSource.Log.CalculatorException(
       CalculatorEventSource.FormatException(args.Exception));

    if (CalculatorException != null)
        CalculatorException(this, args);
}
```

Note: An alternative to have the calling code format an exception, or any other object, before invoking the event method in the **EventSource** class is to define a non-private non-event method, which the application code will invoke passing the unformatted objects as parameters, and a private event method, which takes the formatted objects as payload and is invoked by the non-event method after formatting the objects. With this approach application code is not involved in the formatting, but the implementation of the **EventSource** is more complex because of the additional methods.

To run the application

1. Rebuild the solutionSelect the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision and click the **Calculate** button.
2. View the application console log for messages from the **EnoughPI** source. There will be logs from all events executed.



```
alse] , EventName : CalculatingInfo, Timestamp : 2013-07-29T07:46:57.8516323Z

EventId : 101, Level : Informational, Message : , Payload : [digit : 73] [pi : 3
.1415926535897932384626433832795028841971693993751058209749445923078164061 [canc
elled : False] , EventName : CalculatingInfo, Timestamp : 2013-07-29T07:46:57.87
26468Z

EventId : 101, Level : Informational, Message : , Payload : [digit : 82] [pi : 3
.1415926535897932384626433832795028841971693993751058209749445923078164062862089
98] [cancelled : False] , EventName : CalculatingInfo, Timestamp : 2013-07-29T07
:46:57.8976621Z

EventId : 101, Level : Informational, Message : , Payload : [digit : 91] [pi : 3
.1415926535897932384626433832795028841971693993751058209749445923078164062862089
98628034825] [cancelled : False] , EventName : CalculatingInfo, Timestamp : 2013
-07-29T07:46:57.9236790Z

EventId : 101, Level : Informational, Message : , Payload : [digit : 100] [pi :
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
99862803482534211706791 [cancelled : False] , EventName : CalculatingInfo, Timest
amp : 2013-07-29T07:46:57.9557070Z

EventId : 100, Level : Informational, Message : , Payload : [digits : 100] , Eve
ntName : CalculatedInfo, Timestamp : 2013-07-29T07:46:57.9887275Z
```

In addition to the Event information, the payload (parameters passed to the **WriteEvent** method) is written to the log as well.

Task 3: Formatting Log Messages

This output is difficult to read and discern what is happening in the application since all logs are nearly identical. To remedy this, you can customize each of the event methods in your **CalculatorEventSource** class and format the logs to be more human readable.

To add a formatter

1. Add the following highlighted code to the **Main** method in **EntryPoint.cs**.

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionEventHandler(
           Application_ThreadException);

    //create formatter
    var formatter = new JsonEventTextFormatter
```

```
       (EventTextFormatting.Indented);

    //TODO: Set up and enable the event listener
    using (var listener = new ObservableEventListener())
    {
        listener.LogToConsole(formatter);
        listener.EnableEvents(CalculatorEventSource.Log,
EventLevel.LogAlways, Keywords.All);

        Form entryForm = new MainForm();
        Application.EnableVisualStyles();
        Application.Run(entryForm);
    }
}
```

This creates a formatter and uses it in the listener to make the text of the logs more readable and easier to parse.

To customize event methods

Each event method's **Event** attribute can include the named parameters, which can help when consuming and interpreting log messages. Some of these named parameters are Message, Level, Keywords, and Task.

1. Start by adding the **Message** parameter to your Events. These formats can be treated like those for the **String.Format** method, with variables passed in the **WriteEvent** method being used as format items in the final string.

```
[Event(100, Message="PI was calculated to {0} digits")]
internal void Calculated(int digits)
{
    this.WriteEvent(100, digits);
}

[Event(101, Message="Calculating PI from {0} digits. Current value
is {1}.")]
internal void Calculating(int digit, string pi, bool cancelled)
{
    this.WriteEvent(101, digit, pi, cancelled);
}

[Event(102, Message="Calculation canceled!")]
internal void CalculationCanceled()
{
    this.WriteEvent(102);
}

[Event(103, Message="Calculator Exception Thrown!")]
internal void CalculatorException(string exception)
{
    this.WriteEvent(103, exception);
}
```

2. Next, add Event Levels for each of the event methods. This will make events much easier to consume, enabling you to filter by level or create a listener to consume events of a particular level. Assign the **Calculated** and **Calculating** methods a **Level** of **EventLevel.Informational.** Make the **CalculationCanceled** method **EventLevel.Warning.** Finally, make the **CalculatorException** method's **Level EventLevel.Error**.

