

Windows Management Instrumentation: Create WMI Providers to Notify Applications of System Events

Article • 10/24/2019 21 minutes to read



Windows Management Instrumentation: Create WMI Providers to Notify Applications of System Events

J. Andrew Schafer

This article assumes you're familiar with WMI, C++, and COM

Level of Difficulty 1 2 3

Download the code for this article: AppLog.exe (69KB)

Browse the code for this article at **Code Center**: WMI_AppLog

SUMMARY Windows Management Instrumentation (WMI) is based on an industry-wide standard for notifications used to manage objects and devices across a network. By receiving WMI events, an application can be notified of changes to data in WMI itself. This allows the developer to notify the consuming application that certain system configuration data has changed, without the application having to poll WMI continuously for this data. The author presents an explanation of the different types of events in WMI, then goes on to develop an event provider.



Windows Management Instrumentation (WMI) is the Microsoft implementation of the Web-based Enterprise Management (WBEM) initiative—an industry initiative for standardizing the conventions used to manage objects and devices on many machines across a network or the Web. WMI offers a great alternative to traditional managed storage mediums such as the registry, disk files, and even relational databases. The flexibility and manageability of WMI are among its

greatest strengths.

Besides simply storing data, WMI has the power to tell consumers when that data has changed. This ability has many practical applications. In this article, I'll describe how to take advantage of this capability by developing an event provider and using events in your production environment.

If you are new to WMI, I suggest you read two articles in the May 2000 issue of *MSDN Magazine*: "Administering Windows and Applications across an Enterprise" and "Building a WMI Provider to Expose Your Object Info".

Building an Event Provider

The example that I will use in this article is a WMI event provider that generates events for each WMI instance of a custom WMI class. A WMI class provides a structural template for each instance. This class, and therefore each of the associated instances in my provider, contains configuration information on how and where to log messages. For each application that is using the logger, one instance is created to store the configuration information.

I chose a logger because the information from a logger is something that anyone working in an enterprise development environment would find useful. For the logger presented in this article, WMI instances represent the app that the logger is tracking. Therefore, the values of each WMI instance are in turn associated with the application represented by that instance. Properties can, for instance, specify what information is logged and where it goes. An enterprise manager or developer might want to control these values; the presented conceptual structure is an important intuitive foundation upon which to build the WMI provider.

Most traditional configuration implementations require an application to keep the same configuration settings for its lifetime, so the application doesn't have to continuously poll the configuration database for changes. By storing configuration data with WMI, you can capture events that tell the application when configuration changes occur and therefore allow near real-time response.

Event Types

If you're going to capture events from WMI, you'll need to know what types of events can be generated. In WMI, events are represented as instances of WMI classes. WMI supports three types of events from event providers: timer events, intrinsic events, and extrinsic events.

Timer events These can occur in two ways: once at a predetermined time, or periodically. Timer events are special in that they are only generated after being explicitly set up by a consumer. I will not be covering timer events in this article, so for more information on this subject, see the WMI platform SDK documentation.

Intrinsic events These events are generated in response to changes to data in WMI. Specifically, intrinsic events are fired when a namespace, class, or instance is created, modified, or deleted.

In most cases, every namespace, class, or instance stored in the WMI repository supports intrinsic events through the default WMI implementation. This is accomplished entirely by WMI itself rather than anything special that the developer does.

When an event consumer registers to receive intrinsic events of data items stored statically in the WMI repository, WMI will begin scanning the repository for data changes. When a change in

data occurs, WMI will generate an intrinsic event if the data change corresponds to a registered event.

Some data items in WMI are dealt with dynamically through their own provider. When an event consumer registers to receive intrinsic events of data items that are created dynamically, coding becomes a little more complicated. First, WMI will check for the existence of an event provider that supports the consumer's intrinsic event query. If such an event provider exists, then WMI will let the event provider take responsibility for the intrinsic event. More on how this works later.

However, if a supporting event provider is not available, WMI will use its own internal version. WMI does this by polling the dynamic data for changes. Because polling is used, it is better to have a supporting intrinsic event provider if performance is a concern. WMI will poll the data at an interval based on information provided in the query.

Intrinsic events are represented as instances of a special set of system classes. The class names and descriptions are listed in Figure 1 . If a provider does provide intrinsic events, the provider must supply the events for all of the instances of a class.

Extrinsic events These are user-defined events. WMI provides this facility so that in cases where intrinsic events are unable to provide the information needed, extrinsic events can be used in a manner that best suits these needs. Intrinsic events manifest themselves as instances of a user-defined class.

