15

Managing With Windows Management Instrumentation

In this chapter, we cover the following recipes:

* Exploring WMI architecture
* Using the CIM Cmdlets
* Exploring WMI namespaces
* Exploring WMI Classes
* Obtaining WMI Class instances
* Using WMI Methods
* Using WMI events
* Implementing permanent WMI eventing

# Introduction

Windows Management Instrumentation (WMI) is a Windows component you use to help manage Windows systems. WMI is Microsoft’s proprietary implementation of the standards-based of Web-Based Enterprise Management (WBEM), an open standard promulgated by the Distributed Management Task Force. WBEM aims to utilize standards-based Internet technologies to unify the management of distributed computing environments.

In addition to WMI for Windows, there are other implementations of WBEM, including OpenWBEM. You can read more about the DMTF and WBEM at https://www.dmtf.org/about/faq/wbem\_faq, and check out OpenWBEM over at http://openwbem.sourceforge.net/.

Microsoft first introduced WMI as an add-on component for Window NT 4. They later integrated WMI as an essential component of the Windows client, from Windows XP onwards, and Windows Server versions since Windows Server 2000. Subsequently, several feature teams inside the Windows group made heavy use of WMI. Both the storage and networking stacks within Windows, for example, use WMI. Many of the cmdlets, such as Get-NetAdapter, are based directly on WMI.

The WBEM standards originally specified that the components of WMI communicate using HTTP. To improve performance, Microsoft enabled the different components of WMI to communicate using the Common Object Model (COM). WMI is today, based on COM. PowerShell makes use of the WMI features of .NET to enable you to access the features of WMI.

PowerShell 1.0 came with a set of cmdlets that you could use to access WMI. These were basic cmdlets that work in all subsequent versions of Windows PowerShell. However, there is no support for these cmdlets directly in PowerShell 7. You can invoke older WMI cmdlet based scripts on a machine using PowerShell remoting.

With PowerShell V3, Microsoft did some significant work on WMI, resulting in a new module, CimCmdlets. There were several great reasons behind both the new module and the associated updates to some of the WMI internals to assist developers. In this chapter, you make use of the CimCmdlets module to access the features of WMI. You can read more about why the team decided to build new cmdlets in a blog post at https://devblogs.microsoft.com/powershell/introduction-to-cim-cmdlets/.

Before looking at the CimCmdlets and what you can do with them, it is useful to understand the WMI architecture within Windows. The run-time architecture of WMI is the same in Windows 10 and Windows Server 2022. The following diagram shows the WMI conceptual architecture:



Figure 15.1: WMI Architecture

**Insert image B42024\_15\_01.png**

As an IT Pro, you use the CIM cmdlets to access WMI. These cmdlets use .NET to communicate, via transport protocol, with the WMI core and the CIM Object manager on a local or remote host. The transport protocols include DCOM and WSMan. The core of WMI, particularly the CIM Object Manager (CIMOM), are COM components that exist on every Windows host.

The CIMOM stores information in a database referred to as the Common Information Model (CIM) or the CIM database. This database is, in effect, a subset of an ANSI-SQL database. The CIM cmdlets enable you to access the information within the database. The CIM database organizes the information into namespaces of classes. .NET also organizes classes inside namespaces, except for .NET classes, the class name includes then namespace name. With WMI, namespace names and class names are separate and supplied to the CIM cmdlets using different parameters.

WMI classes contain data instances that hold relevant management data. For example, the WMI namespace Root\CimV2 contains the class Win32\_Share. Each instance within this class represents one of the SMB shares within your host. With PowerShell, you would normally use the SMB cmdlets to manage SMB shares. There is useful information contained in other WMI classes for which there is no cmdlet support.

When you retrieve instances of, for example, the Win32\_Share class, .NET gets the instance information and returns it in a .NET wrapper object. Strictly speaking, a WMI object instance is an instance of a specialized .NET class with data returned in WMI. For this reason, you treat WMI objects using the same methods you employ with other .NET Objects.

Many WMI classes have methods you can invoke which perform some operation on either a given WMI instance or statically based on the class. The Win32\_Share class, for example, has a Create() method you can use to create a new share. Each instance of that class has a Delete() method, which deletes the SMB share. A method can either be an instance method acting solely on an instance of a WMI class. WMI class can also have static methods that operate on the class itself. The Delete() method of the Win32\_Share WMI class is an instance method, deleting just a single SMB share. The Create() method allows you to create a new instance of the class, a new SMB share.

An important architectural feature of WMI is the provider. A WMI provider is an add-in to WMI that implements WMI classes inside a given host. The Win32 WMI provider, for example, implements hundreds of WMI classes, including Win32\_Share and *Win32\_Bios*. A provider also implements class methods and class events. For example, the Win32 provider is responsible for performing both the Delete() to delete an SMB share and the Create() method to create a new SMB share.

