vSphere SDK for Perl Programming Guide

ESXi 6.5

vCenter Server 6.5

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-002349-00

vmware®

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

Copyright © 2007–2016 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

About This Book

The *vSphere SDK for Perl Programming Guide,* provides information about writing and running VMware® vSphere SDK for Perl scripts on ESX/ESXi or vCenter Server systems.

IMPORTANT This book discusses the SDK for Perl that allows you to access vSphere by using the vSphere Web Services SDK, which is available for all supported versions of vSphere. This book does not discuss Perl bindings to vAPI, which was released with vSphere 6.0 for the first time.

Because SDK subroutines allow you to manage vSphere hosts using vSphere API calls, a brief description of the server-side object model is included. This guide focuses on explaining how to access and modify server-side objects using the vSphere SDK for Perl and on discussing some programming techniques.

Intended Audience

This book is intended for administrators with different levels of Perl scripting experience:

- All administrators can use the utility applications and sample scripts included with the vSphere SDK for Perl to manage and monitor the hosts in the vSphere environment.
- Experienced Perl programmers can examine the source code for the available scripts. They can then modify those scripts or write new scripts using the vSphere SDK for Perl subroutines to access the objects on the vSphere host and manipulate those objects using Perl. This document includes a discussion of the vSphere object model and explains how you can preview and retrieve the objects and their attributes and methods.

Getting Started with vSphere SDK for Perl

1

The vSphere SDK for Perl lets you automate a wide variety of administrative, provisioning, and monitoring tasks in the vSphere environment. This chapter introduces the SDK architecture, explains the basic use model, and gets you started running a simple script.

The chapter includes the following topics:

vSphere SDK for Perl Architecture

The interaction model between the SDK and the vSphere API on the host directly affects how each script is structured, and is the basis for troubleshooting.

vSphere SDK for Perl subroutines interact with the host and perform variations of these basic tasks:

- Connect to a remote host using user-supplied connection parameters, and disconnect.
- Find objects on the remote host (server-side objects). For example, find all virtual machines on a host.
- Retrieve or modify server-side objects, for example, manage the virtual machine life cycle (start, stop, suspend, and so on).
- Collect information from server-side objects.
- Manage sessions.

Most routines retrieve a vSphere API object and make it available as a Perl object (called a view object) that you can then manipulate with your script.

The vSphere SDK for Perl has these components:

- **vSphere SDK for Perl Runtime –** Client-side runtime components that include:
  - A complete Perl binding of the vSphere API, which makes all server-side operations and data structures available. The SDK handles the data type mapping between server-side and client-side objects transparently.
  - VMware Perl modules (`VIRuntime.pm` and `VILib.pm`) that provide subroutines for basic functionality.
- **vSphere SDK for Perl Utility Applications –** Management applications that you can run without modification in your virtual datacenter. You run each application with connection parameters and other, application-specific parameters. See the *vSphere SDK for Perl Utility Applications Reference.*

- **Sample Scripts –** Scripts that you can customize for your needs and that illustrate the vSphere SDK for Perl's functionality. You must know Perl to customize the scripts. Unlike the utility applications, sample scripts are not supported by VMware.

A vSphere SDK for Perl installation also includes the following libraries:

- **Web Services for Management Perl Library and Examples** – The WS-Management Perl Library allows you to write scripts that retrieve CIM data from the ESXi host using CIMOM, a service that provides standard CIM management functions over a WBEM (Web-Based Enterprise Management). See "Web Services for Management Perl Library" on page 51.
- **Credential Store Library and Examples** – Allows vSphere SDK for Perl applications to manage the vSphere credential store. The credential store allows vSphere SDK for Perl scripts to authenticate themselves to ESX/ESXi or vCenter Server systems. See "Credential Store Perl Library" on page 61.
- vCLI Commands - Set of host management commands and a set of DCLI commands for managing vCenter services.

