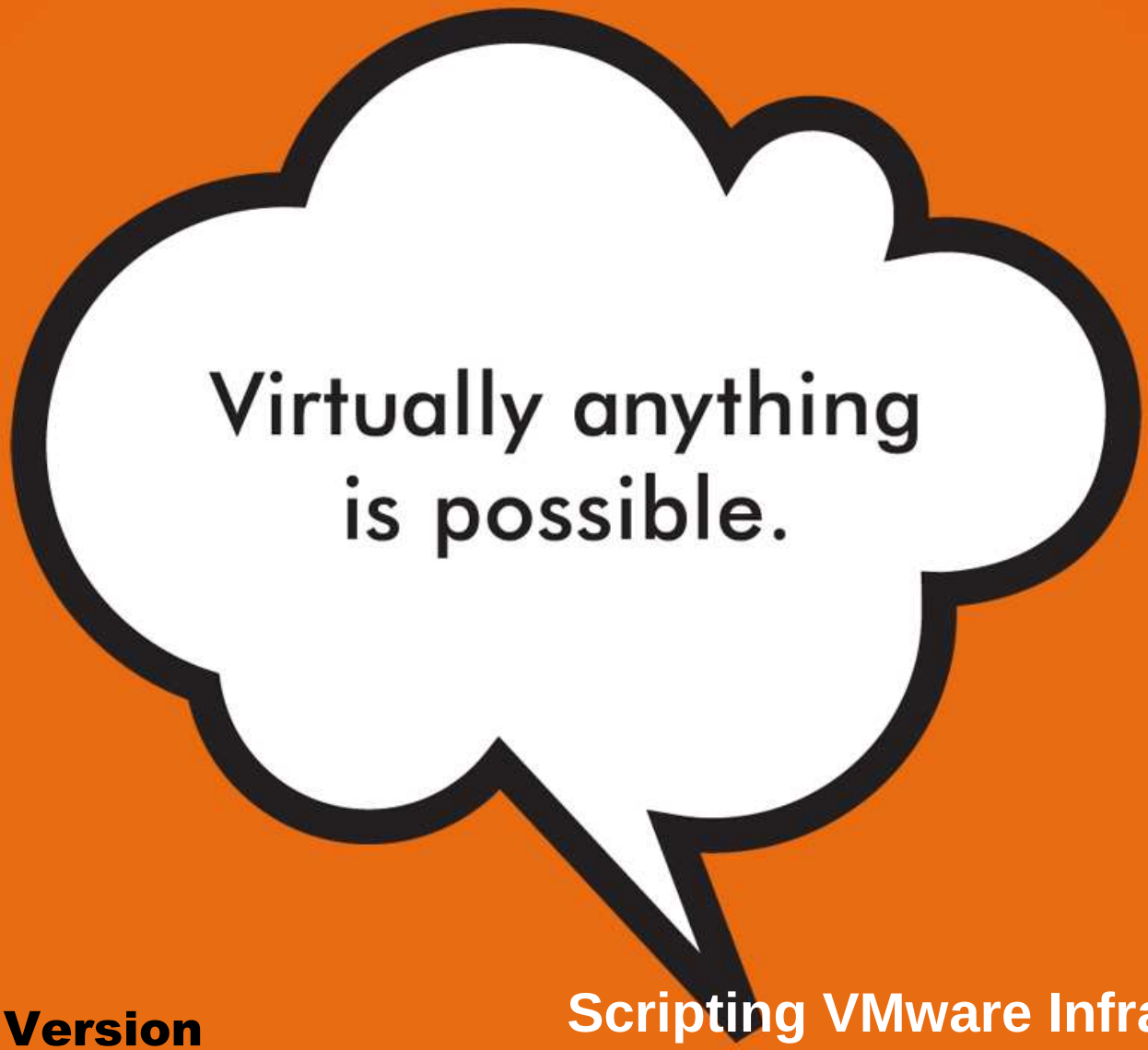
vmworld 2008

Hands-On Lab



Powershell Version

Scripting VMware Infrastructure:

Automating, Integrating, and Extending VMware Infrastructure

Contents

Instructors

Doug Baer-Principal Consultant, IT partners

David Deeths- Senior Technical Alliance Manager, VMware (Team Captain)

Shridhar Deuskar-Technical Solutions Architect Systems Integrators/Systems Outsourcers, VMware

Lisa Guinn-Technical Trainer, VMware

Terry Lyons- Senior Technical Alliance Manager, VMware

Aaron Miller- Senior Systems Engineer, VMware

Alket Memushaj- Senior Consultant Professional Services,VMware

Owen Thomas-Consulting Engineer, New Age Technologies

Brian Watrous- Senior Technical Course Developer, VMware

Alton Yu- Technical Alliance Manager, VMware

We would also like to thank our great tech ops and program management team:

Lupe Vidal (Associate Producer)

Parag Patel (tech-ops lead)

Conor Dolan

Jacob Levin

John Rolla

Michael Hasenkamp

Navin Chhabra

About This Lab

This lab will provide hands-on training for scripting a VMware® environment. The goal of this lab is to provide all the tools that you need to automate, integrate, and extend VC for your environment. For scripting VMware VirtualCenter Server and VMware ESX™, participants can choose to use the VMware Infrastructure Toolkit (for Windows), which is based on Windows PowerShell, or the VMware Infrastructure Perl Toolkit. You will walk away from the lab with a better understanding of VMware Infrastructure, resources for using the VI Toolkits, and a variety of scripts for performing common administrative tasks. Exercises will cover performing tasks, examining virtual machine attributes, performing actions on many virtual machines at a time, and exporting performance data. This lab does not require programming or scripting experience.

Introduction

Welcome to the ***Scripting VMware Infrastructure*** lab at VMworld® 2008. This lab will demonstrate some of the capabilities of both the VI Toolkit (for Windows) and the VI Perl Toolkit. These toolkits are programming interfaces that allow anyone to write programs that can monitor and manage any aspect of VMware Infrastructure. The toolkits allow programs to connect to a Web service interface from either VirtualCenter or a VMware ESX server.

This hands-on lab consists of two manuals, one for participants who are programming in Windows PowerShell and another for participants who are programming in Perl. *Please take a moment to verify that you are using the appropriate manual.* The lecture portion of the hands-on lab will be the same for both toolkits.

This lab will include a short presentation as well as hands-on experience writing programs for administration of VMware Infrastructure. A programming background is helpful but not required; the lab will stress VMware Infrastructure scripting concepts and architecture more than programming techniques. The lab will attempt to minimize the need to understand language-specific issues. The background sections of this handbook will concentrate on the theory and data structures involved in SDK programming, and the hands-on programming exercises will focus on the specific implementation.

As part of this lab, you will write or modify Windows PowerShell or Perl routines to demonstrate key SDK functionality. These hands-on exercises will include an examination of methods to gather virtual machine configuration information and performance history and to reconfigure a virtual machine. Experience with a programming language and familiarity with virtual machines, VMware ESX server, and VirtualCenter are suggested but not required.

Because of the lab's security configuration, you cannot create virtual machines in the root folder. In this lab, you must create virtual machines in folders and resource pools that have been configured for you. Your folder name is `studentN` where `N` is your student ID; your two resource pools are `studentN-esx1` and `studentN-esx2`.

Key Concepts

There are a number of important key concepts for using the VI Toolkits. The sections below outline some of the key areas to understand.

Interfacing with VMware Infrastructure

The VMware Infrastructure API (VI API) is the most commonly used API and is for developers who want to create client applications that can manage, monitor, and maintain VMware Infrastructure (made up of VMware ESX servers, VirtualCenter instances, and VMware Server systems that are managed by VirtualCenter). The VMware Infrastructure SDK (VI SDK) is based on the VI API, as are the VI Toolkits. The SDK and toolkits simplify access to the VI API.

The VI API works with any language that supports Web services. The API includes pre-built Web Services Description Language (WSDL) stubs that simplify the process of using the API with Java (using Apache Axis) and C# (using Microsoft .NET). The pre-built stubs, libraries, and development tools for Java and C# form the VI SDK. The VI SDK essentially exposes the VI API Web service directly to the appropriate language; all the objects and the API and SDK methods are the same and little abstraction exists.

In addition to the VI SDK, Perl and Windows PowerShell VI Toolkits allow interfacing to the VI API. The toolkits allow the same capabilities as the VI SDK but have a higher degree of abstraction. The level of abstraction in the toolkits generally makes actions on the VI API easier to perform than does the VI SDK. However, this abstraction might hide some of the fine-grained control that the VI SDK provides.

Underlying API	Specific Method	Development Language
VI API	VI API Web Service (direct access to XML)	Direct XML over HTTP or through a Web services development environment
	VI SDK	Java
		C#
	VI Toolkits	VI Perl Toolkit
		VI Toolkit (for Windows)

In addition to the VI API, VMware provides several other interfaces for working with the VMware virtual infrastructure. These interfaces represent different APIs, each intended for different developer uses.

The following table clarifies some of the differences between the VI API (including the VI SDK and VI Toolkits) and the other VMware APIs.

API Category	API Name	Intended Use
Management	VI API (used by VI SDK and VI Toolkits)	Provides VMware Infrastructure management and monitoring of virtual machines, hosts, VMware ESX configuration, and other resources managed by VMware ESX and VirtualCenter
	CIM SMI-S	Uses the Common Information Model (CIM) standard to report the organizational structure of virtual machines and associated storage
	CIM SMASH	Uses the CIM standard to monitor and control the underlying hardware running VMware ESX
Virtual Disk Access	VDDK	Can access virtual disks remotely, perform actions on VMware Virtual Machine File System (VMFS), and move virtual disk files on and off the VMware VMFS
Virtual Machine Automation	VIX API	Automates virtual machine operations on VMware Server and VMware Workstation
Guest	Guest SDK	Enables read-only access to VMware Infrastructure 3 state and performance data from inside a guest operating system
Legacy	VmPerl	Deprecated interfaces that should not be used in new programming projects; do not confuse the old VmPerl with the newer VI Perl Toolkit
	VmCOM	

Versions of the VI API

VI API version numbers are derived from the version number for the corresponding VirtualCenter release, not the version number of the VMware ESX release. So VI API 2.0 covers all the features in VirtualCenter 2.0, regardless of VMware ESX version, and VI API 2.5 covers the features in VirtualCenter 2.5.

The version numbers of the toolkits are independent of the VI API version. The VI Perl Toolkit version 1.0 uses VI API 2.0, and the VI Perl Toolkit 1.5 and 1.6 use VI API 2.5. The VI Toolkit (for Windows) 1.0 uses VI API 2.5.

Connecting to VirtualCenter and VMware ESX

The VI API operates as a Web service. The toolkit clients connect over HTTP or HTTPS and send requests, in the form of XML documents, to the server. The server responds with the results, also in the form of XML documents. The toolkits hide this data transfer from the programmer and represent the objects and associated XML as local Perl or Windows PowerShell objects.

Connecting to the Web service requires a login and authorization. You can use any user accounts with the toolkits to access VMware Infrastructure. When connected through the toolkits or VI API, users have the same permissions that they would have when using the VMware Infrastructure Client.

Users can connect to either VirtualCenter Server or a VMware ESX server. The structure of the scripts and commands is the same with either server. However, VirtualCenter provides several functionalities that are not available in VMware ESX, such as VMware VMotion and VMware Distributed Resource Scheduler (VMware DRS). In addition, the inventory structure is a bit different when connecting through VirtualCenter because VirtualCenter has more layers of abstraction (data centers, clusters, and folders). Because of this, it is best to always operate through Virtual Center when it is available in an environment.

For a complete list of the API operations supported on ESX server and VirtualCenter Server, refer to the SDK v2.5 Programming Guide, Appendix B (http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/visdk25programmingguide.pdf).

Objects in the VMware Infrastructure versus objects in a script

Two types of objects exist in the VI SDK: *managed objects* and *data objects*. A managed object represents either a physical item in the VMware Infrastructure inventory (such as a virtual machine or a host system), an inventory grouping (such as a folder or datacenter), or a service involving inventory items(such as the login session management system). In the Windows PowerShell version of the toolkit, Managed Objects are sometimes referred to as *server-side objects*.

Because managed objects represent objects that make up the structure of VMware Infrastructure, they exist only on the server, not on the client machine that is running a VMware Infrastructure script. VMware Infrastructure scripts can interact only indirectly with managed objects, by using representations of those objects (called *managed object references*) and objects in the script (called *data objects*). Managed object references essentially specify the ID of the managed object. Data objects are used by VI scripts and represent parts of a managed object and appropriate methods to interact with managed objects on the server. In the Windows PowerShell version of the toolkit, managed object references are sometimes also referred to as *server-side object names.* When you are using the Perl version of the toolkit, think of managed object references as names because they will display like names when printed. Keep in mind that the managed object reference name of an object is not the same as the name that a user gives the object when creating it; managed object references are more similar to ID numbers.