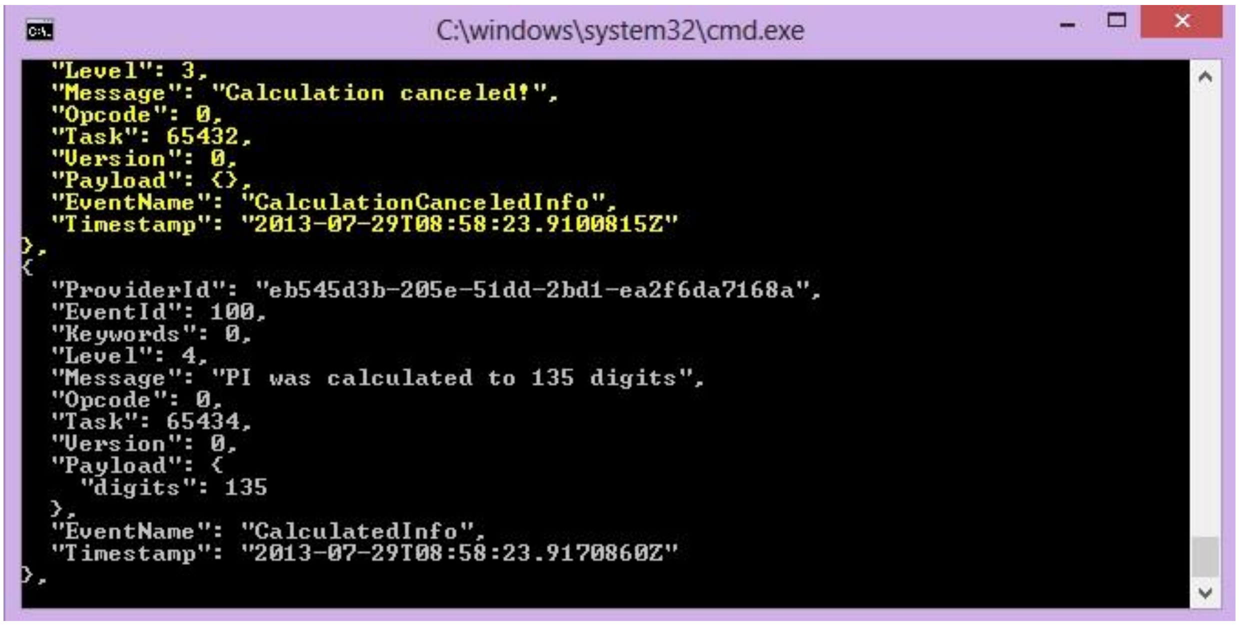
```
[Event(100, Message="PI was calculated to {0} digits", Level =
EventLevel.Informational)]
internal void Calculated(int digits)
{
    this.WriteEvent(100, digits);
}

[Event(101, Message="Calculating PI from {0} digits. Current value
is {1}.", Level = EventLevel.Informational)]
internal void Calculating(int digit, string pi, bool cancelled)
{
   this.WriteEvent(101, digit, pi, cancelled);
}
[Event(102, Message="Calculation canceled!", Level =
EventLevel.Warning)]
internal void CalculationCanceled()
{
    this.WriteEvent(102);
}

[Event(103, Message="Calculator Exception Thrown!", Level =
EventLevel.Error)]
internal void CalculatorException(string exception)
{
    this.WriteEvent(103, exception);
}
```

To view the formatted output

1. Select the **Debug | Start Without Debugging** menu command to run the application. Enter your desired precision (at least 600 would be good) and click the **Calculate** button.

2. Press the cancel button to cancel the calculation process.

   Notice that the console text is much easier to read and uses JSON formatting. Also, because the **ConsoleSink** class comes with color mapping for **EventLevels**, the cancellation is very easy to spot in the logs.



```
  "Level": 3,
  "Message": "Calculation canceled!",
  "Opcode": 0,
  "Task": 65432,
  "Version": 0,
  "Payload": {},
  "EventName": "CalculationCanceledInfo",
  "Timestamp": "2013-07-29T08:58:23.9100815Z"
},
{
  "ProviderId": "eb545d3b-205e-51dd-2bd1-ea2f6da7168a",
  "EventId": 100,
  "Keywords": 0,
  "Level": 4,
  "Message": "PI was calculated to 135 digits",
  "Opcode": 0,
  "Task": 65434,
  "Version": 0,
  "Payload": {
    "digits": 135
  },
  "EventName": "CalculatedInfo",
  "Timestamp": "2013-07-29T08:58:23.9170860Z"
},
```

st

To verify that you have completed the exercise correctly, you can use the solution provided in the Lab01\end folder.