Using vSphere SDK for Perl

This section explains how to get started with vSphere SDK for Perl by looking at two typical usage scenarios. It also lists common vSphere SDK for Perl tasks and discusses programming conventions.

Getting Started

If you want to use the SDK to retrieve performance information for a host, you might perform the following tasks:

1 Check the *vSphere SDK for Perl Utility Applications Reference* or the `apps` directory for a script that retrieves performance information.

Check `/usr/lib/vmware-vcli/apps` on Linux and `Program Files\VMware vSphere CLI\Perl\apps` on Windows. All utility applications are fully supported.

The `viperformance.pl` script retrieves performance counters from the host.

NOTE If you cannot find a utility application, examine the sample scripts. You can use sample scripts as starting points for your application. On Linux, `/usr/share/doc/vmware-vcli/samples`, on Windows, `Program Files\VMware\VMware vSphere CLI\Perl\samples`. Sample scripts are not supported.

---

2 Run the script with the `--help` option or without any options to see its online documentation. More detailed information is in the *Utility Applications Reference* included in the vSphere SDK for Perl documentation set and available from the VMware Web site.

3 Run the `viperformance.pl` script against an ESXi host.

```
viperformance.pl --url https://<host>:<port>/sdk/vimService --username nemo --password fi\$h
                 --host Aquarium --countertype net --interval 30 --samples 3
```

Escape characters must precede special characters in passwords. See Table 1-3, "Options Available for All SDK for Perl Commands," on page 16 for a complete list of connection parameters.

If you want to use the SDK for a task that none of the utility applications can perform, you might perform the following tasks:

1 Check the `/samples` folder for a sample script that performs a similar task. The scripts in the `samples` folder are available for customization.

2 If a script that performs a similar task is available, modify the script. If none of the scripts is suitable, write a new script using the vSphere SDK for Perl subroutines.

The following materials are available for modifying or writing scripts:

Source	Description
"Writing vSphere SDK for Perl Scripts" on page 19.	In-depth discussion of scripts that includes an example.
Chapter 4, "vSphere SDK for Perl Subroutine Reference," on page 41.	Reference to vSphere SDK for Perl subroutines.
"Web Services for Management Perl Library" on page 51.	Allows you to write scripts that retrieve CIM data from the ESX/ESXi host using CIMOM, a service that provides standard CIM management functions over a WBEM (Web-Based Enterprise Management).
"Credential Store Perl Library" on page 61.	Allows vSphere SDK for Perl applications to manage the vSphere credential store.
vSphere API Reference documentation.	Reference to the server-side object your script interacts with.

3 Follow these programming conventions when you modify or create vSphere SDK for Perl scripts:

- Use parameter names followed by parameter values, as follows:

```
Vim::<subroutine>(<parameter_name>=><value>, <parameter_name>=><value> );
Util::<subroutine>(<parameter_name>=><value>, <parameter_name>=><value> );
Opts::<subroutine>(<parameter_name>=><value>, <parameter_name>=><value> );
```

- Use the options in Table 1-3, "Options Available for All SDK for Perl Commands," on page 16to specify connection information.
- Use the mechanism discussed in "Step 2: (Optional) Define Script-Specific Command-Line Options" on page 20 for specifying additional script-specific options.

Common vSphere SDK for Perl Tasks

The vSphere SDK for Perl includes utility applications and sample scripts for common administration tasks.

