MICROSOFT ENTERPRISE LIBRARY
SEMANTIC LOGGING APPLICATION BLOCK
6.0.1302.0

Community Technology Preview Release

This version of the Developer's Guide Semantic Logging Application Block chapter is released as part of the CTP release from Microsoft patterns and practices.

This release is provided as preview only. The purpose is to inform the community and to solicit early feedback.

Your feedback is invited. Please send your comments via http://aka.ms/slabfeedback

Copyright

Chapter 6 – Logging What You Mean

Introduction

Why do you need another logging block when the Logging Application Block already has a comprehensive set of features? You'll find that you can do many of the same things with the Semantic Logging Application Block: you can write log messages to multiple destinations, you can control the format of your log messages, and you can filter what gets written to the log. The answer why you might want to use the Semantic Logging Application Block lies with how the application writes messages to the logs.

Using a traditional logging infrastructure, such as that provided by the Logging Application Block, you might record an error in the UI of your application as shown in the following code sample:

C#

```
public void LogUIError(LogWriter myLogWriter, Exception ex)
{
  LogEntry logEntry = new LogEntry();

  logEntry.EventId = 100;
  logEntry.Priority = 2;
  logEntry.Message = "UI Error: " + ex.Message;
  logEntry.Categories.Add("UI Events");

  myLogWriter.Write(logEntry);
}
```

Using the Semantic Logging Application Block, you would record the same error as shown in the following code sample:

C#

```
MyCompanyEventSource.Log.UIError(ex.Message);
```

Notice how 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 write the log message 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.
- 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.

- 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.
- It is easier to correlate log entries from multiple sources.

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.

While it is possible to use the **EventSource** class in the .NET framework to use ETW to write log messages from your application, using the Semantic Logging Application Block makes it easier to incorporate this functionality, and makes it easier to manage the logging behavior of your application. The Semantic Logging Application Block makes it possible to use ETW to write log messages to multiple destinations, such as flat files or the Windows Event Log, and to control what your application logs by setting filters and logging verbosity.

The Semantic Logging Application Block is intended to help you move from the traditional, unstructured logging approach (such as that offered by the Logging Application Block) towards the semantic logging approach offered by ETW. The Semantic Logging Application Block enables you to use the **EventSource** class and semantic log messages in your applications without moving away from the log formats you are familiar with (such as flat files and database tables). In the future, you can easily migrate to a complete ETW-based solution without modifying your application code: you continue to use the same custom Event Source class, but use ETW tooling to capture and process your log messages instead of using the Semantic Logging Application Block event listeners.

#!SPOKENBY(Jana)!#
You can think of the Semantic Logging Application Block as a stepping-stone from a traditional logging approach (such as that taken by the Logging Application Block), to a modern, semantic approach as provided by ETW.

You can also use the Semantic Logging Application Block to create an out-of-process logger. This helps to reduce the logging overhead in your LOB applications because most of the processing of log messages takes place in a separate application. This is especially useful if you have a requirement to collect high volumes of trace messages from a production system.

#!SPOKENBY(Poe)!#

Collecting trace messages from your production system in a separate, out-of-process, application helps to reduce the overhead of tracing on your production application.

Many of the features of the Semantic Logging Application Block are similar to those of the Logging Application Block, and this chapter will refer to Chapter 5, “As Easy As Falling Off a Log,” where appropriate.

What Does the Semantic Logging Application Block Do?

The Semantic Logging Application Block enables you to use the ETW infrastructure to write log messages from your application. You define an ETW provider in your application and use the provider to write log messages from your application. The Semantic Logging Application Block receives notifications whenever the application writes a message to ETW. The Semantic Logging Application Block then writes the message to a destination of your choice based on your configuration of block. The Semantic Logging Application Block includes event listeners that can send log messages to a flat file, the Windows Event Log, a console window, a database, or Windows Azure storage.

Figure 1 illustrates how your application uses the Semantic Logging Application Block in process. It uses a custom event source to enable you to write log messages using the ETW infrastructure. ETW then notifies the event listeners in your application when there is a log message. The event listeners write the log message to a destination such as file, a database table, or a Windows Azure storage table. All of this takes place in process using managed code.

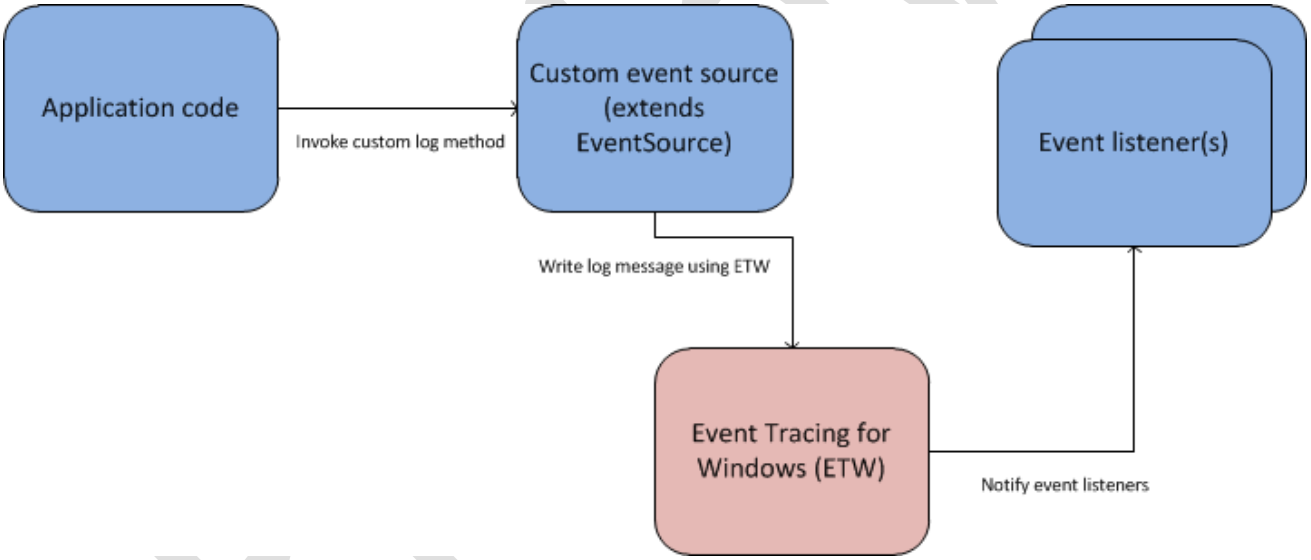

Figure 1

The Semantic Logging Application Block and Event Tracing for Windows

Figure 2 illustrates how you can use the Semantic Logging Application Block out-of-process. This is more complex to configure, but has the advantage that much of the logging overhead is removed from your LOB application. This approach uses managed code in the process that sends the log messages and in the process that collects and processes the log messages; some unmanaged code from the operating system that is responsible for delivering the log messages.

> In the out-of-process scenario, both the LOB application that generates log messages and the logging application that collects the messages must run on the same machine.