Lab 2: Sending Event Messages to Windows Azure Storage

Introduction

In this lab you will use the Semantic Logging Application Block to log events to multiple destinations. The Semantic Logging Application Block includes sinks that enable you to save log messages to the following locations: the console, a database, a Windows Azure Table, and to flat files. The choice of which sinks to use depends on how you plan to analyze and monitor the information you are collecting.

To begin this exercise, open the EnoughPI.sln file located in the Lab02\begin folder.

Task 1: Logging to Windows Azure Table Service

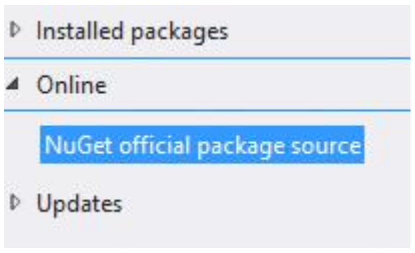
To create and configure a Windows Azure Table Service locations

1. Make sure you have the Windows Azure SDK installed.

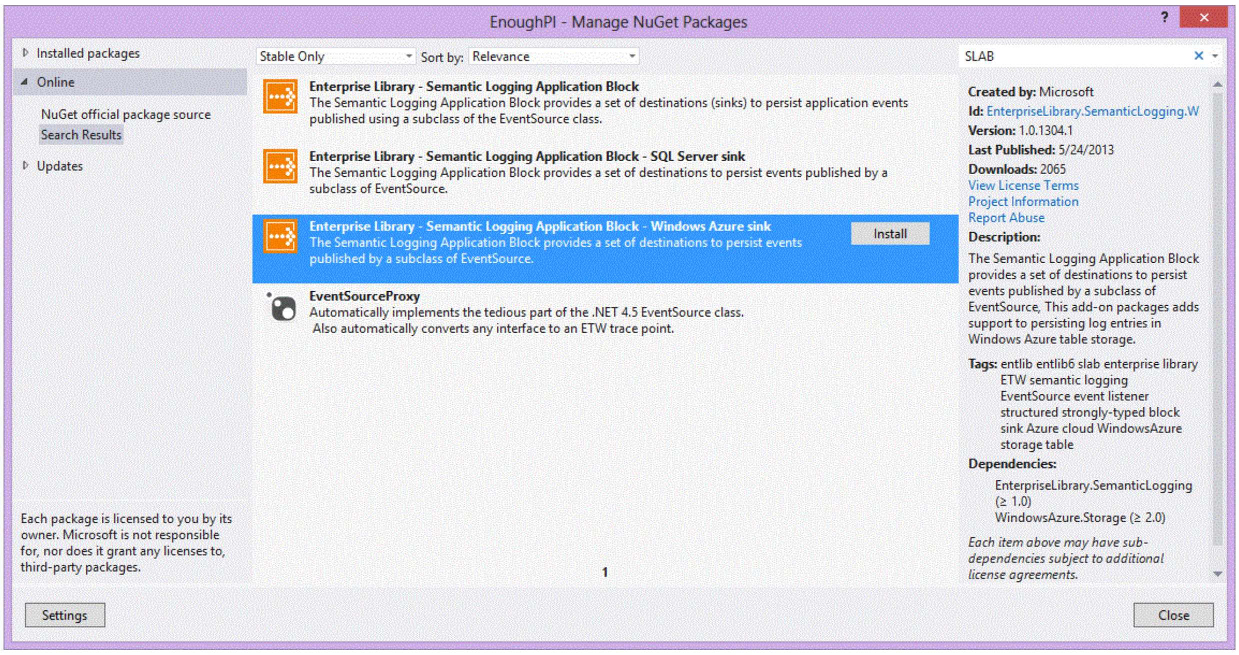
> **Note:** This exercise is configured to run using the local Windows Azure storage emulator. It is possible to target the application to a Windows Azure storage account created in Windows Azure and thus not require a dependency on the Windows Azure SDK, but it is easier to set up and observe the behavior of the application running locally, as it does not require a Windows Azure subscription.