In production, you are more likely to manage SMB shares using the SMB cmdlets and less likely to use WMI directly. Since SMB shares should be very familiar, they make a great example to help you understand more about WMI, and this chapter's recipes make use of the class.

WMI and WMI providers offer a rich eventing system. WMI and WMI provider classes can implement events to which you can subscribe. When the event occurs, the eventing system notifies you, and you can take some action to handle the event occurrence. For example, you can register for a WMI event that occurs when someone changes the membership of an AD group. When this happens, WMI eventing allows you to take some actions, such as emailing a security administrator informing them of the group’s membership change. WMI also implements permanent WMI events. This feature allows you to configure WMI to trap and handle events with no active PowerShell session running. Permanent events even survive a reboot of the host, extremely powerful in lights-off environments.

There is a lot more detail about WMI than can fit in this chapter. For more details about WMI and how you can interact with it, in more detail, consult with Richard Siddaway’s PowerShell and WMI book (© Manning, Aug 2012 - https://www.manning.com/books/powershell-and-wmi). Richard’s book goes into great detail about WMI, but all the code samples make use of the older WMI cmdlets. You should be able to translate the samples to use the CIM cmdlets. A key value of the book is the discussion of WMI features and how they work. The basic functioning of WMI has not changed significantly since that book was published.

# Exploring WMI Architecture

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Using the CIM Cmdlets

With the first version of Windows PowerShell, Microsoft shipped a set of WMI cmdlets that enabled basic access into WMI, although they were not functionally complete and could be hard to use. The WMI cmdlets also relied on DCOM for accessing WMI on remote machines. Although these initial cmdlets were not complete, workarounds were available for the functionality gaps.

With version 3 of Windows PowerShell, Microsoft did some reengineering of WMI. This simplified the implementation of WMI providers, created a new set of cmdlets, and led a drive to push WMI (WBEM) into Linux. The CIM cmdlets were introduced with PowerShell 3 and provided greatly improved feature coverage. The CIM cmdlets also use different .NET class wrappers, which are smaller. This can reduce the bandwidth when accessing WMI remotely.

Another major improvement is that the CIM cmdlets use WinRM for remote communications. WinRM is easier to secure and can be more lightweight than using DCOM. WinRM is also standards-based, opening up interoperation with Linux and Macintosh hosts. The Open Management Infrastructure project has created an open source version of WMI for Linux.

With the CIM cmdlets, you can create a CIM session with a remote machine. Once PowerShell establishes the session, you can run WMI operations on the remote machine without incurring the setup costs. You can also create a CIM session using DCOM, which allows you communicatExploring WMI architecture

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Exploring WMI namespaces

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Exploring WMI classes

A WMI *class* is a definition of a WMI-Managed object. A WMI class, which  
lives in a namespace, contains members including properties and methods.  
One example is the Win32\_Share class, which defines an SMB share. Although  
you can use WMI to manage SMB shares, you are more likely to use the SMB  
cmdlets described in Chapter 6, “Managing Shared Data.”

A *property* is an attribute of a WMI class. A Win32\_Share has a property  
Name, which is the name of the SMB share. Each property has a data type, such  
as String or Int. The Name property in the Win32\_Share class is String. Some  
classes (and properties) have names that begin with two underscore characters  
(\_\_). These are system classes and properties and generally are not of much use  
to IT professionals, with some exceptions.  
A *method* is some action that a WMI object or class is able to carry out. There  
are two method types you can use: dynamic and static. A *dynamic* method is  
one based on an occurrence of a class. For example, the Delete() method in  
the Win32\_Share class deletes the SMB share related to a specific share object.  
A *static* method is one that is carried out by the class itself. The Win32\_Share  
class has a static method Create(), which you could use to create an SMB share.  
A WMI *event* is created, or fired, whenever a specific event occurs within  
Windows. You can subscribe to events and view the information provided. The  
\_\_ClassOperationEvent, for example, is fired whenever an occurrence of a class  
is added, changed, or removed within a namespace. Events in WMI are powerful, as you can see in “Managing WMI Events.”  
When you use the CIM cmdlets to retrieve class instances, PowerShell returns  
the data wrapped in a .NET object. For example, when you retrieve the SMB  
share objects from the Win32\_Share class, the data representing each SMB share is  
returned contained in an object of the type Microsoft.Management.Infrastructure  
.CimInstance. This approach simplifies retrieving data from WMI, as the objects  
are basically the same as the .NET objects you have used throughout this book.

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Obtaining WMI class instances

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Using WMI methods

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

## How to do it...

## How it works...

## There's more...

# Using WMI events

## Getting Ready

This recipe uses SRV1, a domain-joined host. You have instaled PowerShell 7 and VS code on this host.

.

## How to do it...

## How it works...

## There's more...

# Implementing permanent WMI eventing

## Getting Ready

This recipe uses SRV2, a recently added workgroup host. By default, this host is a DHCP client.

## How to do it...

## How it works...

## There's more...