Extrinsic events are always generated through an event provider or client, never by WMI itself. Therefore, all instances of an extrinsic event are dynamic. In fact, trying to generate a static instance from an event class will fail.

Extrinsic events are derived from the abstract system class `__ExtrinsicEvent`. As mentioned earlier, each intrinsic event class must have a properly registered provider to function correctly unless the class serves as an abstract base class for other extrinsic event classes. Any number of classes may be derived from the `__ExtrinsicEvent` class. Furthermore, any number of classes may be derived from these classes, and so on. However, anytime a class is derived from another, the parent must be an abstract class that is not supported by any provider. Once a provider has been registered for a class, no further classes may be derived from it.

Event Consumers

WMI clients can become temporary event consumers by subscribing to WMI events. Note that I say temporary because there is a concept of permanent events in WMI, but those events are outside the scope of this article. When subscribing to an event, the client specifies the event from which it must receive info, the data it wants to get, and the conditions under which the event should be sent.

To specify which events you want to receive, you can use the subscription builder, which designates the events using WMI Query Language (WQL). Subscribing to events using WQL is exactly like making a WMI query for any object, except that the query returns results to the caller.



Figure 2 Relationships

An event consumer must be set up to receive the events that the subscription builder created. The event provider is the piece that generates the events themselves. WMI acts as the manager between the event consumer and the event provider. The relationship between these components is shown in **Figure 2**. Here, I will often refer to event consumers and subscription builders as one in the same.

The Event Provider

The provider I've written generates instance types of intrinsic events as well as its own custom extrinsic event. To do this, the provider has to know when instances were created, modified, and deleted. The only practical way to get this information is to work in conjunction with an instance provider.

Creating an instance provider using the provider framework has been discussed in previous articles in *MSDN Magazine*. However, I will set up some background for my provider.

The class that my instance provider will serve is the `MSJ_AppLogger` class. The complete listing of the Managed Object Format (MOF) file is available in [Figure 3](#). Half of the MOF descriptors in [Figure 3](#) are used for the instance provider that my event provider requires. The instance provider supports the `MSJ_AppLogger` class. When `MSJ_AppLogger` instances are created, modified, or deleted, my event provider will generate the corresponding events.

As I've mentioned, my provider supports all the intrinsic events that deal with instances, and a special extrinsic event that is generated when an instance is modified. The `__ExtrinsicEvent` derived event class that I used in my provider is named `MSJ_AppLoggerEvent`; its schema can also be found in [Figure 3](#). It has the exact same properties as the `MSJ_AppLogger` because the event itself is a reflection of the `MSJ_AppLogger` instance that was modified.

At this point you're probably wondering how WMI knows which events and WMI classes an event provider supports. This is a property of the `__EventProviderRegistration` instance. Each instance of `__EventProviderRegistration` has a property named `EventQueryList`. This property is an array of strings representing each event query that is supported. In this application, `EventQueryList` looks like this:

```

EventQueryList = {"SELECT * FROM MSJ_AppLoggerEvent",
  "SELECT * FROM __InstanceCreationEvent WHERE TargetInstance
    ISA \"MSJ_AppLogger\"",
  "SELECT * FROM __InstanceModificationEvent WHERE TargetInstance
    ISA \"MSJ_AppLogger\"",

```

```
"SELECT * FROM __InstanceDeletionEvent WHERE TargetInstance  
ISA \"MSJ_AppLogger\\\"";
```

Each query listed in the string array is a skeleton framework for the query that the name provider supports. A query can support much more complex parameters, but as long as it builds upon a query listed here, this provider will support it. WMI will take care of the additional parameters and filtering, if needed.

The first query listed in the previous code corresponds with the MSJ_AppLoggerEvent extrinsic event. The remaining three queries are all intrinsic events.

I built the supporting instance provider using the provider framework. However, no such framework exists yet for event providers. So to write an event provider, you have to get your hands dirty and use good old-fashioned COM.

Because the event provider I am writing will generate events as a result of changes to the MSJ_AppLogger instance data, the event provider code will run in the same process as the instance provider. However, it is perfectly acceptable to make your event provider run out-of-process.

IWbemProviderInit

The IWbemProviderInit interface must be implemented by your event provider. It has one method—Initialize:

```
HRESULT IWbemProviderInit::Initialize (LPWSTR pszUser,  
    LONG lFlags, LPWSTR pszNamespace, LPWSTR pszLocale,  
    IWbemServices *pNamespace, IWbemContext *pCtx,  
    IWbemProviderInitSink *pInitSink)
```

Initialize is called the first time WMI needs your provider to prepare to start providing events.