To include the Windows Azure extension for the Semantic Logging Application Block

1. Select the **EnoughPI** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the “Online” option to view NuGet Packages available online.



3. Search for **SLAB** in the search bar. Select **Enterprise Library – Semantic Logging Application Block – Windows Azure sink** and click install.



EnoughPI - Manage NuGet Packages

Installed packages
Online
NuGet official package source
Search Results
Updates

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Stable Only | Sort by: Relevance | SLAB

Enterprise Library - Semantic Logging Application Block
The Semantic Logging Application Block provides a set of destinations (sinks) to persist application events published using a subclass of the EventSource class.

Enterprise Library - Semantic Logging Application Block - SQL Server sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.

Enterprise Library - Semantic Logging Application Block - Windows Azure sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.
Install

EventSourceProxy
Automatically implements the tedious part of the .NET 4.5 EventSource class. Also automatically converts any interface to an ETW trace point.

1

Created by: Microsoft
Id: EnterpriseLibrary.SemanticLogging.W
Version: 1.0.1304.1
Last Published: 5/24/2013
Downloads: 2065
View License Terms
Project Information
Report Abuse
Description:
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource, This add-on packages adds support to persisting log entries in Windows Azure table storage.
Tags: entlib entlib6 slab enterprise library ETW semantic logging EventSource event listener structured strongly-typed block sink Azure cloud WindowsAzure storage table
Dependencies:
EnterpriseLibrary.SemanticLogging (≥ 1.0)
WindowsAzure.Storage (≥ 2.0)
Each item above may have sub-dependencies subject to additional license agreements.

Settings | Close

4. Click **Accept** on the License Acceptance window that pops up.

To configure logging to Windows Azure table storage

1. Update the **ObservableEventListener** in the **Main** method of the **EntryPoint** class to also write messages to a Windows Azure table.

```
static void Main()
{
    Application.ThreadException +=
        new System.Threading.ThreadExceptionEventHandler(
          Application_ThreadException);

    //create formatter
    var formatter = new JsonEventTextFormatter(
        EventTextFormatting.Indented);

    //TODO: Set up and enable the event listener
    using (var listener = new ObservableEventListener())
    {
        listener.LogToConsole(formatter);
        listener.LogToWindowsAzureTable("Calculator",
        "UseDevelopmentStorage=true;");
        listener.EnableEvents(CalculatorEventSource.Log,
EventLevel.LogAlways, Keywords.All);

        Form entryForm = new MainForm();
        Application.EnableVisualStyles();
        Application.Run(entryForm);
    }
}
```

The listener will log with an **InstanceName** of “Calculator” to the storage location specified in the **ConnectionString.** The **ConnectionString** “UseDevelopmentStorage=true;” is used here because you are utilizing the Windows Azure Storage Emulator.

To run the application

1. Make sure the Windows Azure storage emulator is running.
2. Select **Debug | Start Without Debugging**.
3. Select your desired precision and click “Calculate.”

To view logs stored with Windows Azure

1. Select **View | Server Explorer** or press **Ctrl+Alt+S** to view the Server Explorer.
2. Expand **Windows Azure Storage | Development | Tables** or refresh if it is already expanded.
3. Here, there should be a table called **SLABLogsTable**. Either right-click and select **View Table** or double-click its name to view the table.

Note: Due to the nature of the storage emulator, you may have to refresh the Server Explorer view a few times (or wait a few minutes) before the table and the new logs appear. Clicking the **Execute** button will also accomplish this.

In this view, you are able to manipulate the data to quickly glean information from the logs. The table can easily be sorted on any column, and more sophisticated OData filters can be applied to the table to view specific logs of interest.

Because of the dynamic nature of Windows Azure tables, each entry will have its own schema, so each of the parameters used in the strongly typed methods in the **CalculatorEventSource** class will appear in a different named column. If you double-click on the entry, you will see that the data type is even preserved for each of the custom payload columns.

To verify that you have completed the exercise correctly, you can use the solution provided in the Lab02\end folder.

Lab 3: Testing an Event Source

Introduction

In this lab, you will ensure that your Event Source is valid by using the **EventSourceAnalyzer** class to write a unit test.

To begin this exercise, open the EnoughPI.sln file located in the Lab03\begin folder.

