|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EventSource Dynamic Event Specification  **C:\Users\branbray\Desktop\VS_ParticleWave_White.png** | | | | | |
|  | **Spec Status:** Complete | | | **Last updated:** 8/10/2015 7:56 AM | |
|  | | | | | |
| **Product Unit** | |  | **FEATURE TEAM** | |  |
|  | | | | | |

EventSource

Table of Contents

1. Specifying Events Statically (at Compile Time) 1

2. New EventSource Dynamic Events 2

3. Dynamic Events and ETW 3

3.1 Support for Reading Self-Describing Events 4

4. Guidance 5

# Specifying Events Statically (at Compile Time)

By design System.Diagnostics.Tracing.EventSource strongly encourages a ‘contract’ style for defining events. In this style you defined a method for each distinct event **at compile time** such as in this example for a ‘Load’ and ‘Unload’ events

[EventSource(Name = "Samples-EventSourceDemos-RuntimeDemo")]

public sealed class RuntimeDemoEventSource : EventSource

{

// define the singleton instance of the event source

public static RuntimeDemoEventSource Log = new RuntimeDemoEventSource();

// define the Load Event. Calling this method logs the event

public void Load(long baseAddress, string imageName) { WriteEvent(1, baseAddress, imageName); }

// define the Unload Event. Calling this method logs the event

public void Unload(long baseAddress) { WriteEvent(2, baseAddress); }

}

Once the methods were defined you could then use then in your code as the appropriate place.

RuntimeDemoEventSource.Log.Load(0x40000, "MyFile0");

...

RuntimeDemoEventSource.Log.Unload(0x40000);

This style has a number of important advantages (all related to this ‘contract’ notion) including:

1. It keeps the source code change at the call site to an absolute minimum. The only thing at the call site are passing field values. Everything else (the verbosity, keywords, format strings etc) are handled INSIDE the EventSource, thus making it easy to keep consistency.
2. Because the structure of the events was known at compile time, it could be optimized at compile time, making the tracing infrastructure very efficient.
3. The EventSource acts as an API contract between the code being instrumented and logging system. This encourages being thoughtful about versioning of the events, so that logging code that depends on the events to does not break as the instrumented code changes (and perhaps more information needs to be added to the events)
4. It allows a declarative specification of all know events and their shape to be compiled. This ‘manifest’ of events that can be logged is mandated for systems like Windows Event Tracing for Windows (ETW), but it is useful in general when consumption code is written (you know the events and data you need to handle).

However the big disadvantage of this ‘contract’ approach is that it requires the events names and signatures to be known at compile time. The guidance in these situations is to ‘do what you can at compile time’ and serialize the rest. Thus in the worst case you could have a single event called ‘Log’, in which you pass a two parameters: a string ‘eventName’ and another string, ‘JSONData’ which contains the payload to be logged as a JSON serialized string. This at least allows you to log ‘anything’ even if its structure is only know at runtime, but clearly is less convenient since now every logging call needs to call a JSON serializer before doing the actual logging (which will be less efficient). EventSource’s new support for Rich types (see Rich Types Spec), make handling this dynamic case even easier (no need to use a ‘external’ serializer).

But fundamentally EventSource’s design is really predicated on the belief that this ‘very dynamic’ case is very rare. It **IS** a good that you specify all that you can at compile time because it makes the contract ‘stronger’ allowing log consumers to take advantage of that consistency. In the same way it is useful ‘most code’ to be strongly typed in the language and only use ‘object’ and dynamic typing as needed, it is useful to for as much of your logging system to be strongly typed, and reserve dynamic typing (some sort of serialization (e.g. JSON)) for those cases the really needed it. This is why using the contract style for EventSource is still a best practice.

However there are some scenarios where the compile-time nature of EventSource contract specifications is awkward. In particular

1. The module where events are logged versions at a different rate than the module that contains the EventSource subclass. This makes it problematic for new events to be added (since typically the owner of the instrumented code cannot update the EventSource)
2. Many EventSource’s are created whose names are determined at compile time
3. The event names are not known until run time (this happens if you are trying to tunnel from a dynamic language).

Because EventSource was meant to be ‘The’ way of logging events, ANY problematic case is a problem. Thus we needed it to handle ALL scenarios, including very dynamic ones where information is simply not known until runtime.

# New EventSource Dynamic Events

The basic idea behind dynamic events is to make it POSSIBLE to completely define a new EventSource as well as all of its events using only dynamic information. Thus everything that was specified before with compile time artifacts (like subclass definitions, method declarations, and custom attributes) can also be specified by using APIs where data is passed as parameters.

This is done by

1. Adding PUBLIC (not protected) EventSource Constructor that takes a string which is the name of the EventSource
2. Adding Write<T> to EventSource, which can be passed all the information needed to log an event as arguments
3. Provide Axillary classes (e.g. EventSourceOptions) to facility passing arguments to Write<T>

Using this dynamic capability we can implement the ‘Load’ and ‘Unload’ events as follows:

You don’t need to define a subclass to define an EvenSource, you can call the constructor and provide the EventSource name as an argument.

static EventSource runtimeDemoEventSource = new EventSource("Samples-EventSourceDemos-RuntimeDemo");

And then you can use the Write API to log an event, passing sufficient information at the call site to define the event.

runtimeDemoEventSource.Write("Load", new { baseAddress = 0x40000L, imageName = "MyFile0" });

//...

runtimeDemoEventSource.Write("Unload", new { baseAddress = 0x40000L });

Notice that the Write API really depends heavily on the ‘anonymous type’ feature in C# and VB. The syntax

new { baseAddress = 0x40000, imageName = "MyFile0" }

Defines a new type that looks like

class UnnamedType\_1

{

long baseAddress { get; set; }

string imageName { get; set }

}

Which defines a new (unnamed) type with field-backed properties called ‘baseAddress’ and ‘imageName’ and then creates an instance of this types where these fields are set to 0x40000 and “MyFile0”.