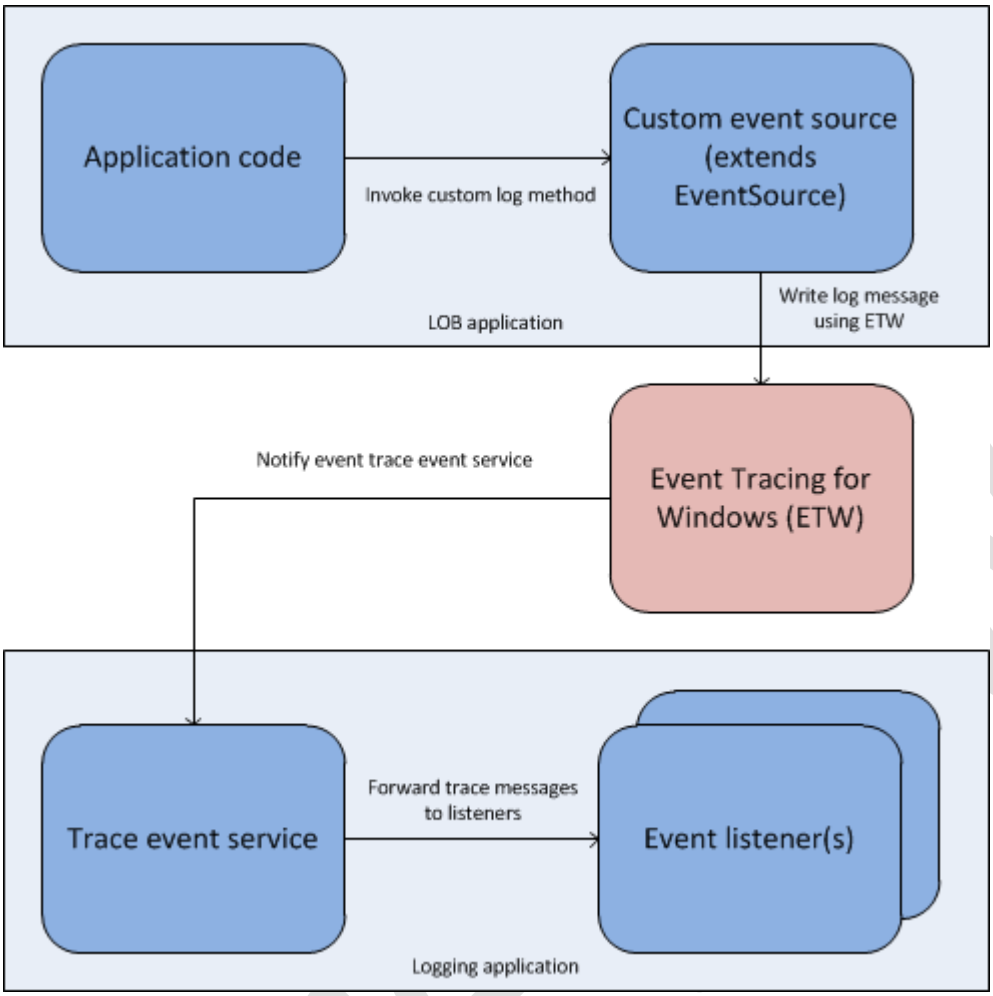

Figure 2

Using Semantic Logging Application Block out-of-process

#!SPOKENBY(Jana)!#

The Semantic Logging Application Block can work in process or out-of-process: you can host the event listeners your LOB application process or in a separate logging process. Either way you can use the same event listeners and formatters.

In-process or Out-of-Process?

When should you use the Semantic Logging Application Block in-process and when should you use it out-of-process? Although it is easier to configure and use the block in-process, there are a number of advantages to using it in the out-of-process scenario.

Using the Semantic Logging Application Block out-of-process is particularly useful in high throughput scenarios, such as collecting trace information from a production application. This is because most of the work related to collecting and processing the log messages from your application happens elsewhere:

partly in the ETW infrastructure and partly in the separate process. This minimizes the overhead associated with logging on your line-of-business application.

Using the block out-of-process also minimizes the risk of losing log messages if your line-of-business application crashes. When the line-of-business application creates a log message it immediately delivers it to the ETW infrastructure in the operating system, so that any log messages written by the line-of-business application will not get lost if that application crashes.

> **#!SPOKENBY(Poe)!#**
>
> You could still lose log messages in the out-of-process scenario if the server itself fails between the time the application writes the log message and the time the out-of-process host saves the message, or if the out-of-process host application crashes before it saves the message. However, out-of-process host is a robust and relatively simple application and is unlikely to crash.

Buffering Log Messages

Some event listeners, such as the **AzureTableEventListener** and **SqlDatabaseEventListener** classes can buffer log messages for a configurable time (ten seconds by default). These event listeners typically communicate over the network with the service that is ultimately responsible for persisting the log messages from your application. The block uses buffering in these event listeners to improve performance: typically, chunky rather chatty communication over a network improves the overall throughput.

However, using buffering introduces a trade-off: if the process that is buffering the messages crashes before delivering those log messages, then you lose those messages. The shorter the buffering period, the fewer messages you will lose in the event of an application crash, but at the cost of a slower throughput of log messages.

One of the reasons for using the Semantic Logging Application Block out-of-process is to mitigate this risk. If the message buffer is in a separate process from your line-of-business application, then the messages are not lost in the event that the application crashes. The out-of-process host applications for Semantic Logging Application Block event listeners are designed to be robust in order to minimize the chances of these applications crashing.

How Do I Use the Semantic Logging Application Block?

It's time to see some examples of the Semantic Logging Application Block use, including how to create an event source, how to configure the block, and how to write log entries.

Adding the Semantic Logging Application Block to Your Project

Before you write any code that uses the Semantic Logging Application Block, you must install the required assemblies to your project. You can install the block by using the NuGet package manager in Visual Studio: in the **Manage Nuget Packages** dialog, search online for the **EnterpriseLibrary.SemanticLogging** package and install it.

Creating an Event Source

Before you can write log messages using ETW, you must define what log messages you will write. This is what semantic logging means. You can specify the log messages you will write by extending the **EventSource** class in the **System.Diagnostics.Tracing** namespace in the .NET 4.5 framework. In the terms used by ETW, an **EventSource** implementation represents an "Event Provider."

The following code sample shows an example **EventSource** implementation. This example also includes the nested classes **Keywords** and **Tasks** that enable you to define additional information for your log messages.