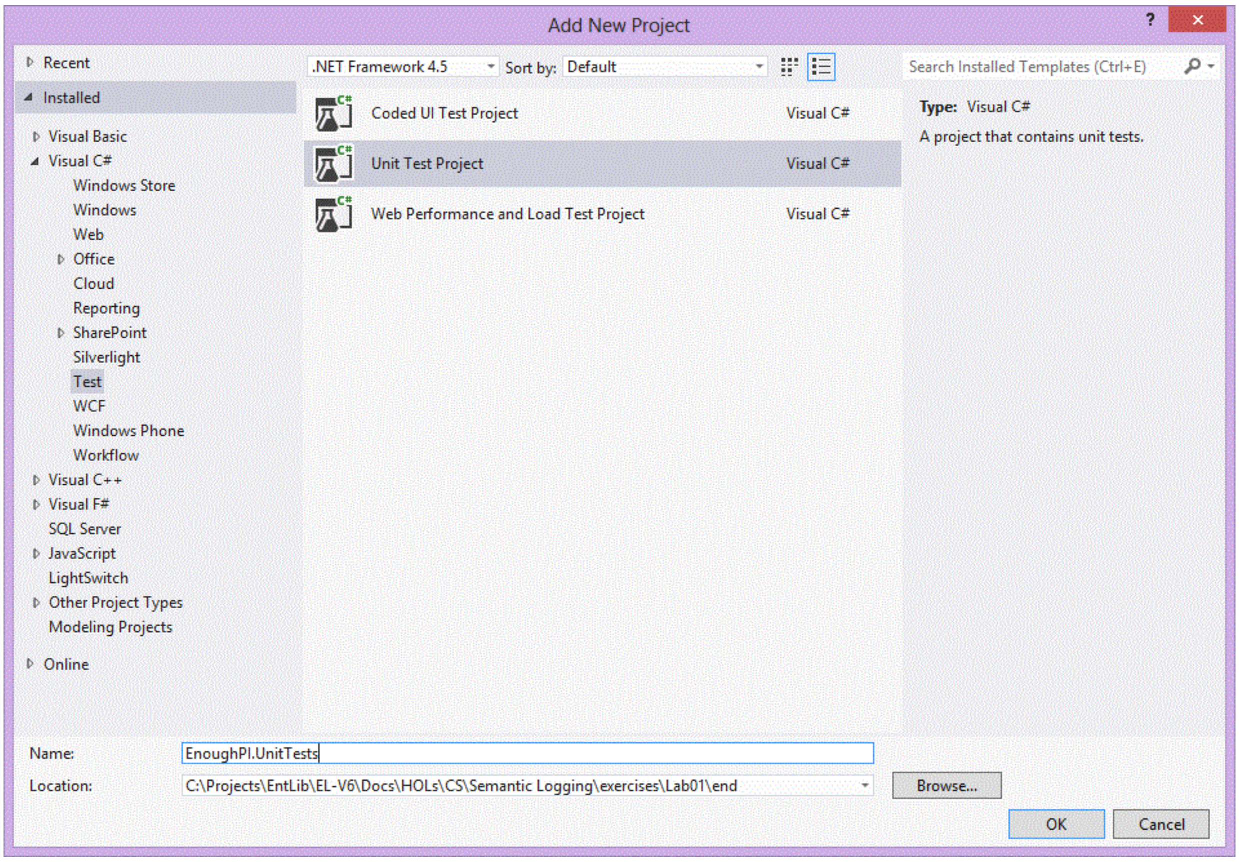
Task 1: Testing the Event Source

Using the Semantic Logging Application Block, you can check that your custom event source for common errors as part of your unit testing. The Semantic Logging Application Block includes a helper class for this purpose named **EventSourceAnalyzer**. Testing the Event Source is not necessary, but if your solution includes unit tests, the **EventSourceAnalyzer** provides a simple way to ensure your Event Source is valid.

Creating a Unit Test Project

To create a unit test project

1. Right-click the **EnoughPI** solution and select the **Add | New Project** menu option.
2. Under **Installed | Visual C#**, select **Test**.
3. Select **Unit Test Project**. Call this “EnoughPI.UnitTests”



Type: Visual C#

A project that contains unit tests.