A data object represents information about a managed object. Any object in the VI API that does not represent a manageable object is a data object. Data objects are typically composed of other data objects, references to managed objects, faults, or other basic data types. Information about the proper variable names, properties, and methods needed to extract the data from VMware Infrastructure are described in detail in the VI SDK documentation.

When using a VI Toolkit, the distinction between managed objects and data objects is informal. In contrast, this distinction is very important when using the VI SDK. However, you must understand the distinction between these types of objects when using the toolkits.

In the VI Perl Toolkit, the objects that are created during each call are data objects, and the VI methods in the toolkit act upon managed objects that the code specifies as managed object references. Data objects are passed as arguments to those methods or returned as output from the methods.

In the VI Toolkit (for Windows), the information that is passed through pipes from a `get-*` method typically represents a managed object reference to a specific object or a list of managed object references describing a group of objects. Managed objects are never manipulated directly.

Advanced: Using managed object references

Because you cannot directly manipulate the managed objects (which represent actual items in the VMware Infrastructure) in a script, any time you want to affect one of those objects, you must get the object's reference ID and use that ID as input to the function that you are using to manipulate the object.

You can obtain managed object references in several ways:

• You can construct the reference (this way works only for the special object that represents your connection to the Web service).

• The reference can be the result of a function or cmdlet that returns a managed object reference.

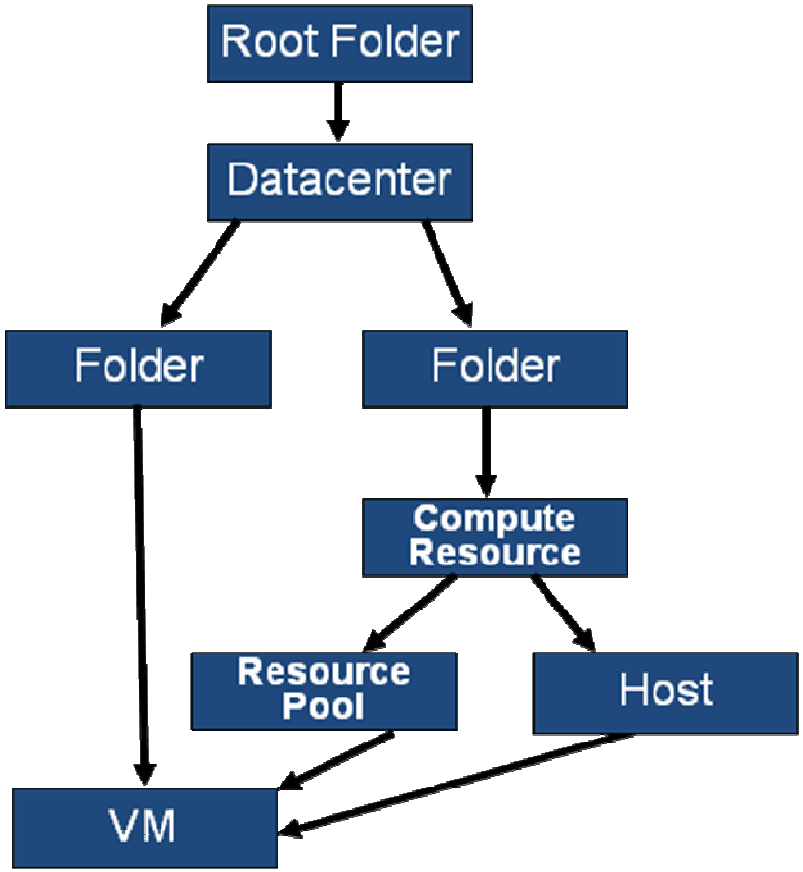
• You can obtain the reference through an accessor method (if a data object type contains a managed object reference as one of its properties).

• You can find the reference by searching for managed objects (typical for managed entities, such as virtual machines and hosts, that are in the inventory tree) or for managed object references that are properties of another object. For the VI Toolkit (for Windows), this way represents a `get-*` cmdlet; for the VI Perl Toolkit, this way represents a call to `find_entity_views` or another method that returns managed object references.

The object hierarchy

Objects in VC are organized as a hierarchy. The illustration below shows a small example of part of the VC hierarchy. The two forks correspond to the two views of VMs in the VI client ("Hosts and Clusters" and "Virtual Machines and Templates").

The diagram shows how objects, such as Datacenters, Hosts, Resource Pools and VMs arecategoried. Some objects, like folders and resource pools, can also be nested. For some of the more advanced exercises, you will need to understand the VI Inventory structure, but there are many sets of tools available for searching.



Using the documentation

When using the toolkits, you should be aware of a few important sources of information. The following list describes the most useful toolkit documentation, as well as general SDK documentation that might be helpful in certain exercises. All the referenced documents are in the `Documentation` directory of the lab clients.

For the VI Toolkit (for Windows), the following guides are most useful for this lab:

- The *Administrator's Guide* covers the basics of how to use the VI Toolkit (for Windows).
- The *VI Toolkit Cmdlets Reference* covers the available cmdlets and how to call them. The *VI Toolkit Cmdlets Reference* is available only as part of the VI Toolkit (for Windows) download.

For the VI Perl Toolkit, the following guides are the most useful:

- The *VI Perl Toolkit Programming Guide* covers how to use the VI Perl Toolkit to write scripts.
- The *VI Perl Toolkit Utilities Reference* is located online http://www.vmware.com/support/developer/viperltoolkit/viperl16/doc/perl_toolkit_utilities_idx.html. and is also referenced in the Toolkit and SDK Docs directory of your VM. This reference provides a list of the utility programs that are included with the VI Perl Toolkit. The utilities are written in Perl and serve not only as useful utilities, but also as great templates for understanding how to perform specific actions using the VI Perl Toolkit.

Regardless which toolkit you are using, the following references are important:

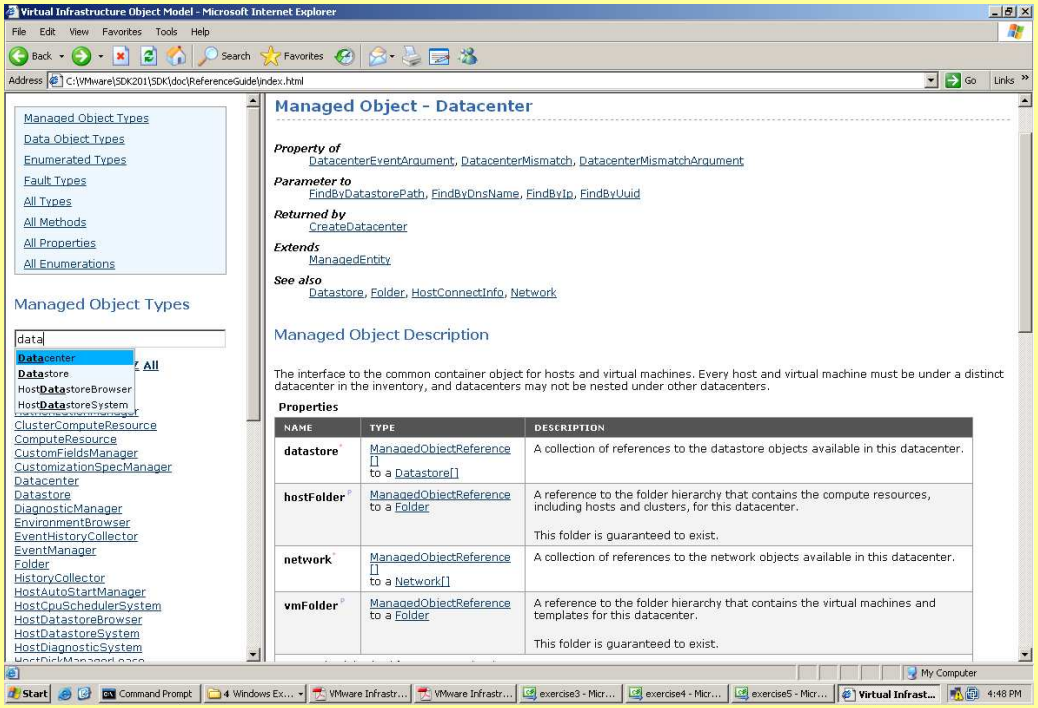
- The *VI SDK Programming Guide* covers the details of programming with the VI SDK. Although you do not need to understand the details of SDK programming to use the VI Toolkits, the appendices of this guide contain important information not found elsewhere.
    - Appendix A of the *VI SDK Programming Guide* describes the performance counters that are available for each object in VMware Infrastructure, as well as which sorts of statistics and rollups are provided for each counter. If you plan to gather performance information from VMware Infrastructure, you will need the information in this appendix.
    - Appendix B of the *VI SDK Programming Guide* describes the privileges required to perform any type of action on a particular VMware Infrastructure object. This appendix also describes which actions require VirtualCenter and which will work on VMware ESX. Although the methods used are specific to the VI SDK and do not always map easily to the VI Toolkits, this is the best reference for trying to understand which privilege is needed for different sorts of tasks.

- The *VI SDK Reference Guide* covers the detailed descriptions of every object available in VMware Infrastructure. The *VI SDK Reference Guide* is very complex (more than 1,300 HTML pages, with most pages including many methods and properties), and the amount of information available in the guide is huge. While nearly everything in Windows PowerShell can be done without referring to the guide, a few operations in Windows PowerShell and many operations in Perl require a deeper understanding of how the managed objects and their properties and methods work. If you plan on going “off the beaten path” in your toolkit usage, you will likely find the *VI SDK Reference Guide* to be a handy tool.

Advanced: Understanding the *VI SDK Reference Guide*

The *VI SDK Reference Guide* (version 2.5) is available online at http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/index.html and in the *Documentation* directory on the client machine for this lab. The other VI SDK documentation covers using the Web services and the theory of the VI SDK. The *VI SDK Reference Guide* is a hyperlinked, highly interactive guide that covers the various managed objects, data objects, and enumerations used in the VI SDK. The easiest way to use the guide is to select which sort of VI SDK feature you are looking for (typically a managed object, data object, enumeration, or fault type) from the links at the left, and then type the object name in the search field. Because the search field is auto-completing, you can use it to find objects whose names you do not know by searching for key terms.

The following figure shows the *VI SDK Reference Guide* in action:



. .

Overview of the VI Toolkit (for Windows)

Introduction

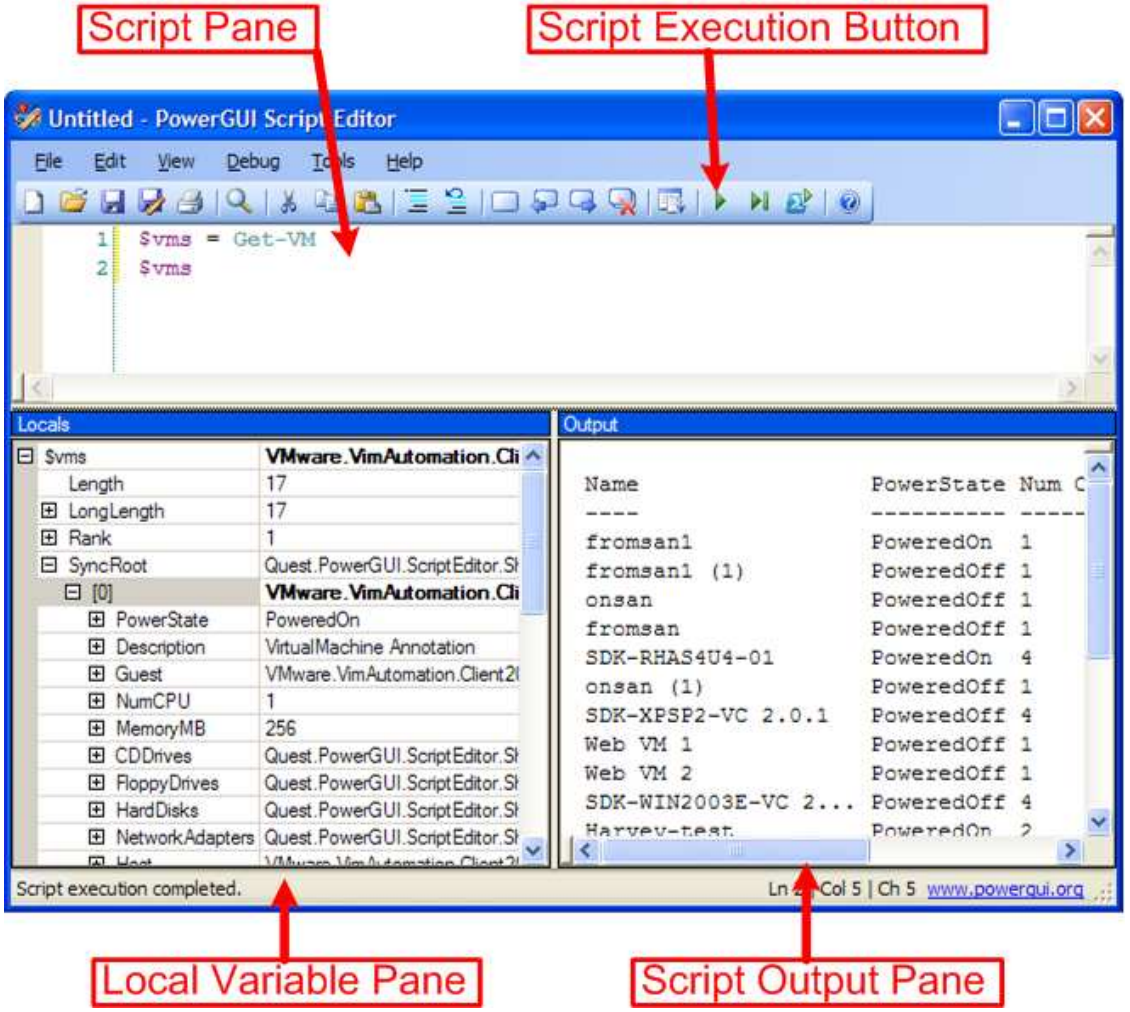
If you are new to Windows PowerShell, or if you have not used it in a while, read this section to get a few pointers before you begin.