Table 1-1. Common Administrative Tasks and SDK Utilities

Task	Script	Location
Discovery (logging in)	`connect.pl`	`/apps/general`
Performance monitoring	`viperformance.pl` (retrieves performance counters from host)	`/apps/performance`
Virtual machine power operations	`vmcontrol.pl`	`/apps/vm`
Virtual machine snapshot and restore functionality	`vmsnapshot.pl`, `snapshotmanager.pl`	`/apps/vm`
Virtual machine migration	`vmmigrate.pl`	`/apps/vm`
Host operations, for example, adding a standalone host to a vCenter Server system, shutting down and rebooting a host, and so on	`hostops.pl`	`/apps/host`
Viewing or changing of CPU or memory share allocation on a virtual machine	`sharesmanager.pl`	`/apps/vm`

Some tasks require additional scripting. See Chapter 2, "Writing vSphere SDK for Perl Scripts," on page 19.

vSphere SDK for Perl Programming Conventions

Several programming conventions are different than you might expect because the SDK interacts with a server using SOAP/WSDL.

- Boolean data types – SDK applications send and receive Boolean values as follows:
  - Input (sending from the client application):

false: Use 0, '0', or 'false' (capitalization ignored)

true: Use 1, '1', or 'true' (capitalization ignored)

- Output (receiving from the server):

  false: Return value is 0

  true: Return value is 1

To match Boolean values in a filter, use the strings true and false. See "Creating and Using Filters" on page 33.

- Date/Time – The server returns a SOAP `dateTime` value. You can use the `Date::Parse` Perl module to process these objects.

  The vSphere SDK for Perl accepts only native SOAP `dateTime` values using standard date time format with or without fractional seconds, and with or without GMT (Z) time zone:

  `YYYY-MM-DDThh:mm:ssTZD`, for example, `1997-07-16T19:20:30+01:00`
  `YYYY-MM-DDThh:mm:ss.sTZD`, for example, `1997-07-16T19:20:30.45+01:00`

  The SDK always returns `dateTime` values in the standard date time format.

- SOAP error message – Most likely indicates an error on the server, not an error with the communication to the server.

vSphere SDK for Perl Common Options

A number of options are available for any vSphere SDK for Perl script. Most of these options allow you to specify the host or hosts to connect to. Most options require an option value.

```
perl <app_name>.pl --<option_name> <option_value>
```

For example, to power on a virtual machine using the `vmcontrol.pl` utility application, you must specify the name of the virtual machine to power on, as follows:

```
perl vmcontrol.pl --server <myserver> --username <admin> --password <mypassword> --operation
               poweron --vmname <virtual_machine_name>
```

Run any application or sample without any options or with `--help` to see its parameters and execution examples. Information about common and script-specific options is included.

IMPORTANT If the host you are targeting is in lockdown mode, you cannot execute Perl scripts against the host.

Specifying Options

You can specify the common options in several ways, discussed in this section.

When you run a vSphere CLI command, authentication happens in the following order of precedence:

Table 1-2. vSphere CLI Authentication Precedence

Authentication	Description	See
Targeting vCenter Server and Using vCenter Single Sign-On	If you are working in an environment that is managed by vCenter Single Sign-On, you can specify the vCenter Single Sign-On server and a user name, password, and target host.	"Using a Session File" on page 13
Command line	Password (`--password`), session file (`--sessionfile`), or configuration file (`--config`) specified on the command line.	"Using a Session File" on page 13
Configuration file	Passwords specified in a `.visdkrc` configuration file.	"Using a Configuration File" on page 15

Table 1-2. vSphere CLI Authentication Precedence (Continued)

Authentication	Description	See
Environment variable	Password specified in an environment variable.	"Setting Environment Variables" on page 14
Credential store	Password retrieved from the credential store.	"Credential Store Perl Library" on page 61
Current account (Active Directory)	Current account information used to establish an SSPI connection. Windows only.	."Using Microsoft Windows Security Support Provider Interface (SSPI)" on page 15
Prompt the user for a password.		

This order of precedence always applies. That means, for example, that you cannot override an environment variable setting in a configuration file.

Authenticating Through vCenter Server and vCenter Single Sign-On

For all ESXi hosts that are managed by a vCenter Server system that is integrated with vCenter Single Sign-On 6.0 and later, you can authenticate directly to the vCenter Server system, or you can authorize to vCenter Server through vCenter Single Sign-On.