Name: EnoughPI.UnitTests

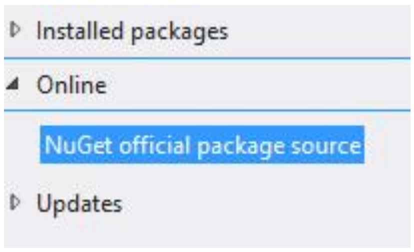
Location: C:\Projects\EntLib\EL-V6\Docs\HOLs\CS\Semantic Logging\exercises\Lab01\end

4. Click **OK.**

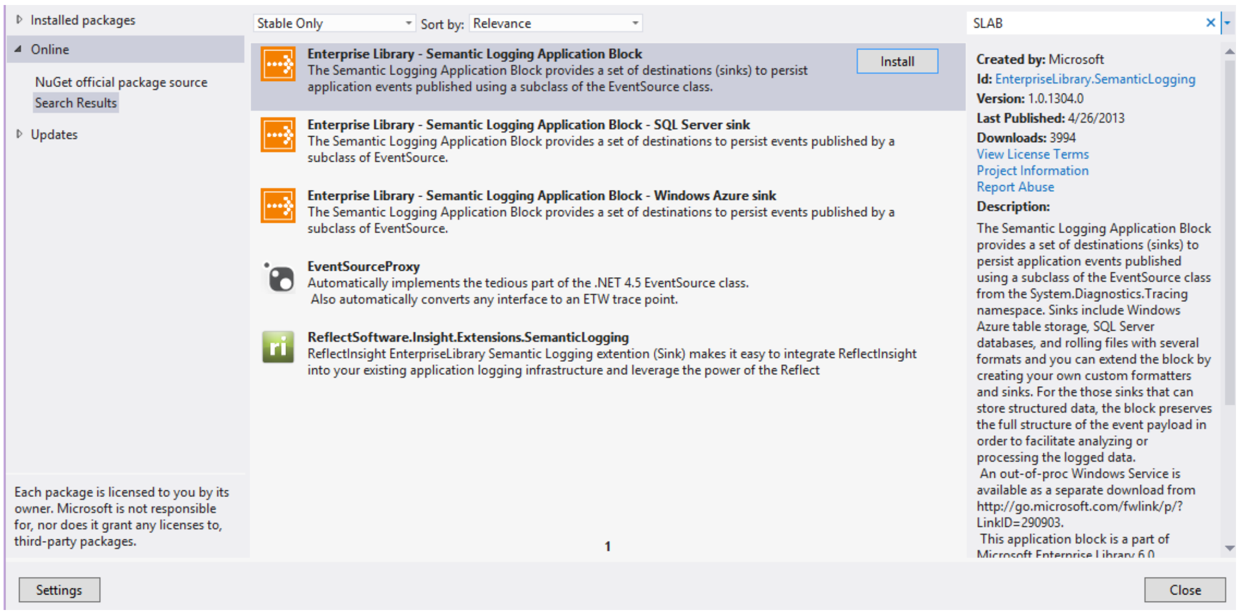
Adding References to the Required Assemblies

To add the necessary NuGet packages

1. Select the **EnoughPI.UnitTests** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the “Online” option to view NuGet packages available online.



3. Search for **SLAB** in the search bar. Select **Enterprise Library – Semantic Logging Application Block** and click install.



▷ Installed packages
◢ Online
NuGet official package source
Search Results
▷ Updates

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Stable Only | Sort by: Relevance

SLAB

Enterprise Library - Semantic Logging Application Block
The Semantic Logging Application Block provides a set of destinations (sinks) to persist application events published using a subclass of the EventSource class.
Install

Enterprise Library - Semantic Logging Application Block - SQL Server sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.

Enterprise Library - Semantic Logging Application Block - Windows Azure sink
The Semantic Logging Application Block provides a set of destinations to persist events published by a subclass of EventSource.

EventSourceProxy
Automatically implements the tedious part of the .NET 4.5 EventSource class.
Also automatically converts any interface to an ETW trace point.

ReflectSoftware.Insight.Extensions.SemanticLogging
ReflectInsight EnterpriseLibrary Semantic Logging extention (Sink) makes it easy to integrate ReflectInsight into your existing application logging infrastructure and leverage the power of the Reflect