Using the PowerGUI Editor

In this lab, you have two choices for executing scripts. Either choice is available as a shortcut on your desktop.

The first option is to execute scripts directly from the VI Toolkit (for Windows). This option provides an environment similar to the default Windows PowerShell environment.

The second approach is to use the PowerGUI script editor. This environment allows you to write and debug scripts easily, as well as to inspect the objects that you receive as output. If you elect to use PowerGUI, the following figure shows what your environment will look like.



To use the editor

In the script pane, enter your script and click the **script execution button**. After the script executes, you will see the output in the script output pane.

The PowerGUI environment provides argument completion, syntax highlighting, debugging and other features, and is a good choice. Use whichever environment is more comfortable for you.

A quick overview of Windows PowerShell

A detailed exposition of Windows PowerShell is outside the scope of this lab. However, this documentation will cover a few high points about what Windows PowerShell is, why it is important, and how to manage VMware by using Windows PowerShell.

Windows PowerShell is an interactive command-line interpreter, much like the familiar DOS prompt or the shell environment in UNIX.

What sets Windows PowerShell apart from other command-line environments is that applications are being built with Windows PowerShell manageability expressly in mind. In the Windows world, Windows PowerShell will become increasingly important for managing many servers and applications.

The rest of this section covers cmdlets that are especially useful for this lab, and how you can learn more about them.

Getting help for a cmdlet

You can get help for a cmdlet by running a script that contains `help <cmdlet name>` from PowerGUI. The help for this cmdlet will be available in the script output pane. You can get more detailed help for a cmdlet by typing **`help <cmdlet name> -full`**.

Helpful cmdlets for this lab

This section discusses the cmdlets that you are most likely to see in this lab.

- `Get-VIServer`: Before you can manage VMware from Windows PowerShell, you must first connect to a VirtualCenter server or VMware ESX host.

  Example:

  ```
  Get-VIServer 192.168.1.1
  ```

  This example connects to the VMware Infrastructure server at 192.168.1.1. To complete the connection, you will be asked for a user name and password.

- `ForEach-Object`: This cmdlet executes a script block for each input object. Input objects usually are piped in to this cmdlet, which is often abbreviated as either **`foreach`** or **`%`**.

  Example:

  ```
  dir | ForEach-Object { $_.name }
  ```

  This example prints the name of each file (including directories) in the current directory.

- `Get-Member`: This cmdlet allows you to inspect an object to determine which methods and properties the method defines. The Get-Member cmdlet is often abbreviated as **`gm`**.

  Example:

  ```
  dir | gm
  ```

  This example shows which methods and properties are defined for file or directory objects.

- `Get-VM`: This VMware cmdlet retrieves managed virtual machines.

  Example:

  ```
  Get-VM | select name, memorymb
  ```

  This example: displays all virtual machines in the virtual infrastructure, along with the amount of memory allotted to the virtual machine.

- `Get-VMHost`: This VMware cmdlet retrieves virtual machine hosts.

  Example:

  ```
  Get-VMHost | select name
  ```

  This example creates a custom report about all virtual machine hosts that are available in the virtual infrastructure.

- `Measure-Object`: This cmdlet is used to count the number of objects in a list.

  Example:

  ```
  dir | Measure-Object
  ```

  This example counts the number of files (including directories) in the current directory.

- `Select-Object`: Use this cmdlet to select certain properties of input objects.

  Select-Object also provides a means of executing a script block to populate a column. This very powerful feature allows you to execute code that uses the object as input and displays the results alongside properties from the object. You will use this feature extensively in this lab.

  Example:

  ```
  dir | Select-Object mode
  ```

  This example selects only the mode of the files (including directories) in the current directory.

  Example:

  ```
  get-wmiobject win32_service | `
    select Name,
           @{Name=”State”;
             Expression= {
            ($_.InterrogateService()).ReturnValue
             }
           }
  ```

  This example shows the advanced querying capability provided by the cmdlet. The example displays a compact report of services along with their state. The cmdlet obtains the state by calling the service’s InterrogateService method and using the returned ReturnValue property.

  You will use this capability extensively in this lab, so become comfortable with it if you plan to perform the intermediate or advanced exercises.

- `Where-Object`: This cmdlet executes a script block against each object. If the script block returns true, the cmdlet emits the input object as an output object. The Where-Object cmdlet usually is used in a pipeline and is often abbreviated as either **`where`** or **`?`**.

  Example:

  ```
  dir | Where-Object { $_.mode –eq “d----“ }
  ```

  This example returns all subdirectories of the current directory.

Pipelining

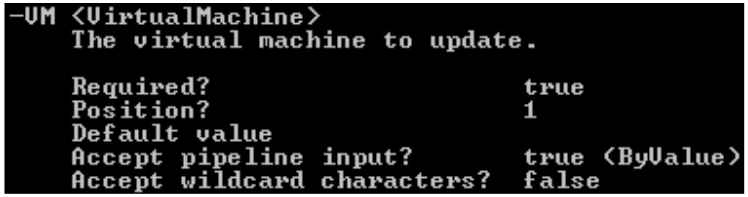
One of the most powerful features of Windows PowerShell is the ability to pipe objects between cmdlets. Doing so causes the output objects of one cmdlet to be used as the input objects of another cmdlet. This lab makes extensive use of pipelining.

Example:

```
get-vm | new-snapshot
```

This example takes a snapshot of all virtual machines in your virtual infrastructure.

To determine which arguments can be piped into a cmdlet, run **`help <cmdlet name> -full`**. The help will include a description of all parameters. For example, the following figure shows the help for the `Set-VM` cmdlet



You can see that the `–VM` argument can be passed in as pipeline input. This means that a command such as

```
Get-VM <my VM> | Set-VM –Name “New Name”
```

is valid, and would rename the virtual machine “New Name.”

Hands-on Lab Overview

The hands-on lab portion of this class consists of several exercises. These exercises are divided into primary and optional exercises. The primary exercises teach the key concepts necessary to perform useful work by using the VI Toolkit (for Windows) or the VI Perl Toolkit. The optional exercises are more advanced and combine the concepts from the primary exercises in novel ways or apply those concepts to new VMware Infrastructure objects and features.

Because of the huge variety and scope of the VI API, this lab focuses on teaching the core concepts, while providing the opportunity to learn some variations and advanced uses of concepts.

Most of the primary exercises include some variations that are not part of the main exercise. You might have difficulty finishing the lab in the allotted time if you try all the variations; consider skipping them unless you have a strong interest in a specific topic. You will be able to try these variations later, outside the lab session.

In addition to the primary exercises and variations, some optional exercises provide an opportunity to work with different objects and features. You will have difficulty completing many of these advanced exercises in the time provided, so concentrate on the one or two that are most interesting to you.

Hands-on Lab Exercises

The hands-on exercises in this lab provide an opportunity to directly interact with the VI toolkits.

This lab setup uses Powershell environment variables to represent various names and parts of names in the environment. You will need to set these environment variables in order to run the lab as specified. In the VI Toolkit (for Windows) toolkit, type the following environment variables, as written, each of which corresponds to object names in your own environment.

What to type in the VI Toolkit (for Windows) command prompt	Meaning
$labVC = "<VC>"	<VC> is the name of the VC server you'll be using
$labVM = "<VM>"	<VM> is the name of a VM you have read/write access to. We used a VM named Student01_VM1.
$labID="<ID>"	<ID> is an ID number used to derive the name of some ESX servers and datastores. Because your environment is different than what was used in the lab, you'll probably need to modify the exercises that use is (exercise 4a and exercise 6). If you have an ESX server named vmw9-1-esx01.vmworld.com, and folders and resource pools setup as in the appendix, the lab will work fine.
$labLogin = "<login>"	<login> is the login you'll use for the lab. You can use any login setup with at least the permissions specified in the appendix. We used l09-student01 for the lab.
$labWinVM = "<ro-vm>"	<ro-vm> is the name of a VM that you have at least read-only access to. You can use the same VM as $labVM. In the lab, we use vmw9-1-winsrv01.

You should verify the environment variables before using them by entering the name of the variable by itself on a line.

Hands-on Exercise 1: Using the VI Toolkit (for Windows) and connecting to the VI API

Exercise 1 teaches you how to list the available cmdlets, set up a simple Windows PowerShell template for use with the VI Toolkit (for Windows), and connect with the VI API.

1a. Listing cmdlets

To read about all the cmdlets that the VI Toolkit (for Windows) provides, you can launch the *VI Toolkit Cmdlets Reference* from the desktop in the `Toolkit and SDK Docs >VI Toolkit (for Windows)` folder. (On a typical VI Toolkit installation, you could find this reference by choosing **Start > All Programs > VMware > VMware VI Toolkit > VI Toolkit Cmdlets Reference**.)

1b. Creating a script template

In the bulk of the exercises that you will perform in this lab, you will interact with the virtual infrastructure by invoking cmdlets on the command line.

To try launching cmdlets from the command line

1. Launch **VMware VI Toolkit (for Windows)** from the desktop in the `Perl and Powershell Toolkits` folder. (On a typical VI Toolkit installation, you would find this file by selecting **Start > All Programs > VMware > VMware VI Toolkit > VMware VI Toolkit (for Windows)**.
2. At the prompt, type `Connect-VIServer *$labVC*`. Because you created an environment variable called $labVC that represents the name of the VC server you'll connect to, this connects to the VI server.
3. Type **`Get-VM`** and press **Enter**.
4. Type **`Disconnect-VIServer`** and press **Enter**.

With the few commands above, you have connected to a VirtualCenter server, authenticated to the server, requested a list of all the VMware ESX servers that are registered to that VirtualCenter server, and gracefully disconnected from the VirtualCenter server.

Although invoking cmdlets on the command line is an easy way to work with the VI Toolkit (for Windows), if you want to schedule Windows PowerShell cmdlets to run periodically, you will need to create scripts that invoke them. Scripts are text files that contain the cmdlet(s) that you want executed.

The purpose of this lab is to create a standard scripting template that you can use in creating VI Toolkit Windows PowerShell scripts. This lab will illustrate how you could create scripts by using a template, but you will *not* use that template in the other exercises in this lab. You might find the template useful for reusing the scripts in this lab after you are back in the office.

Scenario

You are the administrator of the VMware Virtual Infrastructure at your company. You need to write some scripts to automate a process. The first step is to create a script template that will quickly prototype and deploy your VI Toolkit (for Windows) scripts.

Tasks

1. Import the VMware VI Toolkit snapin.
2. Connect to a server and authenticate.
3. Disconnect from the server.

Step-by-step instructions

1. Use notepad.exe to open a new file.
2. Enter the following information into the file:

`# Template for PowerShell Toolkit scripts`	Lines beginning with # are comments in Windows PowerShell.
`# Add the VMware VI Toolkit functionality` `Add-PSSnapin VMware.VimAutomation.Core`	Add the VI Toolkit functionality to your Windows PowerShell environment for this script.
`Connect-VIServer _____________________`	The `Connect-VIServer` cmdlet will connect to the virtual infrastructure so that you can communicate with it.
`# Add your code here`	You will add your code here.
`Disconnect-VIServer _____________________`	Close your connection to the virtual infrastructure when your script complete.

Advanced (optional): Session Files

In your template, you use the `Connect-VIServer` cmdlet to connect and authenticate to the VirtualCenter server.

The VI Toolkit (for Windows) provides an option to save authentication information to a session file. This session file encodes the user name and password information in a token. This method avoids storing this information in plain text. By default, this token expires after 30 minutes. After authenticating once to generate the session file, you can run cmdlets from the command line or from scripts, without needing to provide the user name and password again.

You will not use session files in this lab. Rather, you will continue to use the `Connect-VIServer` cmdlet.