C#

```
public enum MyColor { Red, Yellow, Blue };

[EventSource(Name = "Adatum")]
public class AdatumEventSource : EventSource
{
  public class Keywords
  {
    public const EventKeywords Page = (EventKeywords)1;
    public const EventKeywords DataBase = (EventKeywords)2;
    public const EventKeywords Diagnostic = (EventKeywords)4;
    public const EventKeywords Perf = (EventKeywords)8;
  }

  public class Tasks
  {
    public const EventTask Page = (EventTask)1;
    public const EventTask DBQuery = (EventTask)2;
  }

  [Event(1, Message = "Application Falure: {0}",
         Level = EventLevel.Error,
         Keywords = Keywords.Diagnostic)]
  internal void Failure(string message)
  {
    if (IsEnabled()) WriteEvent(1, message);
  }

  [Event(2, Message = "Starting up.",
         Keywords = Keywords.Perf,
         Level = EventLevel.Informational)]
  internal void Startup()
  {
    if (IsEnabled()) WriteEvent(2);
  }

  [Event(3, Message = "loading page {1} activityID={0}",
         Opcode = EventOpcode.Start,
```

```
        Task = Tasks.Page, Keywords = Keywords.Page,
        Level = EventLevel.Informational)]
  internal void PageStart(int ID, string url)
  {
    if (IsEnabled()) WriteEvent(3, ID, url);
  }

  [Event(4, Opcode = EventOpcode.Stop,
        Task = Tasks.Page, Keywords = Keywords.Page,
        Level = EventLevel.Informational)]
  internal void PageStop(int ID)
  {
    if (IsEnabled()) WriteEvent(4, ID);
  }

  [Event(5, Opcode = EventOpcode.Start,
        Task = Tasks.DBQuery, Keywords = Keywords.DataBase,
        Level = EventLevel.Informational)]
  internal void DBQueryStart(string sqlQuery)
  {
    if (IsEnabled()) WriteEvent(5, sqlQuery);
  }

  [Event(6, Opcode = EventOpcode.Stop, Task = Tasks.DBQuery,
        Keywords = Keywords.DataBase,
        Level = EventLevel.Informational)]
  internal void DBQueryStop()
  {
    if (IsEnabled()) WriteEvent(6);
  }

  [Event(7, Level = EventLevel.Verbose,
        Keywords = Keywords.DataBase)]
  internal void Mark(int ID)
  {
    if (IsEnabled()) WriteEvent(7, ID);
  }

  [Event(8)]
  internal void LogColor(MyColor color)
  {
    if (IsEnabled()) WriteEvent(8, (int)color);
  }

  public static readonly AdatumEventSource Log =
    new AdatumEventSource();
}
```

The class, **AdatumEventSource** that extends the **EventSource** class, contains definitions for all of the events that you want to be able to log using ETW. You can use the **EventSource** attribute to provide a more user friendly name for this event source that ETW will use when you save log messages.

#!SPOKENBY(Jana)!#

This **EventSource** follows the recommended naming convention. The name should start with your company name, and if you expect to have more than one **event source** for your company the name should include categories separated by '-'. Microsoft follows this convention; for example, there is an event source called “Microsoft-Windows-DotNetRuntime.” You should carefully consider this name because if you change it, it will break any users of your event source.

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. Notice that the **Event** attribute includes the same value as its first parameter.

#!SPOKENBY(Markus)!#
The custom event source file can get quite large; you should consider using partial classes to make it more manageable.

In the **AdatumEventSource** class, the log methods include a call to the **IsEnabled** method in the parent class to determine whether to write a log message.

#!SPOKENBY(Carlos)!#
It is good practice to include the call to the **IsEnabled** method before calling the **WriteEvent** method: it avoids a potentially costly logging computation if the event source is disabled. There is an override of the **IsEnabled** method that can check whether the event source is enabled for particular keywords and levels.

For more information about the **EventSource** class and the **WriteEvent** and **IsEnabled** methods, see EventSource Class on MSDN.

The **AdatumEventSource** class also includes a static method called **Log** that provides access to an instance of the **AdatumEventSource** class.

You can use the **Event** attribute to further refine the behavior of specific log methods.

Specifying the Log Message

You can write text directly to the log message by passing a string as a parameter to the **WriteMessage** method as shown in the **DBQueryStart** method that writes a copy of the SQL query to the log. However, you can also use the **Event** attribute’s **Message** parameter to control the string that is written to the log.

The **Startup** method always writes the string “Starting up” to the log when you invoke it. The **PageStart** method writes a message to the log, substituting the values of the **ID** and **url** parameters for the two placeholders in the string "loading page {1} activityID={0}."

> Notice that you must also pass these parameters on to the call to the **WriteEvent** method.

Using Keywords

The example includes a **Keywords** parameter for the **Event** attribute for many of the log methods. You can use keywords to define different groups of log methods so that when you enable an event source, you can specify which groups of log methods to enable: only log methods whose **Keywords** parameter matches one of the specified groups will be able to write log messages.

You must define the keywords you will use in a nested class called **Keywords** as shown in the example. Each keyword value is a 64 bit integer, which is treated as a bit array enabling you to define 64 different keywords. You can associate a log method with multiple keywords as shown in the following example where the **Failure** message is associated with both the **Diagnostic** and **Perf** keywords.

C#

```
[Event(1, Message = "Application Falure: {0}", Level = EventLevel.Error,
       Keywords = Keywords.Diagnostic | Keywords.Perf)]
public void Failure(string message) { WriteEvent(1, message); }
```

The following list offers some recommendations for using keywords in your organization.

- Events that you expect to fire less than 100 times per second do not need special treatment. You should use a default keyword for these events.
- Events that you expect to fire more than 1000 times per second should have a keyword. This will enable you to turn them off if you don’t need them.
- It’s up to you to decide whether events that typically fire at a frequency between these two values should have a keyword.
- Users will typically wan to switch on a specific keyword when they enable an event source, or enable all keywords.

> **#!SPOKENBY(Poe)!#**
>
> Keep it simple for users to enable just the events they need.

Specifying the Log Level

You can use the **Level** parameter of the **Event** attribute to specify when the message should be logged. The **EventLevel** enumeration determines the available log levels: **Verbose (5)**, **Informational (4)**, **Warning (3)**, **Error (2)**, **Critical (1)**, and **LogAlways (0)**. **Informational** is the default logging level. When you enable an event source in your application, you can specify a log level, and the event source will log all log messages with same or lower log level. For example, if you enable an event source with the

warning log level, all log methods with a level parameter value of **Warning**, **Error**, **Critical**, and **LogAlways** will be able to write log messages.

> **#!SPOKENBY(Poe)!#**
>
> You should use log levels less than **Informational** for relatively rare warnings or errors. When in doubt stick with the default of **Informational** level, and use the **Verbose** level for events that can happen more than 1,000 times per second. Typically, users use keywords to control what is logged rather than the log level, so don't worry about it too much.

Using Opcodes and Tasks

You can use the **Opcodes** and **Tasks** parameters of the **Event** attribute to add additional information to the message that the event source logs. The **Opcodes** and **Tasks** are defined using nested classes of the same name in the same way that you define **Keywords**. The example event source includes two tasks: **Page** and **DBQuery**.

> **#!SPOKENBY(Poe)!#**
> The logs contain just the numeric task and opcode identifiers. The developers who write the **EventSource** class and IT Pros who use the logs must agree on the definitions of the tasks and opcodes used in the application. Notice how in the sample, the task constants have meaningful names such as **Page** and **DBQuery**: these tasks appear in the logs as task ids **1** and **2** respectively.

> If you choose to define custom opcodes, you should assign integer values of 11 or above. If you define a custom opcode with a value of 10 or below, messages that use these opcodes will not be delivered.

Configuring the Semantic Logging Application Block

How you configure the Semantic Logging Application Block depends on whether you are using it in-process or out-of-process. If you are using the block in-process, then you provide the configuration information for your event-listeners in code; if you are using the block out-of-process, then you provide the configuration information in an XML file. The section “How do I Use the Semantic Logging Application Block to Log Trace Messages Out-of-Process?” later in this chapter describes the configuration in the out-of-process scenario. This section describes the in-process scenario.