If you look back at the MOF listing in Figure 3 , you will see several properties that are set for the descriptor that creates an instance of __Win32Provider. The properties PerUserInitialization, PerLocaleInitialization, InitializationReentrancy, and InitializeAsAdminFirst all affect initialization. If you are familiar with their meaning from programming for other types of providers, you will be interested to hear that the assigned values will have no effect on event providers. This is because WMI calls Initialize only once for event providers.

The astute reader may wonder if the pszUser and pszLocale parameters have any meaning for event providers, given that Initialize is called only once and the initialization properties of the __Win32Provider instance are ignored. The answer is that for an event provider these will always be NULL. WMI expects the event provider to generate events that are independent of the user or locale contexts.

The other parameters in the Initialize method will probably be used by most event providers. If you look at my implementation of Initialize in Figure 4 , you will see some code that looks like this:

```
IWbemClassObject *pObj = NULL;  
HRESULT hRes = m_pNs->GetObject(  
    CBSTR(L"MSJ_AppLoggerEvent"),  
    0, pCtx, &pObj, 0);
```

This snippet uses the WMI Web services interface (pNamespace) that's passed in through the Initialize call and asks WMI for a class definition interface to the MSJ_AppLoggerEvent object. Through this interface, an event provider can generate WMI instances of the MSJ_AppLoggerEvent class. As you will recall, each event is an instance of an event class. So, by using the IWbemClassObject interface that contains the definition of my event class, I can spawn instances whenever an event must be generated.

My implementation of Initialize stores this class definition interface and uses it later when generating events. The class object interface has a method that easily allows it to generate the event instances needed. Because of this, it is common practice for event providers to retrieve, in the Initialize method, all class definitions from WMI for all event instances that they are providing.

Once the required initialization has been performed, the event provider must let WMI know the result of the initialization. One parameter to the Initialize method is a pointer to the IWbemProviderInitSink interface. Through this interface, the event provider must call SetStatus to let WMI know whether the event provider is fully capable of servicing event requests from applications, WMI, and even other providers. Here is the relevant code from my provider, as shown in Figure 4 :

```
pInitSink->SetStatus(WBEM_S_INITIALIZED,0);
```

Using WBEM_S_INITIALIZED tells WMI that the provider initialized successfully. If the initialization failed, the provider would have passed back WBEM_E_FAILED. The second parameter is reserved and must always be zero.

The Initialize method itself must also return a value indicating the result of initialization. WBEM_S_NO_ERROR should be returned for success and WBEM_E_FAILED should be returned when failure occurs. A call to SetStatus is not required if Initialize returns a result code indicating failure. However, the opposite is not true; SetStatus must also be called if Initialize returns a result

code signifying success. Failure to call `SetStatus` when `Initialize` returns success will block all event consumers who attempt to register for events.

IWbemEventProvider

Once WMI receives a success indication for both operations, it will make a call to `QueryInterface` for the provider's primary interface. The primary interface for an event provider is `IWbemEventProvider`. WMI retrieves this interface in response to a query from an event consumer to register for events for which the associated provider is responsible.

This interface has but a single method, `ProvideEvents`, which is defined as follows:

```
HRESULT IWbemEventProvider::ProvideEvents(  
    IWbemObjectSink *pSink,  
    long lFlags  
);
```

`ProvideEvents` is called by WMI as a signal to the event provider to begin providing events. This call includes a parameter that is an `IWbemObjectSink` pointer that points back to WMI. The provider should save this to use later when delivering events. The `lFlags` parameter is reserved for future use and will be zero.

One important thing to note about the call to `ProvideEvents` is that WMI expects the provider to return as soon as possible when it calls this method. A provider is not permitted to block this call for more than a few seconds.

If `ProvideEvents` succeeds in setting up the provider to begin event delivery, the method should return `WBEM_S_NO_ERROR`. Otherwise, it should return `WBEM_E_FAILED`. After this method is called, a provider should do what is necessary to begin delivering events. This usually means creating a worker thread that processes events as they become available.

My implementation of `ProvideEvents`, shown in Figure 5 , is fairly simple. Note the storage of the `IWbemObjectSink` pointer and the creation of the worker thread. Most event providers will have an implementation that is very similar to this one.

The InstanceThread Function

The thread function that is supplied to `CreateEvent` in the `ProvideEvents` implementation shown in Figure 5 simply calls the `InstanceThread` function in Figure 6 . I use a class method so that my provider has easy access to the class member variables.