Advanced (optional): Additional Cmdlets

In your template, you enable scripts to operate on the virtual infrastructure by invoking the `Add-PSSnapin` cmdlet. Although you can use this cmdlet to access the bulk of the VI Toolkit (for Windows) functionality, you cannot use it to invoke the following:

- `Get-VIServer`
- `Get-VC`
- `Get-ESX`
- `Get-VICommand`
- `New-DatastoreDrive`
- `New-VIInventoryDrive`

By adding a small amount of code to your template, you can enable your scripts to invoke these six aliases and functions. The necessary code is in the following file:

```
C:\Program Files\VMware\Infrastructure\VIToolkitForWindows\Scripts\Initialize-VIToolkitEnviront.ps1
```

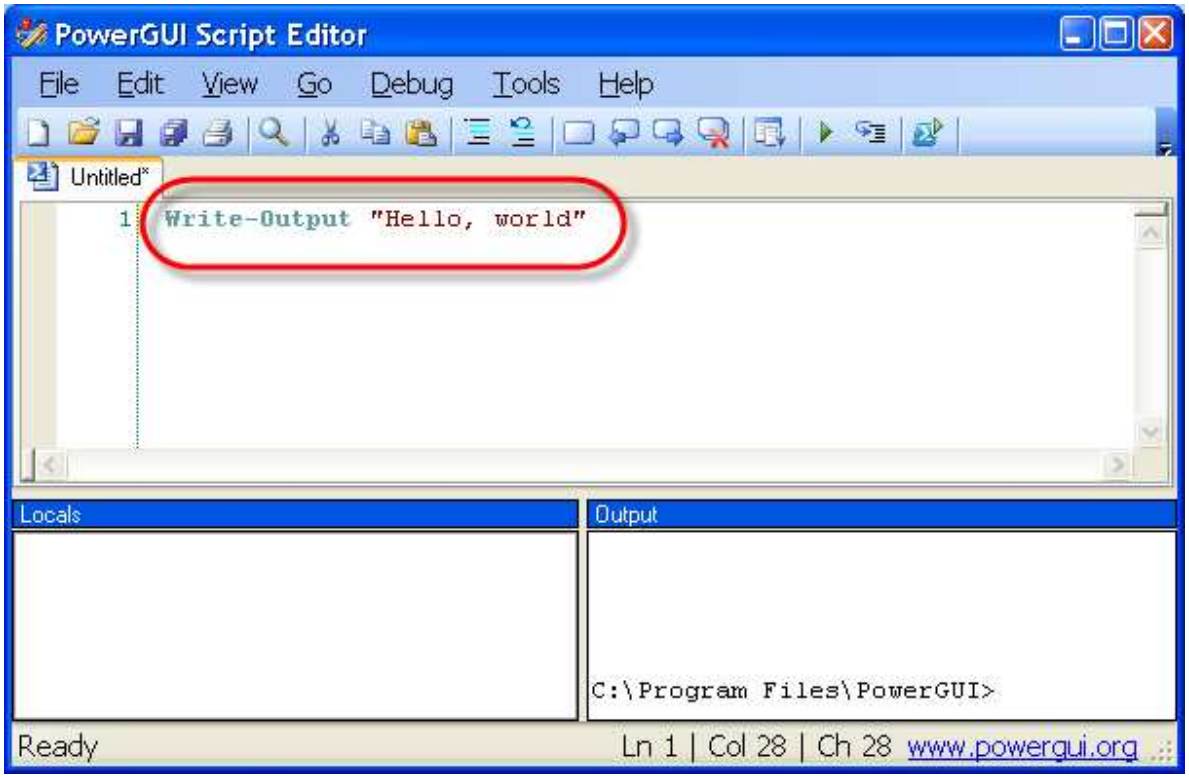
You can simply paste code from this file into the Windows PowerShell template. Paste the code beneath the following line:

```
Add-PSSnapin VMware.VimAutomation.Core
```

So far you have seen two ways to command the VI Toolkit (for Windows): the command-line interface and a script. A third approach is to use a script editor such as PowerGUI.

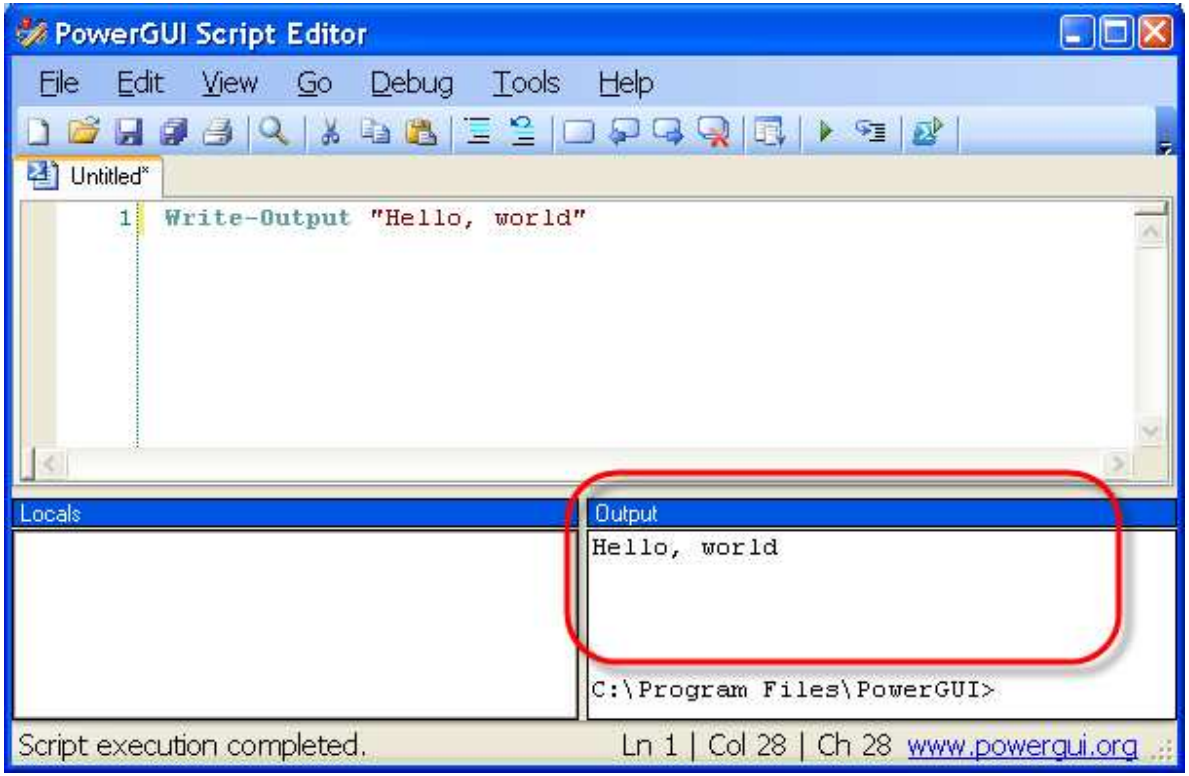
Step-by-step Instructions

1. Launch **PowerGUI** from the desktop (in the `Perl and Powershell Toolkits` folder). On a typical VI Toolkit installation, you would find this file by selecting **Start > All Programs > PowerGUI > PowerGUI Script Editor**.
2. Type **`Write-Output "Hello, world"`** as shown in the following figure:



Notice that the PowerGUI script editor assists you by offering to auto-complete the cmdlets that you type and by color-coding the script as you write it.

3. Click the **Play button**, which looks like this: 
   You can see the output of this simple program in the PowerGUI script editor's Output pane:



1c. Launching scripts

NOTE: This section is for informational purposes only. The lab machines have already been configured using this procedure. Because you do not have administrative rights to the environment, you will not be able to run this example. However, this is the procedure you would need to run in order to execute Powershell scripts in a different environment.

In this lab environment, you are creating and launching Windows PowerShell scripts from within PowerGUI (see http://www.powergui.org). Doing so makes invoking Windows PowerShell scripts easy. The purpose of this lab is to show you how to execute Windows PowerShell scripts without using PowerGUI. Specifically, you will configure security to allow Windows PowerShell scripts to execute and you will learn how to invoke Windows PowerShell scripts from powershell.exe.

Scenario

You are the administrator of VMware Infrastructure at your company. You need to execute some Windows PowerShell scripts from powershell.exe and to configure security to allow the scripts to execute.

Tasks

1. Use the `Get-ExecutionPolicy` and `Set-ExecutionPolicy` cmdlets.
2. Invoke powershell.exe (the base Windows PowerShell application, without the VI Toolkit components).
3. Use the *dot slash* method to invoke and execute a Windows PowerShell script.

Step-by-step instructions

1. Launch **Windows Powershell** from the desktop (in the `Perl and Powershell Toolkits` folder). On a typical VI Toolkit (for Windows) installation, you would find this file by selecting **Start > All Programs > Windows PowerShell 1.0 > Windows PowerShell**.
2. Run the following command to determine Windows PowerShell's execution policy:

   ```
   Get-ExecutionPolicy
   ```

   This command will respond with one of the following:

Execution Policy	Explanation

Restricted	No scripts are allowed to run.
AllSigned	Only scripts that have been signed by a trusted third party are allowed to run.
RemoteSigned	All local scripts are allowed to run, but scripts downloaded from the Internet (e.g., received by email, downloaded from Web sites) are required to have a signature from a trusted third party.
Unrestricted	All scripts are allowed to run.

You will use the `Set-ExecutionPolicy` cmdlet to configure the execution policy. You must run this cmdlet as an administrator.

3. Run the following command to exit Windows PowerShell:

   ```
   exit
   ```

4. Launch **Windows Powershell** from the desktop.
5. Select **Run as**, then select **The following user** option and enter the administrator user name and password. Click **OK.**
6. Run the following command to configure the lab environment to run your scripts:

   ```
   Set-ExecutionPolicy RemoteSigned
   ```

   You no longer need to be logged in as administrator.
7. Type the following command:

   ```
   exit
   ```

8. Launch **Windows Powershell** from the desktop.
9. Type the following:

   ```
   .\yourscript.ps1
   ```

   If you use cmd.exe, you might be accustomed to launching scripts in the current folder simply by typing their name. For security reasons, powershell.exe requires you to type the previous command explicitly. The period (.) character stands for “in the current folder,” so the command means “PowerShell, look in the current folder and run the script called `yourscript.ps1`.”

2a. Finding objects and getting properties

Exercise 2 teaches you how to find server-side objects in VMware Infrastructure after a script has connected, and how to find and list the properties of those objects. The basic concepts of finding server-side objects and listing their properties are applicable to all objects in the VMware Infrastructure, including virtual machines, hosts, virtual switches, port groups, resource pools, data stores, and more. Using these methods, you will be able to locate the configuration or state information for any object. Note that performance information is not considered a basic property of a VMware Infrastructure object, and performance metrics will be covered in Exercise 5.

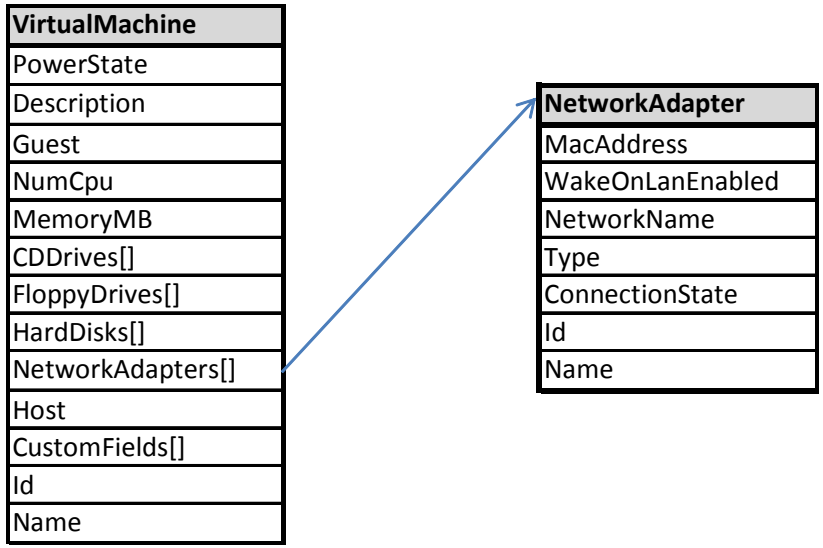
Understanding server-side objects and properties

The goal of this exercise is to gain a basic understanding of the server-side objects that are used for scripting VMware Infrastructure. These objects generally map to the objects that are visible in the Virtual Infrastructure Client (VI Client) GUI. The following are examples of some of the types of server-side objects with which VI Toolkit (for Windows) programs typically interact:

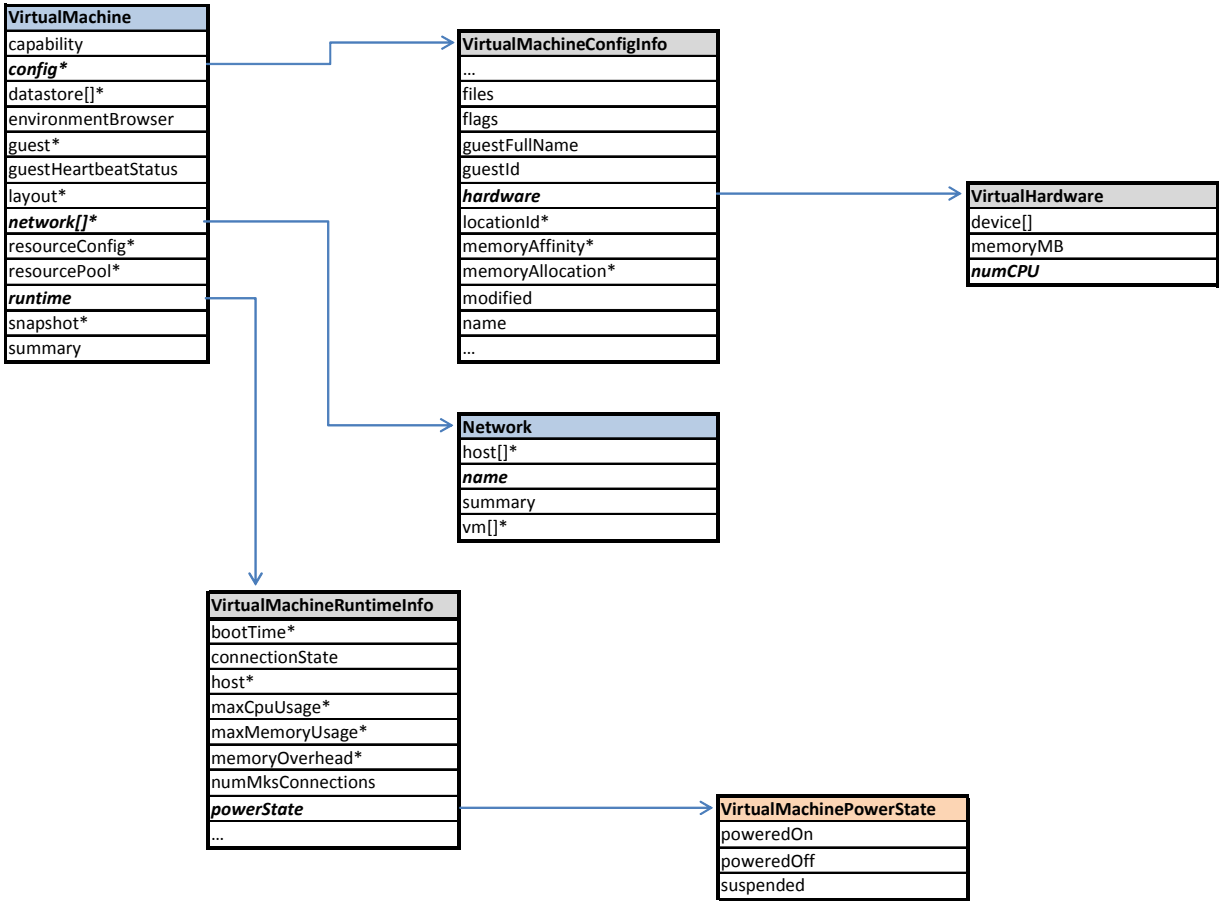
- ComputeResource
- Datacenter
- Datastore
- Folder
- HostSystem
- Network
- ResourcePool
- Task
- VirtualMachine

Each of these server-side objects has several associated properties. These properties represent both the configuration and the state of the object. In addition, these properties can indicate relationships with other objects.

The following figure shows some of the properties belonging to and associated with a virtual machine server-side object that is easily available in Windows PowerShell. Note that any field name with square brackets following it might have multiple instances and must be handled as an array.



The next figure provides a more detailed list of the properties of a virtual machine. This diagram does not include a complete list of properties. You can find a complete list of properties for each object in the *VI SDK Reference Guide* (http://www.vmware.com/support/developer/vc-sdk/visdk25pubs/ReferenceGuide/index.html). You can also use Managed Object Browser (MOB), a VI SDK tool that shows the relationships and properties of server-side objects on a live system.



The first step to interacting with objects in Windows PowerShell scripts is to find the objects. You can do so by using one of the Windows PowerShell `get-*` cmdlets. These cmdlets provide several options for finding objects (e.g., finding the objects by name). When a `get-*` cmdlet is run by itself, it prints information about a matching object. However, the `get-*` command also provides a unique reference ID for the server-side object. Every other command uses this reference ID to refer to the object. The ID is more than just the name of the object—it is an identifier that specifies that object all the time, even if the object name changes. To use the reference ID in other objects, the output from a `get-*` cmdlet can be piped to other commands or put in a variable. Most cmdlets that accept a piped-in reference ID will also allow the option to specify the object with the `-entity` option. Because the actual object (for instance a virtual machine) exists on the server rather than on the machine on which you run the script, must use a `get-*` cmdlet to provide the reference ID of the object each time you want to act on that object.

The second step to interacting with objects is to determine which properties you want to examine. The previous figures show some properties of a VirtualMachine server-side object. As mentioned previously, you can find this information for other objects by using the *VI SDK Reference Guide*. Although you cannot manipulate the server-side object directly, you can use the `get-*` cmdlet to feed the object ID to another cmdlet, to view the properties.

Properties for objects can be complex. Properties holding multiple values are arrays in Windows PowerShell. Also, note that some properties nest within other properties. You might need to use more-complex syntax to read the properties of nested properties or arrays. For example, to refer to a specific property of an object, you can use *dot* syntax. In this syntax, `$VM.NumCpu` represents the number of CPUs (the NumCpu property) of a virtual machine. For properties that represent server-side objects, or for uncommon properties that are not in the default Windows PowerShell views of an object, you might need to use the `Select-object` cmdlet. Running `Select-Object *` on a reference ID will list all of the available properties. Running `Select-object` on a property that is itself a server-side object will return the reference ID for that object. You can use this cmdlet to specify objects by reference ID, even if you accessed them through the properties of another object.

For properties that consist of arrays, you might find the `foreach-object` cmdlet useful. The `foreach-object` cmdlet creates a loop and cycles through every element of an array. The special variable `$_` indicates the currently iterating element of the loop, so using `$_.property` would indicate the property of the current run through the loop.

Scenario

You are the VMware Infrastructure administrator at your company. You need to write a script to validate the compliance of your environment with a VMware Infrastructure best practice and your company's security guidelines.

- Best practice dictates that virtual machines be built with a single virtual CPU, unless the workload warrants multiple virtual CPUs. You need a list of all virtual machines that have more than one virtual CPU.
- Your corporate security policy dictates that no development virtual machines be attached to the production network. The production network is attached to a VMware Infrastructure port group called Production_LAN, and development virtual machines all have names containing DEV.

Tasks

1. Generate a list of the virtual machines in the environment.
2. Locate a specific virtual machine by name, and list attributes of the virtual machine object.
3. (Variation) Find all virtual machines with more than one virtual CPU.
4. (Variation) Find all development virtual machines on the Production_LAN network.

Abstract

1. Listing all virtual machines involves logging in to the VMware Infrastructure environment and querying the environment for all virtual machine objects.
2. Locating a specified virtual machine requires knowledge of the object's property name and the value of that property. Typically, you will locate a particular virtual machine by name. Logically, this property is called `name`. In order to determine the available properties of an object, use the Windows PowerShell `Select-Object` or `Get-Member` cmdlets.
3. With knowledge of the properties of a virtual machine object, you have many options for locating groups that match specific criteria. Use the `Where-Object` cmdlet to locate objects.

Step-by-step instructions

1. List all virtual machines.

`Connect-VIServer` ***`$labVC`***	Connect to the environment. This step requires authentication and might be called with `-User` and `-Password` parameters.
`Get-VM`	Request a list of all virtual machines known to the VirtualCenter server.

2. Locate a virtual machine by name and list the properties of that virtual machine.

`Get-VM vmw9-1-vm`***`${labID}`***	List the properties of VM named **vmw9-1-vm${labID}**, where **labID** is the environment variable representing yor pod #.
`Get-VM vmw9-1-win*`	The **Get-VM** commandlet can also take a pattern. In this case, list all virtual machines beginning with the string **vmw9-1-win**
`Get-VM` **`vmw9-1-vm`*`${labID}`*** `| Select-Object *`	Request a list of all properties of the **vmw9-1-vm${labID}** and their values
`Get-VM` **`vmw9-1-vm`*`${labID}`*** `| Get-Member`	Request a list of all properties and methods of the **vmw9-1-vm${labID}**object

3. Report on specific properties.
   1. Virtual machines with more than one virtual CPU

   The back-tic (`) character on the first line indicates that the next character should not have a special meaning (an escape character). The example below shows it being used to escape the new line (it needs to be immediately before the return). This allows you to display a single line of Powershell across multiple lines. We will be using this notation frequently throughout the lab to make the code more clear.

`Get-VM | ` `Where-Object {$_.NumCPU -ne 1}`	Use the `Where-Object` cmdlet to filter and list only the virtual machines with more than one virtual CPU.

2. Advanced Variation (optional): Find virtual machines with names beginning with DEV on the **Production_LAN** network

`Get-VM *DEV*`	List the virtual machines with names containing DEV.
`Get-VM *DEV* | ` `  ForEach-Object {` `  Get-VirtualPortGroup -vm $_` `  }`	Obtain the virtual machines with names containing DEV and list all port groups that attach to those virtual machines. Note the use of the `ForEach-Object` cmdlet because there might be multiple network interfaces per virtual machine; the `Get-VirtualPortGroup` cmdlet returns an array of interfaces and must be handled appropriately.
`Get-VM *DEV* | ` `  ForEach-Object {` `  $vm=$_;` `  Get-VirtualPortGroup -vm $_ \| ` `  Select-Object @{name="VM Name"; ` `  expression={$vm.name}},Name` `  }`	Modify the output to display the associated virtual machine name in the output table with the port group names. Use `Select-Object` to generate the **VM Name** column. In this context, **Name** refers to the name of the Port Group. The code `$vm=$_;` is used to preserve a reference to the virtual machine and to list its name in the output table.
`Get-VM *DEV* | ` `ForEach-Object {`	List the virtual machines that have DEV in their names that also have a network interface on the Production_LAN port group.

```
    $vm=$_;
    Get-VirtualPortGroup -vm $_ | `
    Where-Object {
      $_.Name -eq "Production_LAN"
    } | `
    Select-Object @{name="VM
  Name";expression={$vm.name}},Name
  }
```

Note: The last two lines of this sample code must be typed on the same line, not broken as in the code to the left.

Hands-on Exercise 3: Using snapshots to reconfigure existing objects

Exercise 3 teaches how to use the VI Toolkit to reconfigure existing virtual infrastructure objects. To maintain a consistent theme for this section, you will use scripts to manage snapshots of virtual machines. You can manage snapshots from the VI Client, by using the Snapshot Manager feature. However, you must invoke the VI Client Snapshot Manager separately for each virtual machine; there is no view that simultaneously shows all snapshots for all virtual machines.

Understanding reconfiguration

Reconfiguration of objects in Windows PowerShell is straightforward. In the most typical case, you can use a `get-*` cmdlet to obtain a server-side object's reference ID. You then can pipe or pass the ID to a reconfiguration cmdlet. Most reconfiguration cmdlets are easily identified by their names, which begin with `move-*`, `new-*`, or `remove-*`. However, other cmdlets also perform reconfigurations. The following is a list of the reconfiguration cmdlets:

- `Add-VMHost`
- `Dismount-tools`
- `Mount_tools`
- `Move-*`
- `New-*`
- `Remove-*`
- `Restart-*`
- `Set-*`
- `Shutdown-VMGuest`
- `Start-*`
- `Stop-*`
- `Suspend-*`
- `Test-VMHostSnmp`
- `Update-Tools`

When you use a reconfiguration cmdlet in Windows PowerShell, it creates a request to VMware Infrastructure to update the relevant object on the server. By default, execution of the script waits until VMware Infrastructure updates the server-side object.

Understanding snapshots

Snapshots represent the state of a virtual machine at a particular point in time. A virtual machine can be reverted to a snapshot, undoing all changes made to its memory and disk since the snapshot was taken. You can take a snapshot of only the disk state; this type of snapshot is generally used when backing up virtual disks. VMware Infrastructure 3 supports multiple snapshots for a virtual machine and allows you to navigate to any snapshot. As each additional snapshot is taken, a tree of snapshots is formed. This process introduces some complexity in writing scripts for snapshots: You must understand how to find the snapshot objects as well as how to manipulate them.

3a. Creating snapshots

The purpose of this exercise is to learn how to create snapshots through Windows PowerShell scripting.

Scenario

You are the administrator of the VMware Infrastructure at your company. You have been asked to create snapshots for the SAP farm programmatically.

Tasks

1. Identify the virtual machines that require a snapshot.
2. Create snapshots for those virtual machines.

Step-by-step instructions

1. List all virtual machines.

`Connect-VIServer $labVC`	Connect to the environment. This step requires authentication and might be called with `-User` and `-Password` parameters.
`Get-VM vmw9-1-vm*`	Request a list of all virtual machines that support SAP, for example.