1

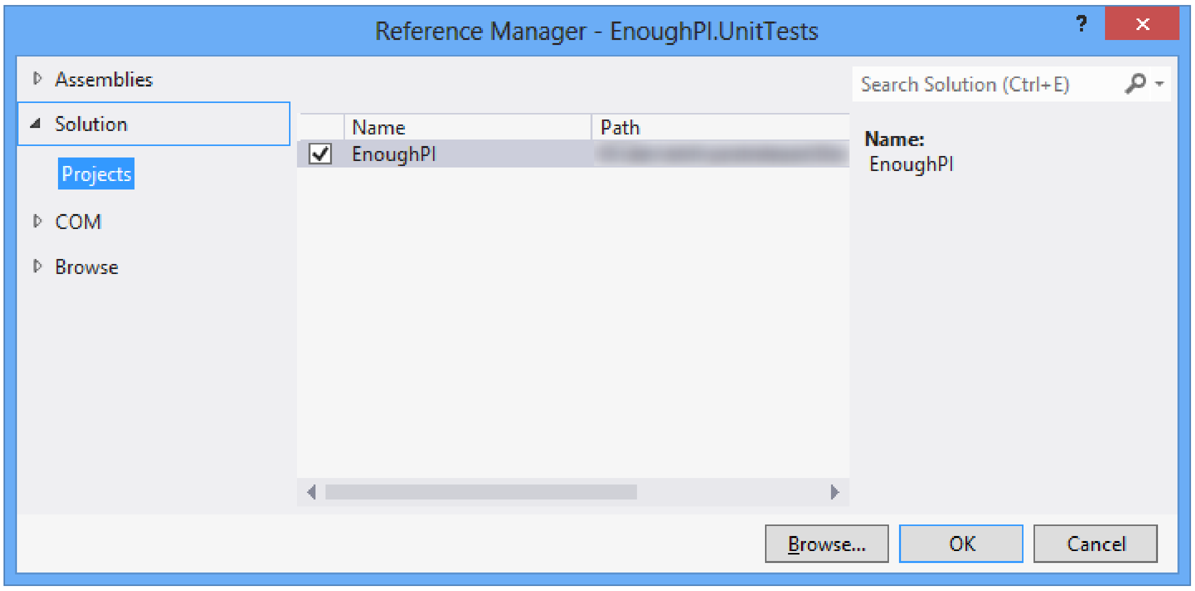
Created by: Microsoft
Id: EnterpriseLibrary.SemanticLogging
Version: 1.0.1304.0
Last Published: 4/26/2013
Downloads: 3994
View License Terms
Project Information
Report Abuse
Description:
The Semantic Logging Application Block provides a set of destinations (sinks) to persist application events published using a subclass of the EventSource class from the System.Diagnostics.Tracing namespace. Sinks include Windows Azure table storage, SQL Server databases, and rolling files with several formats and you can extend the block by creating your own custom formatters and sinks. For the those sinks that can store structured data, the block preserves the full structure of the event payload in order to facilitate analyzing or processing the logged data.
An out-of-proc Windows Service is available as a separate download from http://go.microsoft.com/fwlink/p/?LinkID=290903.
This application block is a part of

Settings | Close

4. Click **Accept** on the License Acceptance window that pops up.

To add a reference to the EnoughPI project

1. Select the **EnoughPI.UnitTests** project. Select the **Project | Add Reference** menu command.
2. Expand the **Solution** node and select the **Projects** entry.



Reference Manager - EnoughPI.UnitTests

▷ Assemblies
◢ Solution
Projects
▷ COM
▷ Browse

	Name	Path
✓	EnoughPI	[redacted]

Search Solution (Ctrl+E)

Name:
EnoughPI

Browse... | OK | Cancel

3. Check the **EnoughPI** project and click OK.

Adding a Unit Test for the Event Source

When you create an **EventSource** class for your application you must follow a number of conventions if the event methods are to work as expected. To help you verify that your **EventSource** class follows these conventions you can use the **EventSourceAnalyzer** class that is included in the Semantic Logging Application Block. You can use the **EventSourceAnalyzer** class in a unit test or invoke the **Inspect** method directly in application created to test your **EventSource** class. The **EventSourceAnalyzer** checks your **EventSource** class by using the following three techniques:

- It checks for errors when it enables an event listener using an **EventSource** instance.
- It verifies that it can request event schemas from the **EventSource** without errors.
- It verifies that it can invoke each event method in the **EventSource** class without errors.

To add a unit test for an event source

1. Make sure the **Semantic Logging Application Block** NuGet package and a reference to your **EnoughPI** project are added to your new unit test project.
2. Open **UnitTest1.cs**.
3. Add the following using statement:

```
using
Microsoft.Practices.EnterpriseLibrary.SemanticLogging.Utility;
using EnoughPI;
```

4. Rename **TestMethod1** to describe its function. Call it "VerifyCalculatorEventSourceIsCorrect"
5. Update the **VerifyCalculatorEventSourceIsCorrect** method to use the **EventSourceAnalyzer** to test the **CalculatorEventSource**.