Typically, the built-in event listeners have multiple constructors that enable you to configure the event listener as you create it. Then, when you enable the listener, you can specify the event source to use, the highest level of event to capture, and any keywords to filter on. The following code sample shows an example that creates, configures, and enables two event listeners.

C#

```
var dbListener = new DatabaseEventListener(
  "SemanticLogging", "dbo.WriteTrace");
var eventLogListener = new EventLogEventListener("SemanticLogging");
```

```
dbListener.EnableEvents(MyCompanyEventSource.Log,
  EventLevel.Informational, Keywords.All);
eventLogListener.EnableEvents(AdatumEventSource.Log,
  EventLevel.Warning, Keywords.All);
```

The console and file based event listeners can also use a formatter to control the format of the output. These formatters may also have configuration options. The following code sample shows an example that configures a JSON formatter, a console listener, and uses a custom color mapper.

C#

```
var formatter = new JsonEventTextFormatter(EventTextFormatting.Indented);
var colorMapper = new AdatumColorMapper();
listener = new ConsoleEventListener(formatter, colorMapper);
listener.EnableEvents(AdatumEventSource.Logger, EventLevel.LogAlways);
```

A color mapper is a simple class that specifies the color to use for different event levels as shown in the following code sample.

C#

```
public class MyCustomColorMapper : IConsoleColorMapper
{
  public ConsoleColor? Map(
    System.Diagnostics.Tracing.EventLevel eventLevel)
  {
    switch (eventLevel)
    {
    case EventLevel.Critical:
      return ConsoleColor.White;
    case EventLevel.Error:
      return ConsoleColor.DarkMagenta;
    case EventLevel.Warning:
      return ConsoleColor.DarkYellow;
    case EventLevel.Verbose:
      return ConsoleColor.Blue;
    default:
      return null;
    }
  }
}
```

Writing to the Log

Before you can write a log message, you must have an event source class in your application that defines what log messages you can write. The section “Creating an Event Source” earlier in this chapter describes how you can define such an event source. You must also add the Semantic Logging Application Block to your application: the easiest way to do this is using NuGet.

The following code sample shows a simple example of how you can use the event source shown previously in this chapter to write messages to a console window.

C#

```
class Program
{
  static void Main(string[] args)
  {
    var consoleEventListener = new ConsoleEventListener();

    consoleEventListener.EnableEvents(AdatumEventSource.Log,
      EventLevel.LogAlways, Keywords.All);

    AdatumEventSource.Log.Startup();
    AdatumEventSource.Log.Failure("Failed to start");

    consoleEventListener.DisableEvents(AdatumEventSource.Log);
    consoleEventListener.Dispose();
  }
}
```

This sample shows how to create an instance of the **ConsoleEventListener** class from the Semantic Logging Application Block, enable the listener to process log messages from the **AdatumEventSource** class, write some log messages, disable the listener, and then dispose of the listener. Typically, you will create and enable an event listener when your application starts up and initializes, and disable and dispose of your event listener as the application shuts down.

> **#!SPOKENBY(Jana)!#**
>
> For those event listeners that buffer log messages, such as the Windows Azure table storage event listener, disposing of the listener flushes the queue.

When you enable an event listener, you can specify which level of log messages should be logged and which groups of log methods (identified using the keywords) should be active. The example shows how to active log methods at all levels and with all keywords.

> If you want to activate log messages in all groups, you must use the **Keywords.All** parameter value. If you do not supply a value for this optional parameter, only events with no keywords are active.

The following code sample shows how to activate log messages with a level of **Warning** or lower with keywords of either **Perf** or **Diagnostic**.

C#

```
consoleEventListener.EnableEvents(AdatumEventSource.Log,
      EventLevel.Warning, Keywords.Diagnostic | Keywords.Perf);
```

How do I Use the Semantic Logging Application Block to Log Trace Messages Out-of-Process?

To use the Semantic Logging Application Block to process log messages out-of-process you must add the necessary code to your LOB application to create trace messages when interesting events occur in the application. You must also create a separate application that is responsible for collecting and processing the messages.

#!SPOKENBY(Jana)!#

You should consider collecting and processing trace messages in a separate process if you anticipate a high volume of messages. Collecting and processing the trace messages in a separate application offloads much of the trace overhead from your LOB application.

You should also consider collecting and processing trace messages in a separate process if you are using an event listener with a high latency such as the Windows Azure table storage event listener.

Running the Out-of-Process Host Application

The Semantic Logging Application Block includes the for you to use; you can run this application as a Windows Service or as a console application.

Typically, you should run the Out-of-Process Host as a console application when you are developing and testing your logging behavior. It's convenient to be able to stop and start the event listener host application and view any log messages in a console window.

In a production environment, you should run the Out-of-Process Host as a Windows Service. You easily can configure a Windows Service to start when the operating system starts. In a production environment, you are unlikely to want to see log messages as they are processed in a console window: more likely, you will want to save the log messages to a file, database, or some other persistent storage.

If you plan to use an out-of-process host in Windows Azure, you should run it as a Windows Service. You can install and start the service in a Windows Azure start up task.

#!SPOKENBY(Poe)!#

As an alternative to the Out-of-Process Host application included with the block, you could configure ETW to save the trace messages to a .etl file and use another tool to read this log file. However, the Out-of-Process Host application offers greater more choices of where to persist the log messages.

By default, the Out-of-Process Host application reads the configuration for the block from a file named SemanticLogging-svc.xml. It reads this file at start up, and then continues to monitor this file for changes. If it detects any changes, it dynamically reconfigures the block based in the changes it discovers.

Creating Trace Messages

You create trace messages in the LOB application in exactly the same way as described previously in this chapter. First, create a custom **EventSource** class that defines all of the trace messages your application uses as shown in the following code sample.

C#

```
[EventSource(Name = "Adatum")]
public class AdatumEventSource : EventSource
{
  ...
  [Event(1, Message = "Application Falure: {0}",
         Level = EventLevel.Error]
  public void Failure(string message)
  {
    WriteEvent(1, message);
  }
  ...

  public static readonly AdatumEventSource Log =
    new AdatumEventSource();
}
```

#!SPOKENBY(Poe)!#

If you use the same name for the event source in multiple applications, all of the log messages from those applications will be collected and processed by a single logging application. If you omit the **EventSource** attribute, ETW uses the class name as the name of the event source.

Second, create the trace messages in your application code as shown in the following code sample.

C#

```
AdatumEventSource.Log.Failure("Failed to start");
```

You don't need to create any event listeners in the LOB application if you are processing the log messages in a separate application.

Choosing Event Listeners

The Semantic Logging Application Block includes event listeners that enable you to save log messages to the following locations: a database, a Windows Azure table, the Windows Event Log, and to flat files. The choice of which event listeners to use depends on how you plan to analyze and monitor the information you are collecting. For example, if you save your log messages to a database you can use SQL to query and analyze the data. If you are collecting log messages in a Windows Azure application you should consider writing the messages to Windows Azure table storage. You can export data from Windows Azure table storage to a CSV file for further analysis on-premises.