It is from the `InstanceThread` function that all the events are processed and then passed to WMI. Once started, the `InstanceThread` function will go into a loop until the provider is shut down.

In this loop, the event provider will wait until signaled that processing of an event is pending. Pending events are placed by the instance provider into a queue that is defined as follows:

```
std::queue<AppLoggerEvent*> g_queueAppLoggerEvents;
```

The queue stores a data structure that defines the events. The data structure is defined in Figure 7 .

As you can see, this mimics the definition of the MSJ_AppLogger and MSJ_AppLoggerEvent WMI classes themselves. When the instance provider I mentioned earlier processes a change to MSJ_AppLogger instances, it will place a pending event on the queue. If you are interested, you may take a look at the online code samples for more detail about this process of placing the items onto the queue (see the link at the top of this article). However, this detail is not important to understanding the event provider itself.

After an item is placed on the queue, the worker thread in the event provider processes each pending event. My provider determines the event type to generate, corresponding to the type of the item in the queue. The code block that does this is shown in Figure 6 . There are three types of events that correspond to the three types of intrinsic events I'll be generating: creation, modification, and deletion. I also generate an extrinsic event for modification.

One thing I did in my provider is to make the extrinsic and intrinsic events convey exactly the same information. The purpose of this is to demonstrate how to generate either type of event. In your own provider, you would probably use extrinsic events for something more practical.

Let's focus on the generation of an extrinsic event. Take a look at this code block for the modification event:

```
hres = m_pAppLoggerEventClassDef->SpawnInstance(0,
    &(aryEventInstances[0]));
FillInProperties( aryEventInstances[0], pALInstance );
lObjectCount++;
```

Simple enough? The m_pAppLoggerEventClassDef is a variable you should recognize as an interface preserved from the provider's Initialization routine. Remember that it was during Initialize that the provider queried WMI for all the class definitions that I plan to use during event generation.

The class object interfaces have a method that can spawn an instance. Because WMI events are just instances of the event class, you can call SpawnInstance on the class object interface to create an instance of each event.

The FillInProperties function populates the new instance with the correct values from the event item that was in the queue. That's all it takes to create the event instance. In a bit, I'll show how to send that event to WMI, where it's distributed to event subscribers.

The next section of the modification event code is the creation of the intrinsic event.

```
m_pInstanceModEventClassDef->SpawnInstance(0,
    &aryEventInstances[1]);
m_pAppLoggerClassDef->SpawnInstance(0, &pInstance);
m_pAppLoggerClassDef->SpawnInstance(0, &pPrevInstance);
```

As with the extrinsic event I just covered, the intrinsic event generation will begin by spawning an instance for the single intrinsic modification event that will be generated. But you will notice that SpawnInstance is called three times instead of once. You might be asking why I spawned two additional instances if I am only going to generate one intrinsic event for modification. The answer is that intrinsic events all have at least one property that references an instance that describes the event itself. In this example, as with the extrinsic event, the first generated instance is in fact the event itself. But this is an intrinsic event that describes a modification. Such an intrinsic modification event contains two properties: one property references a copy of the instance before it was changed, while the other references the instance after it was changed. Therefore the other two instances created are used as the corresponding properties to the intrinsic event itself.

The remaining lines of code for the modification processing fill in the property values of the three instances just created.

Now that the instances for the events the provider wants to generate have been created, the provider must tell WMI about them. This is done with the IWbemObjectSink method named Indicate:

```
HRESULT IWbemObjectSink::Indicate(
    LONG lObjectCount,
    IWbemClassObject **ppObjArray
);
```

This method has many uses in WMI. For now, I will focus on its use in the context of the event provider. Later you will see that the event consumer also uses it to receive events.

For my provider, the method call is used to tell WMI about generated events. WMI will then process the event and forward it to the appropriate subscribers. The method has two parameters. The first is a count to the number of objects in the second array parameter. The second is the array of IWbemClassObject objects. The design whereby an array is used to pass in more than one

IWbemClassObject interface pointer allows for some optimization both in the event provider and the event consumer. Note that my provider takes advantage of this when generating the two different forms of modification events.

IWbemEventProviderQuerySink

Some readers may note that even if a consumer registers for just a single event class, the provider will generate events for all the classes that it has been designed for. If the consumer only receives events from a particular class, then what happens to the other events? The answer is that WMI checks to see if they have any clients, and if not, it just throws the event out.

The designers of the WMI event system foresaw this as a problem. In my provider, generating these events is a little extra work and requiring WMI to filter them out adds a little more, but all in all it is pretty negligible. Of course, some providers may require a lot of overhead in generating an event and it would be wasteful to have it discarded.