Thus all the things that used to be determined by inspecting the method name and signature (which had to be generated at compile time), are now available by inspecting the arguments at run time. Thus no classes need to be defined (explicitly at least),

There are overloads for the Write<T> API that allow you to pass an ‘EventSourceOptions’ structure that defines things that would normally have been specified with custom attributes like verbosity, keywords, opcode etc. For example if we had wanted the ‘Unload’ event be tagged as ‘Verbose’ instead the default ‘informational’ verbosity, we could have called it like this.

runtimeDemoEventSource.Write("Unload",

new EventSourceOptions{ Level = EventLevel.Verbose },

new { baseAddress = 0x40000 });

Notice that logically what is passed to ‘Write’ as the payload is exactly one object (it is actually a Generic T, not object). The idea is that because of the anonymous type support, one object is always enough (simply add the fields you need on the fly). This is nice when you wish to ‘wrap’ the EventSource APIs in some other abstraction. The signature of the wrapper will just be parameterized by a T to all it to ‘pass through’ to the EventSource.

# Dynamic Events and ETW

One Important consequence of dynamic events is that you can no longer create a ‘manifest’ that describes the possible events before they happen. The normal manifest-based Event Tracing for Windows (ETW) serialization depends on being able to build such a manifest, which means that the new dynamic events can’t work with normal ETW serialization. In Windows 10 (and it should be back-ported to Windows 8 and Windows 7), the ETW system was extended to support a different kind of serialization which does not need manifests (so called self-describing event serialization). This form of serialization must be used for dynamic events. Thus dynamic events may not have perfect compatibility with existing ETW based tools.

Thus while it is true that Dynamic events work the same as ‘contract based’ events when being consumed by an EventListener, it is not true that they are the same when being consumed via ETW because dynamic events use a different kind of serialization (manifest based for ‘contract-based’ events that are defined with methods and ‘self-describing’ event serialization for events defined with Write<T>.

Generally using two different forms of ETW serialization in the same provider is not a good idea, since there are simply more ways for things go wrong. Since it is not possible to Write<T> to use manifest based serialization but it IS possible for contract-based events to use self-describing serialization the obvious solution is to ask the EventSource to use self-describing serialization for all events regardless of how the event was specified.

You can do this by specifying the EventSourceSettings.EtwSelfDescribingEventFormat when constructing the EventSource. For example if we added the code in red below

[EventSource(Name = "Samples-EventSourceDemos-RuntimeDemo")]

public sealed class RuntimeDemoEventSource : EventSource

{

// define the singleton instance of the event source

public static RuntimeDemoEventSource Log = new RuntimeDemoEventSource();

// Indicate that Self-Describing format should be used when sending data to ETW.

private RuntimeDemoEventSource() : base(EventSourceSettings.EtwSelfDescribingEventFormat) {}

// define the Load Event. Calling this method logs the event

public void Load(long baseAddress, string imageName) { WriteEvent(1, baseAddress, imageName); }

// define the Unload Event. Calling this method logs the event

public void Unload(long baseAddress) { WriteEvent(2, baseAddress); }

}

This would insure that the code

RuntimeDemoEventSource.Log.Load(0x40000, "MyFile0");

And

runtimeDemoEventSource.Write("Load", new { baseAddress = 0x40000L, imageName = "MyFile0" });

Would serialize in exactly the same way when sent to ETW.

## Support for Reading Self-Describing Events

At the present time (8/2015) the self-describing events are supported in

1. Versions of [Windows Performance Toolkit](http://social.technet.microsoft.com/wiki/contents/articles/4847.install-the-windows-performance-toolkit-wpt.aspx) WPR.exe with a major version > 10
2. In [PerfView](http://www.microsoft.com/en-us/download/details.aspx?id=28567) performance tool Version 1.7.28 or later (V1.8 to be published by 9/2015)
3. By the OS [TDH APIs](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363795(v=vs.85).aspx), for Windows 10 (or backported to Windows 8 or 7)
4. The .NET ETW parsing package [TraceEvent Nuget Package](https://www.nuget.org/packages/Microsoft.Diagnostics.Tracing.TraceEvent/) Version 1.1.35 or later. See [this blog](http://blogs.msdn.com/b/vancem/archive/2014/03/15/walk-through-getting-started-with-etw-traceevent-nuget-samples-package.aspx) for a walk-through of samples.

It is likely that in the future the self-describing event format will become the ‘default’ format for ETW, however in the near term some tools may not work well with it. Thus in the short term changing the format only makes sense if you need to use the dynamic features. However, in the future, it is likely that using the self-describing event format by default will be a good choice.

# Guidance

The new Write() API does allow more ‘at runtime’ control, however it is not without its costs. In particular it scatters the specification of the events to the code sites, making them more complex, and making it to understand the ‘contract’ between source of the events and its consumers. It also will not be as efficient.

For these reasons, we still recommend using the ‘contract based’ approach whenever that is possible. Only in those cases where you really need the ability to specify things at runtime, should you be using Write<T>(). Using the contract based approach is especially relevant in the short term because the Write<T> API cannot support manifest based ETW, where the contracted based approach supports both the manifest and self-describing event format.