2. Create snapshots for those machines. When prompted for a name, name the snapshot “my Snapshot”.

`Get-VM` **`$labVM`** `| New-Snapshot`	Create a snapshot for a particular virtual machine.

3b. Identifying virtual machines that have snapshots

The purpose of this exercise is to learn how to find and list virtual machines that have snapshots.

Scenario

You are the administrator of the VMware Infrastructure at your company. You have been asked to identify all the virtual machines that have snapshots.

Tasks

Identify the virtual machines that have snapshots.

Step-by-step instructions

List all virtual machines that have snapshots.

`Connect-VIServer $labVC`	Connect to the environment. This step requires authentication and might be called with `-User` and `-Password` parameters.
`Get-VM | Get-Snapshot`	List all snapshots. This command will list the names of the snapshots, not including the name of the virtual machine.

The `Get-Snapshot` cmdlet (like most cmdlets) returns only a portion of the available information about the object. To see all the available properties, you can pipe the object to the `select-object *` cmdlet, which lists all the available properties. You can use the `select` command to specify a subset of these fields to output.

`get-vm | get-snapshot | select VM, Name, Description, Created`	This command is slightly more advanced and will list all virtual machines that have snapshots, including details about each snapshot.

3c. Deleting virtual machines snapshots

The purpose of this exercise is to learn how to find and delete virtual machine snapshots.

Scenario

You are the administrator of VMware Infrastructure at your company and have been asked to delete snapshots.

Tasks

1. Identify the virtual machines that have snapshots.
2. Delete those snapshots.

Step-by-step instructions

1. Identify and delete all the snapshots.

`Connect-VIServer $labVC`	Connect to the environment. This step requires authentication and might be called with `-User` and `-Password` parameters.
`Get-Vm` **`$labVM`** `| Get-Snapshot | Remove-Snapshot`	This very simple, brute-force set of commands offers you the ability to delete all snapshots. the command lists each snapshot name and asks for confirmation. The output does not clarify to which virtual machine each snapshot belongs. Because you have limited permissions in this lab, please specify the VM to avoid pages full of authorization errors.

or

1. Identify all the snapshots.

`Get-VM $labVM | New-Snapshot`	Recreate the snapshot so you have something to ID. Name the snapshot anything.
`$snaps = Get-Vm` **`$labVM`** `| Get-Snapshot`	This solution is more complicated but more elegant. First, use the `$snaps` variable to collect all the snapshots and save the result.

`Write $snaps | Select VM, Name, ID | Format-Table -autosize`	Print out a table that lists all the snapshots by name and includes details. The `-autosize` command widens the column for the snapshot name so that the names are not cut off.

2. Delete the snapshots.

`Remove-Snapshot $snaps`	This command lists each snapshot and asks for confirmation before deleting them. You can refer back to the table to correlate each snapshot name to a virtual machine.

3d. Deleting virtual machine snapshots older than a certain date

The purpose of this exercise is to learn how to find and delete old snapshots.

Scenario

You are the administrator of VMware Infrastructure at your company. You have been asked to delete older snapshots that are no longer needed.

Tasks

1. Identify snapshots that are older than a certain date.
2. Delete those snapshots.

Step-by-step instructions

1. Identify the old snapshots.

`Connect-VIServer $labVC`	Connect to the environment. This step requires authentication and might be called with `-User` and `-Password` parameters.
`$date = get-date`	Get the current date and store it in the `$date` variable.
`$pastdate = $date.AddMinutes(-30)`	Create a "`$pastdate` variable that specifies the cutoff date. You can use `Date.AddHours`, `Date.AddDays`, `Date.AddMonths`, or `Date.AddYears`. These examples are all Microsoft .NET functions and accept negative values to subtract time.
`$snaps = get-vm` **`$labVM`** `| get-snapshot | ` `` ` `` `where { $_.Created –lt $pastdate }`	Collect all the snapshots that are older than `$pastdate`. Use the `Where-Object` cmdlet to filter.
`write $snaps | select VM, Name, ID, Created`	Print out a table that lists all the snapshots by name and

	includes details.

2. Delete the old snapshots

`remove-snapshot $snaps`	This command lists each snapshot and asks for confirmation before deleting them. You can refer back to the table to correlate each snapshot name to a virtual machine and creation date.

Hands-on Exercise 4: Logs and events

Exercise 4 is divided into two sections: logs and events.

4a: Working with logs

The goal of this exercise is to gain a basic understanding of the objects that are used for monitoring virtual infrastructures. This exercise consists of two parts: VMware ESX logs and VirtualCenter events. You can use the `Get-Log` cmdlet to access VMware ESX logs; you can use the `Get-VIEvent` cmdlet to access VirtualCenter events .

Understanding VMware ESX logs

VMware ESX logs are stored as text files on the host but are also available through a server-side object in the VI API. The VI SDK provides several server-side objects that provide access to specific types of data (such as events, tasks, alarms, and logs). Windows PowerShell uses the `Get-log` cmdlet to get information from logs and the `get-vievent` cmdlet to get information from the VirtualCenter event list.

Scenario

You are the VMware Infrastructure administrator at your company and you need to perform an audit of your environment. Rather than visit each VMware ESX host, you would like to write a script to extract data from the log files. Best practice is to restrict SSH access to the service console and disallow root SSH connections. Your goal is to discover which users have been making SSH connections to the VMware ESX hosts in the environment.

Tasks

1. Determine which VMware ESX logs are available through Windows PowerShell. The information that you need regarding SSH connections is contained in the `messages` log on the VMware ESX host.

2. Retrieve the specified log file or retrieve a subsection of the specified log file (range of lines).
3. Perform a search through the specified log file for a given pattern and return only the matching lines.
4. Save the results to an external file.

Step-by-step instructions

1. To determine which logs are available, use the `Get-LogType` cmdlet.

`Get-LogType -host` **`vmw9-1-esx${labID}.vmworld.com`**	This is a VMware ESX operation, so you must specify a target host to the `Get-LogType` cmdlet.

The output should look similar to the following:

```
Key                  Summary
---                  -------
hostd                server log in 'plain' format
messages             server log in 'plain' format
vmkernel             server log in 'plain' format
vmksummary           server log in 'plain' format
vmkwarning           server log in 'plain' format
vpxa                 VirtualCenter agent log in 'plain' format
```

2. To retrieve a specific log file, use the `Get-Log` cmdlet and specify the desired log file through the `-Key` switch. The log data is contained in the `Entries` property.

`$messages = Get-Log ` `` ` `` `-Host vmw9-1-esx${labID}.vmworld.com ` `` ` `` `–Key messages`	This command retrieves the entire `messages` log from the `esxXX.vmworld.com` host and puts it into the `$messages` variable.
`$messages.Entries`	This command prints the lines from the `messages` log to the console. You can pipe the command through `more` for page-at-a-time viewing.

To limit the number of returned lines, use the Windows PowerShell sub-array syntax.

`$messages.Entries[-10..-1]`	This command returns the 10 most recent lines from the `messages` log.

3. Filtering the log involves the `Where-Object` cmdlet's `-match` operator.

`$messages.Entries \|` `Where-Object {$_ -match "sshd"}`	Match and print all lines containing `sshd` in the `messages` log.

4. You can save the output to an external file in several ways. In this example, use the `Out-File` cmdlet.

`$messages.Entries \| ` `Where-Object {$_ -match "sshd"} |` `Out-File C:\sshlogs.txt`	This command matches all lines containing `sshd` in the `messages` log and writes the output to `C:\sshlogs.txt` in the current working directory on the local machine.

4b. Working with VirtualCenter Events

Scenario

You are the VMware Infrastructure administrator at your company.

- A production virtual machine was powered off during the middle of the day. Your first task is to write a script to report the VirtualCenter power operations that were performed on this virtual machine.
- You have some concerns that someone else may be trying to hack your account. Log-in to see what actions have been performed using your account.

Tasks

1. Retrieve all VirtualCenter events.
2. Retrieve a subset of the VirtualCenter events.

3. Perform a search through the VirtualCenter events for a given pattern and return only the matching lines.
   1. Virtual Machine PowerOn/PowerOff operations
   2. Events tied to your account

Step-by-step instructions

1. When attached to the virtual infrastructure, use the `Get-VIEvent` cmdlet to retrieve events from VirtualCenter.

`Get-VIEvent`	This command, by itself, will return all events from the attached VirtualCenter server. The information returned for each event is dependent on its source (e.g., virtual machine, host, cluster). Regardless of type, all events contain a `fullFormattedMessage` property.

2. You can use standard Windows PowerShell `Select-Object` cmdlet (often abbreviated `select`) criteria to filter the output .

`Get-VIEvent | Select-Object –last 10`	You can pipe output from this command through `Select-Object` to return the most recent (`-last ##`) or oldest (`-first ##`) events.

3. You can use the `Where-Object` cmdlet, often abbreviated `where`, to search through the log for specific items.
   1. The following code detects power cycles. If this code does not yield any result, powercycle a virtual machine.

`Get-VIEvent | ` `` ` `` `Where-Object ` `` ` `` `{$_.fullFormattedMessage ` `` ` `` `-like "*power*"} | ` `` ` `` `select fullFormattedMessage`	This command returns all events that contain the string `power` in the `fullFormattedMessage` property, then returns only that field.

   2. Return all events tied to a given user.

```
Get-VIEvent  | `
where {$_.userName -like "*${labLogin}*"}
```

This command returns all events that were logged with your username

Return all events tied to your account, grouped by event description and count occurrences.

```
Get-VIEvent  | `
where {$_.userName -like "*${labLogin}*"} | `
Group-Object -Property fullformattedmessage | `
select count,name | Format-List
```

This more-advanced script returns the same events as the previous code but uses additional Windows PowerShell functionality to format the output.

Advanced (Optional):

You have many options for parsing events and returning only the information that you want. A few possibilities are contained in this section.

- To list all failed VirtualCenter login events

```
Get-VIEvent  | `
where {$_.fullformattedmessage `
-like "*failed login*"}
```

- To return any events in which the state was transitioned to red

```
Get-VIEvent  | `
where {$_.to -eq "red"}
```

- To count the red events instead of returning the data

```
(Get-VIEvent  | `
where {$_.to -eq "red"} | `
measure-object).count
```

- To return the red events, adding the event's date and time to each record

```
Get-VIEvent  | `
where {$_.to -eq "red"} | `
select -property createdTime,fullFormattedMessage | `
format-list
```

- To get both the transition to and from red events to get a clearer picture of what might be happening

```
Get-VIEvent  | `
where {($_.to -eq "red") -or ($_.from -eq "red")} | `
select -property createdTime,fullFormattedMessage | `
format-list
```

Hands-on Exercise 5: Getting performance data

The goal of this exercise is to facilitate an understanding of the performance data that is available in VirtualCenter and VMware ESX hosts and how to collect and manipulate this data. Both VirtualCenter and VMware ESX hosts maintain performance data. A VMware ESX server maintains performance data only about itself and the virtual machines and resource pools that exist on that host. VirtualCenter maintains performance data for all its clusters, hosts, resource pools, and virtual machines. Performance data is collected at multiple defined intervals. The information is collected on both the host and the VirtualCenter system for a 20-second sample interval and a 5-minute interval, retained for 1 hour and 1 day, respectively. VirtualCenter maintains historical performance data for the 1-hour sample interval, 4-hour sample interval, and 1-day sample interval, which are retained for 1 week, 1 month, and 1 year, respectively.

Understanding performance counters and metrics

You must understand a few key points about collecting performance information in VMware Infrastructure. The first point is that performance information is not part of the normal properties of an object; rather performance information is accessed through a special server-side object in the VI API. This object is called PerformanceManager.

The second point is the difference between counters and values. A counter indicates a type of collected performance information. Counters represent a description of how some performance measure is recorded. A counter does not provide any information about the actual performance data or the object on which the data was collected. The counter only describes a type of data that could be collected. Counters store the following:

- Type of units (e.g., percent, milliseconds, KB)
- Text description of what the counter measures
- Statistic type (amount of change, point-in-time value, or rate)
- How the information is rolled up over longer periods (average, minimum, maximum, summation over time, of the latest value)
- Minimum VirtualCenter log level at which the statistic is recorded

Counters are divided into groups according to categories (such as cpu, disk, memory), and are usually represented by a dotted string notation, which includes the counter's group, name, and roll-up type. For example, `cpu.usage.min` indicates the minimum CPU usage in a collected sample. A list of all VMware Infrastructure 3 performance counters is in Appendix B of the *VI SDK Programming Guide*.

By contrast, a performance value represents the actual information collected, rather than the description of that information.

5a. Getting performance data

The VI Toolkit (for Windows) includes the `get-stat` cmdlet, which enables the collection of performance data for default counters and allows you to specify the counter in which you are interested.

Scenario

You are the VMware Infrastructure administrator and would like to view performance data about one or more virtual machines.