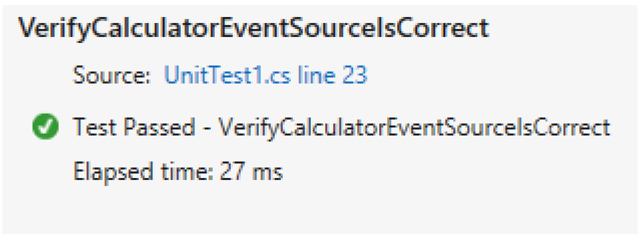
```
        [TestMethod]
        public void VerifyCalculatorEventSourceIsCorrect()
        {

EventSourceAnalyzer.InspectAll(CalculatorEventSource.Log);
        }
```

Running the Unit Test for the First Time

To run the unit test

1. **Build** your solution.
2. Select **Windows | Test Explorer** from the **Test** menu.
3. In the Test Explorer pane, click **Run All**.
4. This test will succeed because the event source built so far is valid since it satisfies all of the **EventSource** conventions. These include (but are not limited to):

- **EventSource** class has at least one event method.
- All event methods invoke the **WriteEvent** method.
- Parameters of event method have same type and order in the call to the **WriteEvent** method.
- All event methods are decorated with the **Event** attribute


VerifyCalculatorEventSourceIsCorrect

Source: UnitTest1.cs line 23

Test Passed - VerifyCalculatorEventSourceIsCorrect

Elapsed time: 27 ms

Adding an Invalid Event

A valid event requires that the event id and the event parameters passed to the **WriteEvent** method match the event id in the **Event** attribute and the all the parameters in the event method, in the same order in which they appear in the method's signature.

To add the invalid event

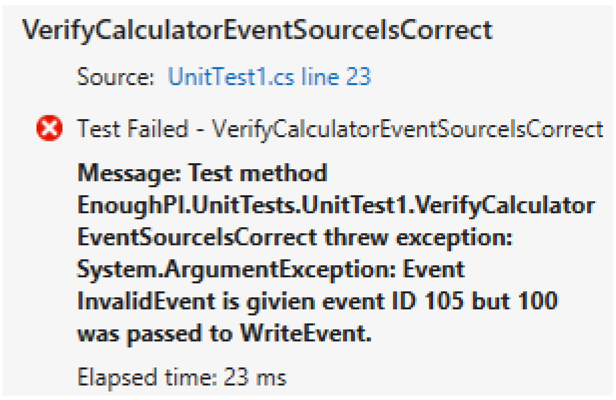
1. Add a new **InvalidEvent** method to the **CalculatorEventSouce** class configured so that the event id in the **Event** attribute and the parameter passed to the **WriteEvent** method do not match.

```
[Event(105)]
internal void InvalidEvent()
{
    this.WriteEvent(100);
}
```

Running the Failing Unit Test

To run the unit test

1. **Build** your solution.
2. Select **Windows | Test Explorer** from the **Test** menu.
3. In the Test Explorer pane, click **Run All**.
4. This time the test will fail because the event source contains an improperly authored event method, and the failure message will indicate the first issue detected in the event source.



VerifyCalculatorEventSourceIsCorrect

Source: UnitTest1.cs line 23

Test Failed - VerifyCalculatorEventSourceIsCorrect

Message: Test method EnoughPI.UnitTests.UnitTest1.VerifyCalculatorEventSourceIsCorrect threw exception: System.ArgumentException: Event InvalidEvent is givien event ID 105 but 100 was passed to WriteEvent.

Elapsed time: 23 ms

Fixing the Invalid Event

Fixing the event requires changing the code so that the event id and the parameters match.

To fix the invalid event

1. Update the event id passed to the **WriteEvent** method to the value specified in the **Event** attribute.

```
[Event(105)]
internal void InvalidEvent()
{
    this.WriteEvent(105);
}
```

Running the Fixed Unit

To run the unit test

1. **Build** your solution.
2. Select **Windows | Test Explorer** from the **Test** menu.
3. In the Test Explorer pane, click **Run All**.
4. This time the test will succeed again.

To verify that you have completed the exercise correctly, you can use the solution provided in the Lab03\end folder.

More Information

For more information about the Semantic Logging Application Block, see the documentation in the Enterprise Library 6 Developer's Guide and the Enterprise Library 6 Reference documentation.

patterns & practices

proven practices for predictable results

Copyright