The console event listener is useful during development and testing because you can easily see the trace messages as they appear, however the console event listener is not appropriate for use in a production environment.

If you use the **XmlEventTextFormatter** class to save the log messages formatted as XML in a flat file, you will need to perform some post-processing on the log files before you can use a tool such as Log Parser to read them. This is because the flat file does not have an XML root element, therefore the file does not contain well-formed XML. It's easy to add opening and closing tags for a root element to the start and end of the file.

Although the block does not include a CSV formatter, you can easily export log messages in CSV format from Windows Azure table storage or a SQL Server database.

The Log Parser tool can add a header to some file formats such as CSV, but it cannot add a header or a footer to an XML file.

Collecting and Processing the Log Messages

To collect and process the log messages from the LOB application you run the Enterprise Library Semantic Logging Out-of-Process Windows Service/Console Host (Out-of-Process Host) application (SemanticLogging-svc.exe) included with the Semantic Logging Application Block. You can run this application as a console application or install it as a Windows service. This application uses configuration information from an XML file to determine which event sources to collect trace messages from and which event listeners to use to process those messages.

#!SPOKENBY(Jana)!#

If you want to collect and process messages out-of-process in Windows Azure, you should run the Out-of-Process host application as a Windows Service. It's not possible to use it in console mode in a Windows Azure start up task. You install and start the service using the **–install** switch.

The default name for the configuration file is SemanticLogging-svc.xml. When you edit this XML files in Visual Studio, you can use the supplied schema file (SemanticLogging-svc.xsd) to provide Intellisense support in the editor.

The following snippet shows an example configuration file that defines how to collect trace messages written by another application using the Adatum event source. This example uses three event listeners to write save messages to three different destinations. Each event listener is configured to log messages with different severity levels.

XML

```
<?xml version="1.0"?>
<configuration
  xmlns="http://schemas.microsoft.com/practices/2013/entlib/semanticlogging/etw"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.microsoft.com/practices/2013/entlib/
semanticlogging/etw SemanticLogging-svc.xsd">
```

```
  <!-- Optional settings for this host -->
  <traceEventService/>

  <!-- EventSource reference definitons used by this host to
  listen ETW events emited by these EventSource instances -->
  <eventSources>
    <eventSource name="Adatum">
      <eventListeners>
        <eventListener name="DatabaseEventListener"
                       level="Warning"/>
        <eventListener name="RollingFlatFileEventListener"
                       level="Error"/>
        <eventListener name="ConsoleEventListener"
                       level="LogAlways"/>
      </eventListeners>
    </eventSource>
  </eventSources>

  <!-- Event Listener definitions used by event sources -->
  <eventListeners>
    <consoleEventListener name="ConsoleEventListener"
      formatterName="text"/>
    <rollingFlatFileEventListener
      name="RollingFlatFileEventListener"
      fileName="RollingFlatFile.log" timeStampPattern="yyyy"
      rollFileExistsBehavior="Overwrite" rollInterval="Day"/>
    <databaseEventListener name="DatabaseEventListener"
      instanceName="Demo"
      connectionString="Data Source=(localdb)\v11.0;
      AttachDBFilename='|DataDirectory|\Logging.mdf';
      Initial Catalog=SemanticLoggingTests;
      Integrated Security=True"/>
  </eventListeners>

  <!-- Formatters definitions used by Event Text Listeners -->
  <formatters>
    <eventTextFormatter name="text"
      header="+=======================================+"/>
  </formatters>

</configuration>
```

If you are collecting high volumes of trace messages from a production application, you may need to tweak the settings for the trace event service in the XML configuration file. For information about these settings see the topic Configuring the Block for High Throughput Scenarios on MSDN.

A single instance of the console application or Windows service can collect messages from multiple event sources in multiple LOB applications. You can also run multiple instances, but each instance must use a unique session name.

#!SPOKENBY(Poe)!#

By default, each instance of the console application has a unique session name. However, if you decide to provide a session name in the XML configuration file as part of the trace event service settings, you must ensure that the name you choose is unique.

Customizing the Semantic Logging Application Block

The Semantic Logging Application Block provides a number of extension points that enable you to further customize its behavior. These extension points include:

- Creating custom formatters for use both in-process and out-of-process.
- Creating custom event listeners for use both in-process and out-of-process.
- Creating a custom application to collect trace messages from your LOB applications.

Creating Custom Formatters

The Semantic Logging Application Block includes three text formatters to format the log messages written to the console or to text files: **EventTextFormatter**, **JsonTextFormatter**, and **XmlTextFormatter**.

There are three different scenarios to consider if you are planning to create a new custom event text formatter for the Semantic Logging Application Block. They are listed in order of increasing level of complexity to implement:

- Creating a custom formatter for use in-process.
- Creating a custom formatter for use out-of-process without Intellisense support when you edit the configuration file.
- Creating a custom formatter for use out-of-process with Intellisense support when you edit the configuration file.

Creating a Custom In-Process Event Text Formatter

Using text formatters when you are using the Semantic Logging Application Block in-process is a simple case of instantiating and configuring in code the formatter you want to use, and then passing it to the constructor of the event listener you are using. The following code sample shows how to create and configure an instance of the **JsonEventTextFormatter** formatter class and pass it a listener:

C#

```
var formatter = new JsonEventTextFormatter(EventTextFormatting.Indented);
var listener = new ConsoleEventListener(formatter);
listener.EnableEvents(MyCustomEventSource.Logger, EventLevel.LogAlways);
```

To create a custom text formatter, you create a class that implements the **IEventTextFormatter** interface and then use that class as your custom formatter.

The following code sample shows part of the built-in **JsonEventTextFormatter** formatter class as an example implementation.

C#

```
public class JsonEventTextFormatter : IEventTextFormatter
{
  private Newtonsoft.Json.Formatting formatting;
  private const string EntrySeparator = ",";

  ...

  public JsonEventTextFormatter(EventTextFormatting formatting)
  {
    this.formatting = (Newtonsoft.Json.Formatting)formatting;
  }

  public EventTextFormatting Formatting
  {
    get { return (EventTextFormatting)formatting; }
  }

  // Implement the IEventTextFormatter interface
  public void WriteEvent(EventEntry eventData, TextWriter writer)
  {
    using (var jsonWriter = new JsonTextWriter(writer)
      { CloseOutput = false, Formatting = this.formatting })
    {
      jsonWriter.WriteStartObject();
      jsonWriter.WritePropertyName("SourceId");
      jsonWriter.WriteValue(eventData.EventSourceId);
      jsonWriter.WritePropertyName("EventId");
      jsonWriter.WriteValue(eventData.EventId);

      ...

      jsonWriter.WriteEndObject();

      jsonWriter.WriteRaw(EntrySeparator);

      if (jsonWriter.Formatting == Newtonsoft.Json.Formatting.Indented)
      jsonWriter.WriteRaw("\r\n");
    }
  }
}
```

Creating a Custom Out-of-Process Event Text Formatter without Intellisense Support

Creating an event text formatter to use when you are using the Semantic Logging Application Block out-of-process requires some additional work to support configuring the formatter from the XML file that contains the configuration information for the event sources, event listeners, and formatters.