Best practice is to authenticate through vCenter Single Sign-On. The vCenter Single Sign-On service is included in the Platform Services Controller. The Platform Services Controller can be embedded in your vCenter Server installation, or one Platform Services Controller can handle authentication, certificate management, and some other tasks for multiple vCenter Server systems.

NOTE You cannot use this approach if vCenter Server is integrated with vCenter Single Sign-On 5.0.

You use the `--psc` option and, optionally, the `--server` option.

- `psc` - Specifies the Platform Services Controller instance associated with the vCenter Server system that manages the host.
- `server` - Specifies the vCenter Server system that manages the host. Required if the Platform Services Controller instance is associated with more than one vCenter Server system.
- `vihost` - Specifies the ESXi host, as in earlier versions of vCLI.

Example

```
vminfo.pl --server <vc_HOSTNAME_OR_IP> --psc <psc_HOSTNAME_OR_IP> --vihost
              <esxi_HOSTNAME_OR_IP>--username root --password vmware --vihost <esxi_host>
```

If the specified user is known to vCenter Single Sign-On, a session is created. You can save the session with the `--savesessionfile` argument, and later use that session with the `--sessionfile` argument.

Using a session file results in less overhead and better performance than connecting to the Platform Services Controller repeatedly.

Using a Session File

The `save_session.pl` script in the `apps/session` directory illustrates how to create a session file. You can modify the script and include it in your own application, or create a session file using the script on the command line, and then pass in that session file when running vSphere SDK for Perl commands. See "Saving Sessions" on page 36.

The session file does not reveal password information. If a session file is not used for 30 minutes, the session expires.

To create and use a session file

1 Connect to the directory where the script is located, for example, on Windows:

```
cd C:\Program Files\VMware\VMware vSphere CLI\Perl\apps\session
```

2 Run `save_session.pl`. You must supply connection parameters and the name of a session file in which the script can save an authentication cookie.

```
perl save_session.pl --savesessionfile <location> --server <esxi_host>
```

For example:

```
perl save_session.pl --savesessionfile C:\Temp\my_session --server my_server
```

If you specify a server but no user name or password, the script prompts you.

3 You can now run scripts in the `\apps` or `\samples` directory or your own scripts and pass in the session file using the `--sessionfile` parameter as follows:

```
<command> --sessionfile <sessionfile_location> <command_options>
```

For example:

```
perl hostinfo.pl --sessionfile C:\Temp\my_session
```

NOTE If you use a session file, any other connection parameters are ignored.

You can use the code in the `\apps\session\save_session.pl` utility application inside your own vSphere SDK for Perl application. If a call to the server throws an exception, your application should terminate the session to avoid session leaks. You could do this with an error handler that runs `disconnect()` or `logout()`, for example:

```
eval {
    # ... insert program here ...
  };
  if ($@) {
    print "Fatal error: $@";
    Util::disconnect();
    exit(1);
  }
```

You can also use the `_END_` pseudo-signal handler to perform a disconnect, as follows:

```
$SIG{__END__} = sub { Util::disconnect(); }
```

Passing Parameters at the Command Line

Pass parameters at the command line using option name and option value pairs (some options have no value).

```
--<optionname> <optionvalue>
```

The following example connects to the server as user `snow-white` with password `dwarf$`. The first example (Linux) uses an escape character before each special character, the other examples use single quotes (Linux) and double quotes (Windows).

Linux

```
vminfo.pl --server <server> --username snow\-white --password dwarf\$ --vmname <name>
vminfo.pl --server <server> --username 'snow-white' --password 'dwarf$' --vmname <name>
```

Windows

```
vminfo.pl --server <server> --username "snow-white" --password "dwarf$" --vmname <name>
```

Setting Environment Variables

You can set environment variables in a Linux profile, in the Environment properties dialog box of the Microsoft Windows System control panel, or, for the current session, at the command line. Environment variables are listed when you run a command with `--help`.