The WMI architecture has an interface, IWbemEventProviderQuerySink, to prevent this kind of scenario. If you implement this interface, you'll give your provider the ability to parse all event queries and decide internally whether it should generate specific events.

IWbemEventProviderQuerySink has two methods that the event provider must implement. The first is NewQuery:

```
HRESULT IWbemEventProviderQuerySink::NewQuery(  
    unsigned long dwId,  
    WBEM_WSTR wszQueryLanguage,  
    WBEM_WSTR wszQuery  
);
```

When this method is called, a unique query ID is passed in as dwId. The query itself will be in the wszQuery parameter, and the wszQueryLanguage parameter will contain the name of the query language used. For the current version of WMI, wszQueryLanguage will always be set to "WQL". Providers should return WBEM_E_FAILED on a call to NewQuery if something failed internally and the provider is unable to process any further event requests; otherwise, WBEM_S_NO_ERROR should be returned.

NewQuery is called every time that WMI identifies the provider as responsible for an event query. In NewQuery, the provider can parse the query to decide which events and even event properties the provider must generate.

Figure 8 provides an example of the implementation of NewQuery that I used in my provider. For my example, I only parsed for class names. However, as the developer of your own provider, you can parse to whatever detail you want.

When my provider finds a class name that it recognizes, it bumps a reference count to some flag variables and stores a mapping of the unique ID to the query string. The flag variables signify that instances of a particular event class should be generated. The ID-to-string map is used so that

my provider can track a query and know later which query was canceled. A query is canceled with the other `IWbemEventProviderQuerySink` method, `CancelQuery`:

```
HRESULT IWbemEventProviderQuerySink::CancelQuery(  
    unsigned long dwId  
);
```

`CancelQuery` is called by WMI when the consumer is finished with a query. Figure 8 also contains my implementation of `CancelQuery`. In `CancelQuery`, a provider can undo any action taken by `NewQuery`. Providers should always return `WBEM_S_NO_ERROR` from a call to `CancelQuery`, regardless of anything that may have happened in the code itself.

The Consumer

Of course, to use the provider you need to create a consumer. In the online samples I have included a simple event consumer and the event provider. This sample will allow the user to input a WQL query or select from a set of existing queries to subscribe to events. When events are received, a listbox will display the type of event that's incoming.

To use a consumer, the event provider must be compiled and registered correctly. Previous articles have explained how to do this properly and because a provider is a COM object, most of you will be familiar with this anyway. Remember that the associated MOF file must be compiled as well.

The consumer starts by connecting to WMI by calling `ConnectServer` on an `IWbemLocator` interface that it retrieved through `CoCreateInstance`. Upon successful connection, the consumer program will have an `IWbemServices` interface pointer.

When the user specifies a query and clicks connect, the consumer will use the `IWbemServices` interface to execute the query by calling the `ExecNotificationQueryAsync` method:

```
HRESULT IWbemServices::ExecNotificationQueryAsync(  
    const BSTR strQueryLanguage,  
    const BSTR strQuery,  
    long lFlags,  
    IWbemContext *pCtx,  
    IWbemObjectSink *pResponseHandler  
);
```

The strQueryLanguage value will always be "WQL" for this version of WMI. strQuery will be the query itself. IFlags allows the client to receive periodic status updates by passing in WBEM_FLAG_SEND_STATUS; otherwise, it should be zero. pCtx will usually just be NULL.

pResponseHandler is a pointer to an IWbemObjectSink interface that is specified by the consumer. Each time an event is generated, the Indicate method of the object pointed to by pResponseHandler will be called by WMI. The values of the Indicate method should be exactly identical to the values passed into Indicate by the event provider. Event providers do not call directly into event consumers, though. As shown in **Figure 2**, WMI still manages event calls between event providers and consumers.

Conclusion

This article only touched on some of the features of the WMI event system, but there is much more information in the WMI SDK. Topics not covered here include event security and advanced WQL event queries. Also not covered are event filters and permanent subscribers, which allow an event consumer to receive notifications that occurred even while the consumer is offline.

Events are an efficient way to tell consumers that something has happened. WMI offers an event system that allows consumers to selectively choose which events they want to receive through well-designed WQL queries. Developers can not only take advantage of WMI as powerful way to store data, but also as a management system that can notify their apps of changes to their data.

J. Andrew Schafer is a solution development consultant at Avanade Inc. (<https://www.avanade.com>). He can be reached at aschafer@bigfoot.com.

From the September 2001 issue of MSDN Magazine.