Tasks

1. Get common performance data for a virtual machine.
2. Get default, resource-specific statistics for a virtual machine.
3. Get default statistics for a specific period.
4. Get default, resource-specific statistics for a specific interval.
5. Specify the counter for which you want performance data.
6. Get an average for CPU Ready for all running virtual machines.
7. Get an average for each default counter for the given virtual machine.

Step-by-step instructions

1. Get common performance data for a virtual machine.

`get-vm ${labWinVM} | get-stat -common -realtime`	Use the `get-vm` cmdlet to obtain a virtual machine object and pipe to `get-stat`.
`get-stat -entity (get-vm ${labWinVM}) -common`	Alternatively:

2. Get default, resource-specific statistics for a virtual machine.

`get-vm ${labWinVM} | get-stat -cpu` `get-vm ${labWinVM} | get-stat -mem` `get-vm ${labWinVM} | get-stat -disk` `get-vm ${labWinVM} | get-stat -net`	Each of these commands retrieves the default number of samples for one of the resource types.

3. Get default statistics for a specific period.

`get-vm ${labWinVM} | get-stat -cpu ` `	This command gets the default CPU counter values for the past 4 hours. The lab roll-ups for VC only run every 2

`-Start (get-date).AddHours(-4) ` `-Finish (get-date)`	hours, so a request for the last one hour will not get data.

4. Get default, resource-specific statistics for a specific interval. An interval can be 0, 5, 30, 120, 1440. You can specify a different number, and the cmdlet will match it to the closest proper interval. For example, entering 1000 would give you the 1440 interval.

`get-vm ${labWinVM} | get-stat -cpu ` `-realtime`	This command retrieves the default CPU counter values for the realtime interval.
`get-vm ${labWinVM} | get-stat -cpu ` `-intervalmins 5` `get-vm ${labWinVM} | get-stat -cpu ` `-intervalmins 30`	Examples of specifying different interval IDs.

5. Specify the counter for which you want performance data. Not all counters have data for every available interval; the data is adjusted based on VirtualCenter performance collection levels.

`get-vm ${labWinVM} | get-stat -stat ` `cpu.ready.summation -realtime`	This command gets the values for the counter CPU Ready for the realtime interval ID.

6. Get an average for CPU Ready for all running virtual machines.

```
get-vm | where {$_.powerstate -eq "poweredon"} | `
foreach {$_ | select name, @{n="avg"; `
e={$_ | get-stat -stat cpu.ready.summation -intervalmins 0 | `
```

```
foreach {$s=0;$c=0}{$s+=$_.Value;$c++}{$s/$c}}}} `
| Format-Table -autosize
```

7. Get an average for each default counter for the given virtual machine.

```
get-vm ${labWinVM} | get-stat -realtime | group MetricId | `
foreach {$_ | select name, @{n="Avg"; `
e={$_.Group | foreach {$s=0;$c=0}{$s+=$_.Value;$c++}{$s/$c}}}} `
| Format-Table -autosize
```

5b. Advanced (optional): Getting Available Counters

When troubleshooting or trying to understand what counters are available it is useful to have a script that lists all the available counters. This script uses some advanced VI Toolkit features that are outside of the scope of this lab (the specifics are explained in the Perl version of this lab). However, we've included this script here because it is extremely useful.

For the following exercise, open a text editor and type the following.

```
$vmMoRef = (get-vm $args | get-view).Moref
$perfManager = (get-view ServiceInstance).Content.PerfManager
$availCounters = (get-view $perfManager).QueryAvailablePerfMetric( `
       $vmMoRef,(Get-Date).AddHours(-1),(Get-Date),20)
$myHash = @{}
(get-view $perfManager).PerfCounter | `
foreach  {$myHash.Add($_.Key, $_.GroupInfo.Key + "." + `
$_.NameInfo.Key + "." + $_.RollupType)} # + "." + $_.UnitInfo.Key)} `
foreach ($c in $availCounters) {
        $myHash[$c.CounterId]
}
```

Save the file as get-counter.ps1

You can now call this function using the following syntax:

```
.\get-counter.ps1 ${labWinVM} | sort

cpu.extra.summation
cpu.guaranteed.latest
cpu.ready.summation
cpu.system.summation
cpu.usage.average
cpu.usagemhz.average
cpu.usagemhz.average
```

```
      cpu.used.summation
      cpu.wait.summation
       ……cont'd..
```

Hands-on Exercise 6: Creating a VM from a Template

The purpose of this exercise is to automate the task of deploying virtual machines from a template that exists in inventory on your VirtualCenter server. This exercise is similar to Exercise 3 in that it requires a reconfiguration, but in this case, the reconfiguration creates a new object.

Scenario

You are the administrator of VMware Infrastructure at your company and need to deploy numerous virtual machines from a template. Typical examples of such a scenario could be deploying identical virtual machines in a Quality Assurance lab or rapidly provisioning desktop virtual machines for VMware Virtual Desktop Infrastructure.

Tasks

1. Import the VMware VI Toolkit snapin.
2. Connect to a server and authenticate.
3. Disconnect from the server.

Step-by-step instructions

1. The following code deploys a new VM from a template. A simple loop will allow you to deploy multiple virtual machines (if you do so, please be considerate of the other lab students and lab staff by limiting the deployment to a few virtual machines).

```
new-vm -template `
  (get-template "vmw9-1-winsrvVM-DEV-Template") `
  -name "Student${labID}_VM2" `
  -host (get-vmhost vmw9-1-esx${labID}.vmworld.com) `
  -location (get-folder –name “Student${labID}”) `
  -pool (get-resourcepool Student${labID}) `
  -Datastore (get-datastore vmw9-1-esx${labID}-lun0)
```

The `Connect-VIServer` cmdlet will connect to the virtual infrastructure so that you can communicate with it.

Advanced Exercises (optional)

The advanced exercises that follow provide some additional examples of things you can do with the toolkit. Because the basic concepts have already been covered, these exercises contain little explination… that's left to the student to learn and explore.

Advanced Exercise 7: Working with datastores and virtual disk files

The goal of this exercise is to facilitate an understanding of the datastores and virtual disk files. This exercise works with both ESX and VC servers.

7a. List the virtual disk files for every VM

Scenario:

You are the VMware Infrastructure administrator and would like to list the virtual disk files for every virtual machine.

Tasks:

Step-by-Step Instructions:

1. Get the VI .NET virtual machine view objects

`$Vms = Get-VM | Get-View`	Use the get-view cmdlet to obtain a VI .NET view object
`PrintVms_Volume_Path($Vms)`	Pass the array of VI .NET virtual machine view objects to the PrintVms_Volume_Path function

2. Function definition

`function PrintVms_Volume_Path($Vms){` `  foreach ($Vm in $Vms) {` `  $Vm.Config.Hardware.Device | where { $_.DeviceInfo.Label -like 'Hard Disk*' } |` `  %{ Write-Host $Vm.Name $_.DeviceInfo.Label  $_.CapacityInKB "KB" $_.Backing.Filename }`	Retrieve the properties of every virtual disk for every virtual machine.

```
  }
}
```

7b. List all files in a datastore

Scenario

You are the VMware Infrastructure administrator and would like to list all the files (or a specific type of file) on datastores

Step-by-Step Instructions

`$Dss = Get-Datastore | Get-View`	Use the get-view cmdlet to obtain a VI .NET view object
`info_datastore_files($Dss)`	Pass the array of VI .NET datastore view objects to the info_datastore_files function

Function definition

```
function info_datastore_files($Dss) {
  $SearchSpec = New-Object VMware.Vim.HostDatastoreBrowserSearchSpec
  $SearchSpec.MatchPattern = "*"
  foreach($Ds in $Dss) {
    Write-Host
    "Datastore Name: "  $Ds.Info.Name  "URL: " $DS.Info.Url  `
    "Free Space: "  $Ds.Info.FreeSpace   "Max File Size: "  $Ds.Info.MaxFileSize
    $Browser = get-view ($Ds.Browser)
    $DsPath = "[" + $ds.Info.Name + "]"
    $Task = $Browser.SearchDatastoreSubFolders_Task($DsPath,$SearchSpec)
    $TaskView = get-view ($Task )
    foreach($Folders in $TaskView.Info.Result) {
      foreach($File in $Folders.File) {
```

1. Build a HostDatastoreBrowserSearchSpec object – which specifies what files to search for
2. Loop through the array of datastore views and print the Name, URL, Free space and Max File Size for every datastore.
3. Obtain the Datastore Browser view
4. Invoke the SearchDatastoreSubFolders_Task(,) method of the Datastore Browser view
5. Loop through the results and print the path for every file

```
        Write-Host $File.Path
      }
    }
  }
}
```

Hands-on Exercise 8: Check Permissions

The purpose of this lab is to check if a specific user has the requisite privileges to perform a specific operation.

Scenario:

You are the administrator of the VMware Virtual Infrastructure at your company and you need an automated way to check if certain users have the requisite permissions to perform certain operations. For example, you may need to confirm that a NOC user has the ability to power-cycle certain VMs.

Tasks:

4. Check that a given user has the privilege to perform the given operation on a given entity which could be either an ESX host or a VirtualMachine.

Step-by-Step Instructions:

```
function CheckUserPermission($AuthMgr)
{
$Permissions = $AuthMgr.RetrieveEntityPermissions(
$Entity.MoRef , $true )
foreach($Permission in $Permissions)
{if($Permission.Principal -eq  $User){
$RoleID  = $Permission.roleid }}

$RoleList = $AuthMgr.RoleList
foreach($Role in $RoleList)
{if($Role.roleid -eq  $roleid)
{ foreach($Privilege in $Role.Privilege ) {
if($Privilege -eq $Operation) { return $true} } }
}
```

Write a function that will be used by the Main program to actually check permissions of a user against a given entity and operation.

Get the Roleid of the User.

Get the privileges assigned to the user from the roleid.

```
return $false
}

Connect-VIServer -Server $labVC -User $labLogin -
Password vmware
$svcRef = new-object VMware.Vim.ManagedObjectReference
$svcRef.Type = "ServiceInstance"
$svcRef.Value = "ServiceInstance"
$servicInstance = Get-view $svcRef

$authorizationManager = Get-View
$servicInstance.Content.AuthorizationManager

Write-Host "Please enter the Entity Type [HostSystem |
VirtualMachine]:"
$EntityType = Read-Host
Write-Host "Please enter the Entity Name:"
$EntityName = Read-Host

if ($EntityType -eq "HostSystem") {
  $entity = Get-Host $EntityName
}

if ($EntityType -eq "VirtualMachine") {
  $entity = Get-VM $EntityName | Get-View
}

Write-Host "Please enter the user (with domain):"
$User = Read-Host

Write-Host "Please enter the operation (Like:
VirtualMachine.Interact.PowerOff):"
$Operation = Read-Host
```

This is the main program
Use the Connect-VIServer to connect to the lb VC server

```
$Granted = CheckUserPermission($authorizationManager)
if($Granted -eq $true)
{
Write-Host "User " $User " has the privilege to perform
the " $Operation " operation on " $EntityName
}
else
{
Write-Host "User " $User " does not have the privilege
to perform the " $Operation " operation on "
$EntityName
}
```

Save the file as CheckPermissions.ps1

Run the script by calling ./CheckPermissions.ps1

Advanced Exercise 9: Power Operations and Maintenance

The purpose of this lab is to automate the task of shutting down or relocating virtual machines. This exercise consists of two parts; virtual machine and host. The first part involves shutting down a batch of virtual machines. The second part involves the host and makes functional decisions based on user permissions. The third part of this exercise takes a look at reconfiguring a cluster's HA state.

Scenario

You are the administrator of the VMware Virtual Infrastructure at your company and need an automated way to handle the evacuation of an ESX host in the event that a loss of power is imminent. Think of this as the UPS software integration script for ESX: something would trigger this script if the UPS had 15 minutes of battery life remaining.

Tasks

1. Shut down a group of virtual machines on a given host
2. Shut down a host
    a. Cluster Validation – Checking the DRS configuration
    b. Permissions Validation – Ensure that a given user has permissions to put a host into maintenance mode
    c. Decision time: Shut down the VMs or use DRS to evacuate the host?
3. Disable VMware HA on a cluster

Instructions

This is an advanced exercise, so step-by-step instructions will not be provided. Instead, we will touch on the highlights of the code.

The Bottom Line – Part 9.1

We need a list of virtual machines that are currently powered up and match a given name pattern. Once we have that list, we must call the `Stop-VM` commandlet and pass it that list.

`Connect-VIServer --Server $labVC --User $labLogin`	Connect to the VirtualCenter server

Once attached to the VirtualCenter server, this script becomes a simple one-liner, which we examine here.

`Get-VM -Name "*DEV" | ` `Where { $_.PowerState -eq "PoweredOn" } | ` `% { Stop-VM -VM $_ }`	Locate all VMs whose name ends with **DEV** and whose powerstate is PoweredOn Walk through that list and pass each to **Stop-VM**. Note here that we used the shortcut **%**. In this context, this is a quick way of writing **foreach**

To change the scope of our exercise to focus on a specific ESX host machine, we can either login to that host and execute the code above, or we can get a reference to the host in question (MYESXSERVERNAME) and use the `-Location` option of the `Get-VM` commandlet to filter.