The following example shows the contents of a `/root/.visdkrc` file that uses environment variables:

```
VI_SERVER = <server>
VI_USERNAME = <usr>
VI_PASSWORD = <root_password>
```

```
VI_PROTOCOL = https
VI_PORTNUMBER = 443
```

Do not escape special characters in the file that specifies environment variables.

If you have set up your system to run this file, you can run scripts on the specified server afterwards.

Using a Configuration File

A configuration file is a text file that contains variable names and settings. Variables corresponding to the connection options are shown in Table 1-3, "Options Available for All SDK for Perl Commands," on page 16. Use `--config` if the configuration information is saved in a different file than `./visdkrc`. If you specify `--config`, the system ignores the `./visdkrc` settings.

CAUTION Limit read access to a configuration file, especially if it contains user credentials.

You can use the `--config` option to run a script with the configuration file, for example:

```
connect.pl --config <my_saved_config> --list
```

Using a configuration file is useful for repeatedly entering connection details. If you have multiple vCenter Server or ESX/ESXi systems and you administer each system individually, you can create multiple configuration files with different names. When you want to run a command or a set of commands on a server, you pass in the `--config` option with the appropriate filename at the command line.

Using Microsoft Windows Security Support Provider Interface (SSPI)

You can use the `--passthroughauth` command-line argument to log in to a vCenter Server system (vCenter Server version 2.5 Update 2 or later). Using `--passthroughauth` passes the credentials of the executing user to the server. If the executing user is known by both the machine from which you access the vCenter Server system and the machine running the vCenter Server system, no additional authentication is required.

NOTE Using SSPI is supported only when you run commands from Windows, and use a vCenter Server system as the target server.

If SDK commands and the vCenter Server system run on the same machine, a local account for the executing user works. If they run on different machines, then the executing user must have an account in a domain trusted by both machines.

SSPI supports a number of protocols. By default, it selects the `Negotiate` protocol, which indicates that client and server attempt to find a mutually supported protocol. Alternatively, you can use `--passthroughauthpackage` to specify another protocol supported by SSPI. Kerberos, the Windows standard for domain-level authentication, is commonly chosen.

If the vCenter Server system is configured to accept only a specific protocol, specifying the protocol to vSphere SDK for Perl commands with `--passthroughauthpackage` might be required for successful authentication to the server. If you use `--passthroughauth`, you do not have to specify authentication information in any other way. For example, to run `connect.pl` on the server, you can use the following command at the command line.

```
<command> <login_params> --passthroughauth
```

See the Microsoft Web site for a detailed discussion of SSPI.

The following example connects to a server that has been set up to use SSPI. When you run the command, the system calls `vminfo.pl` with the `--vmname` option. The system does not prompt for a user name and password because the current user is known to the server.

```
vminfo.pl --server <vc_server> --passthroughauth --passthroughauthpackage "Kerberos"
        --vihost my_esx --vmname <name>
```

Common Options Reference

The following table lists options that are available for all vSphere SDK for Perl scripts. Use the parameter on the command line and the variable or the parameter in configuration files.

Table 1-3. Options Available for All SDK for Perl Commands

Option and Environment Variable	Description

`--cacertsfile <certsfile>`
`-t <certs_file>`
`VI_CACERTFILE=<cert_file_path>`
	ESXCLI commands only.
	Used to specify the CA (Certificate Authority) certificate file, in PEM format, to verify the identity of the vCenter Server system or ESXi system to run the command on.
	In vCLI 6.0 and later, you can only run ESXCLI commands if a trust relationship exists between the host you are running the command on and the system you are targeting with the `--server` option (ESXi host or vCenter Server system). You can establish the trust relationship by specifying the CA certificate file or by passing in the thumbprint for each target server (ESXi host or vCenter Server system).