The XML schema file for this configuration file already contains a definition for an element named **customEventTextFormatter** that you can use with a custom formatter class. The following XML sample shows how to use this built-in element.

XML

```
<?xml version="1.0"?>
<configuration
  xmlns="http://schemas.microsoft.com/practices/2013/entlib/semanticlogging/etw">

  <traceEventService sessionName="Adatum Outproc Listener" />

  <eventSources>
    <eventSource name="Adatum">
      <eventListeners>
        <eventListener name="ConsoleEventListener" level="LogAlways"/>
      </eventListeners>
    </eventSource>
  </eventSources>

  <eventListeners>
    <consoleEventListener name="ConsoleEventListener"
      formatterName="customtext"/>
  </eventListeners>

  <formatters>
    <customEventTextFormatter name="customtext"
      type="CustomTextFormatter.PrefixEventTextFormatter, CustomTextFormatter">
      <parameters>
        <parameter name="header" type="System.String"
          value="========================================================="/>
        <parameter name="footer" type="System.String"
          value="========================================================="/>
        <parameter name="prefix" type="System.String" value="> "/>
        <parameter name="dateTimeFormat" type="System.String" value="O"/>
      </parameters>
    </customEventTextFormatter>
  </formatters>

</configuration>
```

This example illustrates how you must specify the type of the custom formatter, and use **parameter** elements to define the configuration settings. The out-of-process host application looks in the folder it is run from a DLL that contains the custom formatter types.

You can use a custom in-process formatter class out-of-process. The following code sample shows the **PrefixEventTextFormatter** class referenced by the XML configuration.

C#

```
public class PrefixEventTextFormatter : IEventTextFormatter
{
  public PrefixEventTextFormatter(string header, string footer,
    string prefix, string dateTimeFormat)
  {
    this.Header = header;
    this.Footer = footer;
    this.Prefix = prefix;
    this.DateTimeFormat = dateTimeFormat;
  }

  public string Header { get; set; }

  public string Footer { get; set; }

  public string Prefix { get; set; }

  public string DateTimeFormat { get; set; }

  public void WriteEvent(EventEntry eventEntry, TextWriter writer)
  {
    // Write header
    if (!string.IsNullOrWhiteSpace(this.Header))
    writer.WriteLine(this.Header);

    // Write properties
    writer.WriteLine("{0}SourceId : {1}",
      this.Prefix, eventEntry.EventSourceId);
    writer.WriteLine("{0}EventId : {1}",
      this.Prefix, eventEntry.EventId);
    writer.WriteLine("{0}Keywords : {1}",
      this.Prefix, eventEntry.Schema.Keywords);
    writer.WriteLine("{0}Level : {1}",
      this.Prefix, eventEntry.Schema.Level);
    writer.WriteLine("{0}Message : {1}",
      this.Prefix, eventEntry.FormattedMessage);
    writer.WriteLine("{0}Opcode : {1}",
      this.Prefix, eventEntry.Schema.Opcode);
    writer.WriteLine("{0}Task : {1} {2}",
      this.Prefix, eventEntry.Schema.Task,
      eventEntry.Schema.TaskName);
    writer.WriteLine("{0}Version : {1}",
      this.Prefix, eventEntry.Schema.Version);
    writer.WriteLine("{0}Payload :{1}",
      this.Prefix, FormatPayload(eventEntry));
```

```
    writer.WriteLine("{0}Timestamp : {1}",
      this.Prefix,
      eventEntry.GetFormattedTimestamp(this.DateTimeFormat));


    // Write footer
    if (!string.IsNullOrWhiteSpace(this.Footer))
    writer.WriteLine(this.Footer);

    writer.WriteLine();
  }

  private static string FormatPayload(EventEntry entry)
  {
    var eventSchema = entry.Schema;
    var sb = new StringBuilder();
    for (int i = 0; i < entry.Payload.Count; i++)
    {
      try
      {
        sb.AppendFormat(" [{0} : {1}]",
          eventSchema.Payload[i], entry.Payload[i]);
      }
      catch (Exception e)
      {
        SemanticLoggingEventSource.Log.EventEntryTextWriterFailed(
          e.ToString());
        sb.AppendFormat(" [{0} : {1}]", "Exception",
          string.Format(CultureInfo.CurrentCulture,
          "Cannot serialize to XML format the payload: {0}",
          e.Message));
      }
    }
    return sb.ToString();
  }
}
```

You must ensure that the order and type of the **parameter** elements in the configuration file matches the order and type of the constructor parameters in the custom formatter class.

Creating a Custom Out-of-Process Event Text Formatter with Intellisense Support

Instead of using the **customEventTextFormatter** and **parameter** elements in the configuration file, you can define your own custom element and attributes and enable Intellisense support in the Visual Studio XML editor. In this scenario, the XML configuration file looks like the following sample:

XML

```
<?xml version="1.0"?>
<configuration
  xmlns="http://schemas.microsoft.com/practices/2013/entlib/semanticlogging/etw">
```

```
  <traceEventService sessionName="Adatum Outproc Listener" />

  <eventSources>
    <eventSource name="Adatum">
      <eventListeners>
        <eventListener name="ConsoleEventListener" level="LogAlways"/>
      </eventListeners>
    </eventSource>
  </eventSources>

  <eventListeners>
    <consoleEventListener name="ConsoleEventListener"
      formatterName="customtext"/>
  </eventListeners>

  <formatters>
    <prefixEventTextFormatter xmlns="urn:demo.etw.customformatter"
      name="customtext"
      header="=========================================="
      footer="=========================================="
      prefix="# "
      dateTimeFormat="O" />
  </formatters>

</configuration>
```

#!SPOKENBY(Markus)!#

Notice how the **prefixEventTextFormatter** element has a custom XML namespace.

The custom XML namespace that enables Intellisense behavior for the contents of the **prefixEventTextFormatter** element is defined in the XML schema file shown in the following sample.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="PrefixEventTextFormatterElement"
    targetNamespace="urn:demo.etw.customformatter"
    xmlns="urn:demo.etw.customformatter"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xs:element name="prefixEventTextFormatter">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="header" type="xs:string" use="required" />
      <xs:attribute name="footer" type="xs:string" use="required" />
      <xs:attribute name="prefix" type="xs:string" use="required" />
      <xs:attribute name="dateTimeFormat" type="xs:string" use="required" />
```

```
    </xs:complexType>
  </xs:element>

</xs:schema>
```

You should place this schema file, along with the XML configuration file, in the folder where you are running the listener host application.

In addition to creating the XML schema, you must also create an element definition that enables the block to read the configuration from the XML file and create an instance of the formatter. The following code sample shows the element definition class for this example.

C#

```
[XmlRoot("prefixEventTextFormatter", Namespace = "urn:demo.etw.customformatter")]
public class PrefixEventTextFormatterElement : EventTextFormatterElement
{
  [XmlAttribute("header")]
  public string Header { get; set; }