`$Host = Get-VMhost MYESXSERVERNAME` `Get-VM -Name "*DEV"` **`-Location $Host`** `| ` `Where { $_.PowerState -eq "PoweredOn" } | ` `% { Stop-VM -VM $_ }`	Locate all VMs whose name ends with **DEV** and whose powerstate is PoweredOn Walk through that list and pass each to **Stop-VM**. Note here that we used the shortcut **%**. In this context, this is a quick way of writing **foreach**

The Bottom Line – Part 9.2

In this exercise, we need to shut down a host machine in the quickest, automated manner possible while maintaining uptime of our virtual machines, if the configuration and permissions make that possible. In the absence of VMotion, we can either Shutdown/Power off the virtual machines or suspend them. If DRS is enabled and in Fully Automated mode, we can simply put the host into Maintenance Mode and allow DRS to do its thing. If DRS is not available, but VMotion is possible, the script can be modified to enumerate and VMotion the machines off the host. However, that is beyond the scope of this exercise.

The core of this script is relatively straightforward, as illustrated from this outline:

1. Connect to the VirtualCenter server
2. Attach to the Authorization Manager and verify user has required permission
3. Determine cluster DRS configuration
4. Decision and action

The first step was introduced in Exercise 1 and used throughout the lab. The second step is a little more complicated because

there is no commandlet to directly attach to the Authorization Manager; this requires diving into the API a little:

```
$svcRef = `
   New-Object VMware.Vim.ManagedObjectReference
$svcRef.Type = "ServiceInstance"
$svcRef.Value = "ServiceInstance"
$serviceInstance = Get-View $svcRef

#Get Authorization manager to check user permission
$AuthMgr = Get-View `
   $serviceInstance.Content.AuthorizationManager

#Check user permission
```

The Authorization Manager is a property of the ServiceContent object and must be accessed via that managed object. In Powershell, the **New-Object** commandlet is used to create the object and **Get-View** is used to retrieve it.

Once we have a reference to the ServiceContent, we use that to access the Authorization Manager

An understanding of the VI Roles, Permissions, Privileges model is necessary to write the permission checking code. The API documentation provides the architecture and should be referenced.

The third step should seem pretty simple following the journey into the API in step 2. In this step, we need to look at the cluster and determine whether DRS is configured and which mode is being used (Partially Automated or Fully Automated). The mode makes a difference because a cluster in Fully Automated mode will automatically evacuate a host's virtual machines when that host is placed into Maintenance Mode. A cluster in Partially Automated mode can only **recommend** that the virtual machines be moved.

This step is broken into two parts: determining which cluster contains our host and checking the cluster's configuration.

```
Write-Host "Please enter the name of the Host:"
$HostName = Read-Host

$HostServer = Get-VMHost $HostName
$Entity = $HostServer | Get-View
$Clus = Get-Cluster -VMHost $HostServer
```

This script demonstrates basic user input request and processing. It prompts for a hostname then uses that to determine which cluster contains the host.

The relevant properties of the cluster object are

`$Clus.DrsEnabled` `$Clus.DrsMode`	shown. **DrsEnabled** is a Boolean (use $true and $false Powershell variables for comparison) **DrsMode** has a string value of "FullyAutomated" or "PartiallyAutomated"

This code outline shows how to check the DRS setting of the cluster and act accordingly based on the current permissions and configuration. The CheckPer() function is something that you must write. It uses the Authorization Manager to determine whether the user has the **Host.Config.Maintenance** permission on the host object. This is the permission required to send a host into Maintenance Mode.

```
If( (($Clus.DrsEnabled -eq $true) -and `
   ($Clus.DrsMode -eq "FullyAutomated")) –and `
   ($CheckPer -eq $true) ) {
      #Put the host on the maintenance mode
      $HostView.EnterMaintenanceMode_Task( 0, $true )
} else {
   #Shutdown the all powered-on VMs on the host
   ShutDownVMs( $HostServer )
}
```

The $HostView variable is a VI API view of the target host. This is needed to be able to call the **EnterMaintenanceMode_Task** method and is obtained using the **Get-View** commandlet.

The **ShutDownVMs** function here is the functionality derived in section 9.1, wrapped into a Powershell user-defined function using the **function NAME($variable){ ACTIONS }** construct

The SOLUTION file for this exercise contains the full code.

The Bottom Line – Part 9.3

If a datacenter power failure is imminent, and it will affect more than a single host, it is desirable to disable VMware HA functionality to prevent the resurrection of virtual machines on hosts that may ultimately be power off abruptly. This exercises takes a look at how that would be accomplished using a script.

```
$ConfigSpec =
   New-Object Vmware.Vim.ClusterConfigSpec
$ConfigSpec.DasConfig =
   New-Object vmware.Vim.ClusterDasConfigInfo
$ConfigSpec.DasConfig.Enabled = $false
$ClusMor.ReconfigureCluster_Task($ConfigSpec, $true)
```

The modification of a managed object generally involves creating a ConfigSpec, making changes to that spec, then applying that ConfigSpec to the object using a **ReconfigureX_Task** method as indicated here.

In this example, the **DasConfig** property is a nested managed object that must be created under the ClusterConfigSpec object.

To explain why this information is located under the **DasConfig** property, some background in VI history may be helpful. Originally, **VMware HA** (High Availability) was called Distributed Availability Services, **DAS**, and the API retains these names.

This functionality can be wrapped into a function and called prior to shutting down an ESX host.

Advanced Exercise 10: Trapping Errors

In this exercise, we take Exercise 9.2 and look at where some error checking could be applied to trap for common errors. When it encounters a terminating error, PowerShell's default behavior is to display the error and halt the command or script execution. To use custom error handling for a terminating error, you define an exception trap handler to prevent the ErrorRecord from being sent to the default error-handling mechanism. The same holds true for nonterminating errors, but PowerShell's default behavior is to display the error and continue execution.

This trap command tells PowerShell to trap all errors (there is an optional parameter before the opening brace that defaults to trapping everything), write the error to the console in red text, then rethrow the exception and kill the existing execution context. Other options are Continue and Return, which will either continue execution or break and return a specified value.

```
Trap {
   Write-Host ( "ERROR: " + $_ ) `
      -ForegroundColor Red; Break
}
```

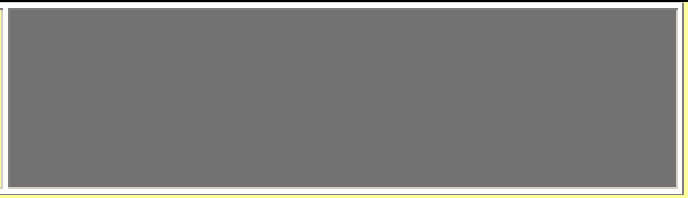
This is modified code for obtaining a reference to an ESX host and determining its cluster membership. If the host is not a member of a cluster, an error message will be printed.

```
#Get the host and cluster from the VM

$HostServer = Get-VMHost $HostName -ErrorAction Stop

$Entity = $HostServer | Get-View

$Clus = Get-Cluster -VMHost $HostServer `
   –ErrorVariable ClusFound

If ( $ClusFound ) {
   Write-Host `
    "Cluster not found for host: " + $HostServer.Name
   return;
```

In this case, we define an error action as part of the commandlet call. This enables an error to be reported, ignored, cause an abrupt exit, or prompt the user. This is implemented within PowerShell, and more information is available by executing the Powershell command
get-help about_CommonParameters

Here, we use an ErrorVariable to capture the error then check against that variable and act in a specific way.

```
} else {
   Write-Host `
    "Cluster found for host: "  $HostServer.Name
}
```



Other than these mechanisms, liberal use of comments and **Write-Host** statements can assist with debugging your code.

Appendix A: Lab setup

The lab we used consisted of a number of ESX servers managed by the same VC server. The VC server managed a number of VMs, some configured to be read-only to the users and others configured to be read/write by the users. The VC should have a DRS cluster configured.

While the names used for many of the VMs are not important, some exercises find VMs based on particular criteria.

Each student should have the following:

1. A folder called Student<ID>
2. A resource pool called Student<ID>
3. The folder Student<ID> should have a VM called Student01_VM1. You can use another VM, as long as <VM> points to it. This VM should allow <login> to have "VMworld rw" permissions.
4. A virtual port group called "Production_LAN". This can be internal only.
5. A resource pool called "Admin RP"
6. The Admin RP resource pool should have a VM called vmw9-1-winsrv01. You can use a different name, as long as <vm-ro> points to it.
7. A role called "VMworld global". It should be a copy of the "read-only" role with the following additional permissions:
    a. Global:Diagnostics
    b. VM:provisioning:customize
    c. VM:provisioning:deploy template
    d. VM:provisioning:read customization spec
    e. VM:provisioning:modify customization spec
    f. Datastore:Browse datastore
8. A role called "VMworld rw". It should have the following permissions:
    a. Global: Log Event
    b. Global: Diagnostics
    c. Datastore:Browse datastore

   d. Datastore:File management
   e. Host:Configuration:Local Operations:Create VM
   f. Host:Configuration:Local Operations:Delete VM
   g. Virtual Machine (all check boxes)
   h. Resource:Assign VM to Resource Pool
   i. Resource:Apply Recommendation
   j. Resource:Migrate
   k. Resource:Relocate
   l. Resource:Query Vmotion
9. A login called l09-student01. You can use a different login, but set the <login> variable appropriately.
10. Add "VMware Global" permission for <login> for all users at the hosts&clusters level (above the datacenter)
11. For the <login> user, remove "VMworld rw" permissions from everywhere but the student resource pools and folders.
12. Add the "VMworld rw" permission to each student resource pools and folders for the corresponding <login>
13. Create a new template. It should be a clone ofa windows VM.
14. Create a VM called vmw9-1-winsrvVM-DEVas a clone of the template. Give it 2 vCPUs and connect it to the "Production_LAN" port group. Put it in the "Admin RP" resource pool.

For large lab environments, where many students will be trying this lab on the same VC server at the same time (and only in many-student environments), there may be an issue with the number of open SDK sessions. This is generally not an issue with "normal" production environments, but large numbers of users opening (and possibly never closing) SDK connections over the course of a lab like this could open a lot of connections. For instance if students are doing all the exercises and requiring 4 attempts for each, may generate 80 or more connections per user over the course of the lab.

To avoid this issue, classroom environments with a "shared" VC approach may want to make the following change to increase the number of allowed SDK sessions in VC:

Make the following addition (see in red below) to the vpxd.conf file (Located in C:\Documents and Settings\All Users\Application Data\VMware\VMware VirtualCenter) on the Virtual Center Server, then restart the Virtual Center service.

```
<config>
        <!-- <ws1x> -->
                <!-- Enables 1.x Web Services compatibility -->
                <!-- <enabled>true</enabled> -->
                <!-- Enables 1.x Built in Perf data Refresh -->
                <!-- <perfRefreshEnabled>false</perfRefreshEnabled> -->
                <!-- Location of ws1x persistent data. Default is same directory as vpxd.cfg -->
                <!-- <dataFile> C:\ws1x.xml </dataFile> -->
                <!-- Location of the ws1x event mapping file. Default is the same directory as vpxd.cfg -->
                <!-- <eventMap> C:\ws1xEventMap.xml </eventMap> -->
        <!-- </ws1x> -->

        <vpxd>
                <das>
                        <serializeadds>true</serializeadds>
                        <slotMemMinMB>256</slotMemMinMB>
                        <slotCpuMinMHz>256</slotCpuMinMHz>
                </das>
                <filterOverheadLimitIssues>true</filterOverheadLimitIssues>
        </vpxd>
        <vmacore>
                <threadPool>
```

```
            <TaskMax>30</TaskMax>
        </threadPool>
        <soap>
            <maxSessionCount>1000</maxSessionCount>
        </soap>
    </vmacore>
</config>
```

About the VMware Sample Code

The sample code is provided "AS-IS" for use, modification, and redistribution in source and binary forms, provided that the copyright notice and this following list of conditions are retained and/or reproduced in your distribution. To the maximum extent permitted by law, VMware, Inc., its subsidiaries and affiliates hereby disclaim all express, implied and/or statutory warranties, including duties or conditions of merchantability, fitness for a particular purpose, and non-infringement of intellectual property rights. IN NO EVENT WILL VMWARE, ITS SUBSIDIARIES OR AFFILIATES BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES, ARISING OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THE SAMPLE CODE.

You agree to defend, indemnify and hold harmless VMware, and any of its directors, officers, employees, agents, affiliates, or subsidiaries from and against all losses, damages, costs and liabilities arising from your use, modification and distribution of the sample code.

VMware does not certify or endorse your use of the sample code, nor is any support or other service provided in connection with the sample code.

For latest VMware SDK Information visit http://vmware.com/developer

VMware, Inc. 3401 Hillview Ave Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com