`--config <cfg_file_full_path>` `VI_CONFIG=<cfg_file_full_path>`	Uses the configuration file at the specified location. Specify a path that is readable from the current directory.
`--credstore <credstore>` `VI_CREDSTORE=<credstore>`	Name of a credential store file. Defaults to `<HOME>/.vmware/credstore/vicredentials.xml` on Linux and `<APPDATA>/VMware/credstore/vicredentials.xml` on Windows. Commands for setting up the credential store are included in the vSphere SDK for Perl, which is installed with vCLI. The *vSphere SDK for Perl Programming Guide* explains how to manage the credential store.
`--encoding <encoding>` `VI_ENCODING=<encoding>`	Specifies the encoding to be used. Several encodings are supported. ■ `utf8` ■ `cp936` (Simplified Chinese) ■ `shftjis` (Japanese) ■ `iso-885901` (German). You can use `--encoding` to specify the encoding vCLI should map to when it is run on a foreign language system.
`--passthroughauth` `VI_PASSTHROUGHAUTH`	If you specify this option, the system uses the Microsoft Windows Security Support Provider Interface (SSPI) for authentication. Trusted users are not prompted for a user name and password. See the Microsoft Web site for a detailed discussion of SSPI. This option is supported only if you are connecting to a vCenter Server system.
`--passthroughauthpackage <package>` `VI_PASSTHROUGHAUTHPACKAGE=<package>`	Use this option with `--passthroughauth` to specify a domain-level authentication protocol to be used by Windows. By default, SSPI uses the `Negotiate` protocol, which means that client and server try to negotiate a protocol that both support. If the vCenter Server system to which you are connecting is configured to use a specific protocol, you can specify that protocol using this option. This option is supported only if you are running vCLI on a Windows system and connecting to a vCenter Server system.
`--password <passwd>` `VI_PASSWORD=<passwd>`	Uses the specified password (used with `--username`) to log in to the server. ■ If --server specifies a vCenter Server system, the user name and password apply to that server. If you can log in to the vCenter Server system, you need no additional authentication to run commands on the ESXi hosts that server manages. ■ If --server specifies an ESXi host, the user name and password apply to that server. Use the empty string `(' '` on Linux and “ “ on Windows) to indicate no password. If you do not specify a user name and password on the command line, the system prompts you and does not echo your input to the screen.
`--portnumber <number>` `VI_PORTNUMBER=<number>`	Uses the specified port to connect to the system specified by `--server`. Default is 443.

Table 1-3. Options Available for All SDK for Perl Commands

Option and Environment Variable	Description
`--protocol <HTTP|HTTPS>` `VI_PROTOCOL=<HTTP|HTTPS>`	Uses the specified protocol to connect to the system specified by `--server`. Default is HTTPS.
`--psc <hostname_or_IP>` `VI_PSC=<hostname_or_IP>`	Host name or IP address of the Platform Services Controller instance that is associated with the vCenter Server system that manages the host. In many cases, the Platform Services Controller is embedded in the vCenter Server system, but external Platform Services Controller instances are supported as well. For those cases, use the `--server` option to specify the vCenter Server system that manages the host. This option implies user authentication with vCenter Single Sign-On. The user you specify must be able to authenticate to vCenter Single Sign-On.
`--savesessionfile <file>` `VI_SAVESESSIONFILE=<file>`	Saves a session to the specified file. The session expires if it has been unused for 30 minutes.
`--server <server>` `VI_SERVER=<server>`	Uses the specified ESXi or vCenter Server system. Default is `localhost`. If `--server` points to a vCenter Server system, you can also specify the `--psc` option to log in to the vCenter Server system with vCenter Single Sign-On. Use the `--vihost` option to specify the ESXi host that you want to run the command against. See "Authenticating Through vCenter Server and vCenter Single Sign-On" on page 13.
`--servicepath <path>` `VI_SERVICEPATH=<path>`	Uses the specified service path to connect to the ESXi host. Default is `/sdk/webService.`
`--sessionfile <file>` `VI_SESSIONFILE=<file>`	Uses the specified session file to load a previously saved session. The session must be unexpired.
`--thumbprint <thumbprint>` `VI_THUMBPRINT=<thumbprint>`	Expected SHA-1 host certificate thumbprint if no CA certificates file is provided in the `--cacertsfile` argument. The thumbprint is returned by the server in the error message if you attempt to run a command without specifying a thumbprint or certificate file.
`--url <url>` `VI_URL=<url>`	Connects to the specified vSphere Web Services SDK URL.
`--username <u_name>` `VI_USERNAME=<u_name>`	Uses the specified user name. ■ If `--server` specifies a vCenter Server system, the user name and password apply to that server. If you can log in to the vCenter Server system, you need no additional authentication to run commands on the ESXi hosts that server manages. ■ If `--server` specifies an ESXi system, the user name and password apply to that system. If you do not specify a user name and password on the command line, the system prompts you and does not echo your input to the screen.
`--vihost <host>` `-h <host>`	When you run a vCLI command with the `--server` option pointing to a vCenter Server system, use `--vihost` to specify the ESXi host to run the command against. NOTE: This option is not supported for each command. If supported, the option is included when you run `<cmd> --help`.