  [XmlAttribute("footer")]
  public string Footer { get; set; }

  [XmlAttribute("prefix")]
  public string Prefix { get; set; }

  [XmlAttribute("dateTimeFormat")]
  public string DateTimeFormat { get; set; }

  public override IEventTextFormatter CreateEventTextFormatter()

    return new PrefixEventTextFormatter(
      this.Header, this.Footer, this.Prefix, this.DateTimeFormat);
  }
}
```

This class extends the **EventTextFormatterElement** class and uses the **XmlRoot** and **XmlAttribute** classes to map the contents of the XML configuration file to the properties of the class.

Creating Custom Event Listeners

The Semantic Logging Application Block includes a number of event listeners such as the **ConsoleEventListener**, the **EventLogEventListener**, and the **RollingFlatFileEventListener**.

There are three different scenarios to consider if you are planning to create a new custom event listener for the Semantic Logging Application Block. They are listed in order of increasing level of complexity to implement:

- Creating a custom event listener for use in-process.

- Creating a custom event listener for use out-of-process without Intellisense support when you edit the configuration file.
- Creating a custom event listener for use out-of-process with Intellisense support when you edit the configuration file.

Creating a Custom In-Process Event Listener

To create a custom event listener to use in-process, you must create a new event listener class that extends the abstract **EventListener** class in the **System.Diagnostics.Tracing** namespace. Typically, you override the **OnEventWritten** method from this class as shown in the following example from the built-in **ConsoleEventListener** class.

C#

```
public class ConsoleEventListener : EventListener
{
  private IEventTextFormatter formatter;
  private IConsoleColorMapper colorMapper;
  private readonly IEventTextFormatter DefaultFormatter =
    new EventTextFormatter();
  private readonly IConsoleColorMapper DefaultColorMapper =
    new DefaultConsoleColorMapper();
  private readonly EventSourceSchemaCache schemaCache =
    EventSourceSchemaCache.Instance;

  private static readonly object lockObject = new object();

  public ConsoleEventListener()
  : this(null, null)
  {
  }

  public ConsoleEventListener(IEventTextFormatter formatter)
  : this(formatter, null)
  {
  }

  public ConsoleEventListener(IConsoleColorMapper colorMapper)
  : this(null, colorMapper)
  {
  }

  public ConsoleEventListener(IEventTextFormatter formatter,
    IConsoleColorMapper colorMapper)
  {
    this.formatter = formatter ?? DefaultFormatter;
    this.colorMapper = colorMapper ?? DefaultColorMapper;
  }
```

```
public IEventTextFormatter Formatter
{
  get { return formatter; }
}

public IConsoleColorMapper ColorMapper
{
  get { return colorMapper; }
}

private void ConsoleWriter(EventLevel level, Action write)
{
  lock (lockObject)
  {
    ConsoleColor? currentColor = null;
    try
    {
      var newColor = this.colorMapper.Map(level);
      if (newColor.HasValue)
      {
        currentColor = Console.ForegroundColor;
        Console.ForegroundColor = newColor.Value;
      }
      write();
      Console.Out.Flush();
    }
    catch (Exception e)
    {
      SemanticLoggingEventSource.Log
        .ConsoleEventListenerWriteEventFailed(e.ToString());
    }
    finally
    {
      if (currentColor.HasValue)
      Console.ForegroundColor = currentColor.Value;
    }
  }
}

protected override void OnEventWritten(EventWrittenEventArgs eventData)
{
  var entry = EventEntry.Create(eventData,
    this.schemaCache.GetSchema(eventData.EventId, eventData.EventSource));

  ConsoleWriter(entry.Schema.Level, () =>
  {
    BufferedWriter.ExecuteWrite(Console.Out, w =>
      this.formatter.WriteEvent(entry, w));
```

```
    });
  }

}
```

To learn how to create a custom in-process event listener you should also examine the source code of some of the other built-in event listeners.

To use an event listener in-process, you can create and configure the event listener in code as shown in the following code sample.

C#

```
var formatter = new JsonEventTextFormatter(EventTextFormatting.Indented);
var listener = new ConsoleEventListener(formatter);
listener.EnableEvents(MyCustomEventSource.Logger, EventLevel.LogAlways);
```

Creating a Custom Out-of-Process Event Listener without Intellisense Support

When you use the Semantic Logging Application Block out-of-process, the block creates and configures event listener instances based on configuration information in an XML file. You can configure the block to use a custom event listener by using the **customEventListener** element as shown in the following XML sample.

XML

```
<?xml version="1.0"?>
<configuration xmlns=
  "http://schemas.microsoft.com/practices/2013/entlib/semanticlogging/etw">

  <traceEventService sessionName="Adatum Outproc Listener" />

  <eventSources>
    <eventSource name="Adatum">
      <eventListeners>
        <eventListener name="ConsoleEventListener" level="LogAlways"/>
        <eventListener name="email" level="Critical"/>
      </eventListeners>
    </eventSource>
  </eventSources>

  <eventListeners>
    <consoleEventListener name="ConsoleEventListener" formatterName="text"/>
    <customEventListener name="email"
       type="SimpleCustomEventListenerExtension.SimpleEmailEventListener,
       SimpleCustomEventListenerExtension">
      <parameters>
        <parameter name="host" type="System.String" value="smtp.live.com" />
        <parameter name="port" type="System.Int32" value="587" />
        <parameter name="recipients" type="System.String"
          value="bill@adatum.com, jane@adatum.com" />
        <parameter name="subject" type="System.String"
```

```
          value="Notification from Email EventListener" />
        <parameter name="credentials" type="System.String" value="etw" />
        <parameter name="formatter"
          type="Microsoft.Practices.EnterpriseLibrary.SemanticLogging
            .Formatters.EventTextFormatter,
            Microsoft.Practices.EnterpriseLibrary.SemanticLogging">
          <parameters>
            <parameter name="header" type="System.String"
              value="**************************************"/>
          </parameters>
        </parameter>

      </parameters>
    </customEventListener>
  </eventListeners>

  <formatters>
    <eventTextFormatter name="text"
      header="============================================="/>
  </formatters>

</configuration>
```

In this example, the class named **SimpleEmailEventListener** defines the custom event listener. The **parameter** elements specify the constructor arguments in the same order that they appear in the constructor. A **parameter** element with nested parameters specifies the formatter to use.

You define an out-of-process event listener differently from an in-process listener: an out-of-process event listener implements the **IEventListener** interface.

#!SPOKENBY(Markus)!#

A custom in-process event listener extends the **EventListener** class; a custom out-of-process listener implements the **IEventListener** interface.

The following code sample shows the **SimpleEmailEventListener** class.

C#

```
public class ConsoleEventListener
{
  public sealed class SimpleEmailEventListener : IEventListener
  {
    private const string DefaultSubject = "EmailEventListener";
    private IEventTextFormatter formatter;
    private MailAddress sender;
    private MailAddressCollection recipients =
      new MailAddressCollection();
    private string subject;
```

```
    private string host;
    private int port;
    private NetworkCredential credentials;

    public SimpleEmailEventListener(string host, int port,
      string recipients, string subject, string credentials,
      IEventTextFormatter formatter)
    {
      this.formatter = formatter ?? new EventTextFormatter();
      this.host = host;
      this.port = port;
      this.credentials =
        CredentialManager.GetCredentials(credentials);
      this.sender = new MailAddress(this.credentials.UserName);
      this.recipients.Add(recipients);
      this.subject = subject ?? DefaultSubject;
    }

    public void OnEventWritten(EventEntry entry)
    {
      using(var writer = new StringWriter())
      {
        this.formatter.WriteEvent(entry, writer);
        Post(writer.ToString()).ConfigureAwait(false);
      }
    }

    private async Task Post(string body)
    {
      using (var client = new SmtpClient(this.host, this.port)
        { Credentials = this.credentials, EnableSsl = true })
      using (var message = new MailMessage(
        this.sender, this.recipients[0])
        { Body = body, Subject = this.subject })
      {
        for (int i = 1; i < this.recipients.Count; i++)
          message.CC.Add(this.recipients[i]);
        client.SendCompleted += (o, e) =>
          Trace.WriteIf(e.Error != null, e.Error);
        await client.SendMailAsync(message).ConfigureAwait(false);
      }
    }
  }
}
```

#!SPOKENBY(Markus)!#

If you are using the **customEventListener** element in the XML configuration file, the order of the **parameter** elements must match the order of the constructor arguments.

Creating a Custom Out-of-Process Event Listener with Intellisense Support

To make it easier to configure the custom event listener, you can add support for Intellisense when you edit the XML configuration file. You can use the same **SimpleEmailEventListener** class shown in the previous section.

The following XML sample shows the XML that configures the custom listener and that supports Intellisense in Visual Studio.

XML

```
<?xml version="1.0"?>
<configuration xmlns=
  "http://schemas.microsoft.com/practices/2013/entlib/semanticlogging/etw">

  <traceEventService sessionName="Adatum Outproc Listener" />

  <eventSources>
    <eventSource name="Adatum">
      <eventListeners>
        <eventListener name="ConsoleEventListener" level="LogAlways"/>
        <eventListener name="email" level="Critical"/>
      </eventListeners>
    </eventSource>
  </eventSources>

  <eventListeners>
    <consoleEventListener name="ConsoleEventListener" formatterName="text"/>
    <emailEventListener xmlns="urn:demo.etw"
                        name ="email"
                        host="smtp.live.com"
                        port="587"
                        credentials="etw"
                        recipients="bill@adatum.com, jane@adatum.com "
                        subject="Demo from Email EventListener"
                        formatterName="text"/>
  </eventListeners>

  <formatters>
    <eventTextFormatter name="text"
      header="============================================="/>
  </formatters>

</configuration>
```

This example uses the custom **emailEventListener** element in a custom XML namespace. An XML schema defines this namespace, and enables the Intellisense support in Visual Studio. The following XML sample shows the schema.

XML

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<xs:schema id="EmailEventListenerElement"
    targetNamespace="urn:demo.etw"
    xmlns="urn:demo.etw"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

  <xs:element name="emailEventListener">
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required" />
      <xs:attribute name="host" type="xs:string" use="required" />
      <xs:attribute name="port" type="xs:int" use="required" />
      <xs:attribute name="credentials" type="xs:string" use="required" />
      <xs:attribute name="recipients" type="xs:string" use="required" />
      <xs:attribute name="subject" type="xs:string" use="optional" />
      <xs:attribute name="formatterName" type="xs:string" use="optional" />
    </xs:complexType>
  </xs:element>

</xs:schema>
```

You should place this schema file in the same folder as your XML configuration file (or in a subfolder beneath the folder that holds your XML configuration file).

In addition to the schema file, you also need a class to enable the Semantic Logging Application Block to read the custom configuration data from the XML configuration file. This **EmailEventListenerElement** class must extend the **EventListenerElement** class and use the **XmlRoot** and **XmlAttribute** attributes to map from the XML to the properties as shown in the following example.

C#

```
[XmlRoot("emailEventListener", Namespace = "urn:demo.etw")]
public class EmailEventListenerElement : EventListenerElement
{
  [XmlAttribute("host")]
  public string Host { get; set; }

  [XmlAttribute("port")]
  public int Port { get; set; }

  [XmlAttribute("credentials")]
  public string Credentials { get; set; }

  [XmlAttribute("recipients")]
  public string Recipients { get; set; }

  [XmlAttribute("subject")]
  public string Subject { get; set; }

 [XmlAttribute("formatterName")]
```

```
  public string Formatter { get; set; }

  public override IEventListener CreateEventListener(
    Func<string, IEventTextFormatter> formatterFactory)
  {
    return new EmailEventListener(this.Host, this.Port,
      this.Recipients, this.Subject, this.Credentials,
      formatterFactory(this.Formatter));
  }
}
```

The override of the **CreateEventListener** method instantiates the custom **EmailEventListener** event listener class.

Place the assembly that contains your **EventListerElement** and **IEventListener** implemenations is the same folder as the XML configuration file.

Creating Custom Event Listener Host Applications

The Semantic Logging Application Block includes an event listener host application to use when you want to use the block out-of-process: this application can run as a console application or as a Windows Service. Internally, both of these applications use the **TraceEventService** and **TraceEventServiceConfiguration** classes.

You can create your own custom out-of-process event listener using these same two classes. The following code sample from the out-of-process console host illustrates their use.

C#

```
using Microsoft.Practices.EnterpriseLibrary.SemanticLogging
  .Etw.Configuration;
using System;
using System.Configuration;
using System.Diagnostics;
using System.Diagnostics.Tracing;
using System.Text;

namespace Microsoft.Practices.EnterpriseLibrary.SemanticLogging.Etw
{
  internal class Application
  {
    private const string LoggingEventSourceName =
      "Logging";
    private const string EtwConfigurationFileName =
      "EtwConfigurationFileName";
    private static readonly TraceSource logSource =
      new TraceSource(LoggingEventSourceName);

    internal static void Main(string[] args)
    {
```

```
      try
      {
        ...

        var configuration =
          TraceEventServiceConfiguration.Load(
          ConfigurationManager.AppSettings[EtwConfigurationFileName]);
        using (var service = new TraceEventService(configuration))
        {
          ...

          service.Start();

          ...


        }
      }
      catch (Exception e)
      {
        ...
      }
      finally
      {
        logSource.Close();
      }
    }

    ...
  }
}
```

You should consider adding a mechanism that monitors the configuration file for changes. The Out-of-Process Host application included with the block uses the **FileSystemWatcher** class from the .NET Framework to monitor the configuration file for changes. When it detects a change it automatically restarts the internal **TraceEventService** instance with the new configuration settings.

Summary

The Semantic Logging Application Block, while superficially similar to the Logging Application Block, offers two significant additional features.

Semantic logging and strongly typed events ensure that the structure of your log messages is known. This makes it much easier to process your log messages automatically whether because you want to analyze them, drive some other automated process from them, or correlate them with another set of events. Creating a log message in your code now becomes a simple case of calling a method to report that an event of some significance has just happened in your application.

You can use the Semantic Logging Application Block out-of-process to capture and process log messages from another application. This makes it possible to collect diagnostic trace information from a live application while minimizing the overhead of logging on that application. The block makes use of Event Tracing for Windows to achieve this: the production application can write high volumes of trace messages that ETW intercepts and delivers to your separate logging application. Your logging application can use all of the blocks event listeners and formatters to process these log messages and send them to the appropriate destination.