Hello Host: Running Your First Script

Before you run your first script, you need the following:

- Successful vSphere SDK for Perl installation. See the *vSphere SDK for Perl Installation Guide* for information.
- Access to one of the supported vSphere hosts. Perform a connection check using the process described in "Use the Managed Object Browser to Explore Server-Side Objects" on page 24.

To run the connect.pl script

1 At a command prompt, change to the `/apps/general` directory.

```
C:\Program Files\VMware\VMware vSphere CLI\Perl\apps\general
```

```
/usr/lib/vmware-vcli/apps
```

2 Run `connect.pl` as follows:

```
connect.pl --url https://<host>:<port>/sdk/vimService --username myuser --password mypassword
```

The script returns an information message and the host time.

You are now ready to run other scripts, or create new scripts.

NOTE You can run any utility application with `--help` to display information about its parameters.

Writing vSphere SDK for Perl Scripts 2

This chapter uses a simple example script to illustrate how to write a vSphere SDK for Perl script. The chapter also explores the basics of the vSphere API object model.

NOTE This chapter does not discuss Perl basics. You are expected to know Perl and to understand its programming conventions. When you develop a vSphere SDK for Perl script, follow Perl standards for filenames, imports, and general processing flow. Use the appropriate filename extension for the type of script or application you are creating (`.pl` on Windows and `.pl` or no suffix on UNIX-like systems).

The chapter includes these topics:

Basic vSphere SDK for Perl Script

vSphere SDK for Perl scripts retrieve objects, such as virtual machines, from the server and work with these objects. vSphere SDK for Perl scripts follow the basic pattern shown in Table 2-1.

IMPORTANT The sample script does not use filters or property filters for efficiency. See "Refining vSphere SDK for Perl Scripts" on page 33 for information about those topics.

Table 2-1. Basic vSphere SDK for Perl Script (`simpleclient.pl`)

Code element	Discussed in

`#!/usr/bin/perl`
`use strict;`
`use warnings;`
`use VMware::VIRuntime;`
	"Step 1: Import the vSphere SDK for Perl Modules" on page 20.

Table 2-1. Basic vSphere SDK for Perl Script (`simpleclient.pl`) (Continued)

Code element	Discussed in

`my %opts = (`
`  entity => {`
`  type => "=s",`
`  variable => "VI_ENTITY",`
`  help => "ManagedEntity type: HostSystem, etc",`
`  required => 1,`
`  },`
`);`
 	
`Opts::add_options(%opts);`
`Opts::parse();`
`Opts::validate();`
	"Step 2: (Optional) Define Script-Specific Command-Line Options" on page 20.

`Util::connect();`	"Step 3: Connect to the Server" on page 22.

`# Obtain all inventory objects of the specified type`
`my $entity_type = Opts::get_option('entity');`
`my $entity_views = Vim::find_entity_views(`
`  view_type => $entity_type);`
	"Step 4: Obtain View Objects of Server-Side Managed Objects" on page 22.

`# Process the findings and output to the console`
 	
`foreach my $entity_view (@$entity_views) {`
`  my $entity_name = $entity_view->name;`
`  Util::trace(0, "Found $entity_type:`
`  $entity_name\n");`
`}`
	"Step 5: Process Views and Report Results" on page 22.

`# Disconnect from the server` `Util::disconnect();`	"Step 6: Close the Server Connection" on page 23.

Step 1: Import the vSphere SDK for Perl Modules

All vSphere SDK for Perl scripts must use the `VMware::VIRuntime` module:

```
use VMware::VIRuntime;
```

This module handles all client-side infrastructure details. For example, it transparently maps data types and provides local Perl interfaces to server-side objects. The module also loads subroutines that you can use to connect to a vCenter Server or ESX/ESXi system and to retrieve views. Views are the client-side Perl objects that encapsulate the properties and operations of server-side managed objects. The subroutines are organized into different packages:

- The `Opts` package subroutines handle built-in options and creating custom options.
- The `Util` package subroutines facilitate routine tasks, such as setting up and closing connections to the server.
- The `Vim` package subroutines access server-side managed objects, instantiate local proxy objects (views), update properties, and run local methods that result in operations on remote servers.

See "vSphere SDK for Perl Subroutine Reference" on page 41.

Step 2: (Optional) Define Script-Specific Command-Line Options

When you run a script from the command line, you usually specify connection information and might also specify other information such as a virtual machine that you want to power off or a host for which you need status information. vSphere SDK for Perl lets you specify these options in a variety of ways. See "Specifying Options" on page 12.

A number of common command-line options, most of them connection options, are already defined for all utility applications (see Table 1-3, "Options Available for All SDK for Perl Commands," on page 16). In addition, most applications have application-specific options you pass to the script at execution time.

The vSphere SDK for Perl has defined all common options using attributes and subroutines specified in the `VILib::Opts` package. You can similarly use the `VILib::Opts` package to create custom options for your own applications and scripts, to simplify use of your script, or to allow users to specify other information.

Example 2-1 defines an `entity` option that must be made available to the script at runtime. The option specifies which of the available entity types is passed as a parameter to the `Vim::find_entity_views()` subroutine for further processing. Any direct or indirect subclass of `ManagedEntity` is a valid option (for example `HostSystem`, `ResourcePool`, or `VirtualMachine`). The example creates and parses a new command-line option.

1 The example declares the option as a hash. The hash key is the option name, and the value is a hashref containing `Getopt::Long`-style option attributes. See Table 2-2 for attribute details.

Example 2-1 creates a required command-line option that accepts a string value, as follows:

```
my %opts = (
        entity => {
        type => "=s",
        variable => "VI_ENTITY",
        help => "ManagedEntity type: HostSystem, etc",
        required => 1,
    },
);
```

Table 2-2 lists all attributes you can use to define command-line options. The code fragment in "Step 1: Import the vSphere SDK for Perl Modules" on page 20 above uses only `type`, `variable`, `help`, and `required`. For related information, see the documentation for the `Getopt::Long` module.

Table 2-2. Attributes for Defining New Options

Attribute	Description
default	Default value used if the option is not explicitly set. An unset option with no default returns `undef` to the calling script.
func	Enables creating derived options. Set `func` to an external code reference to run the code when the SDK queries the value of the option.
help	Descriptive text explaining the option, displayed in the script's help message.
required	If this attribute is set to 1, users must provide a value for this option or the script exits and display the help message. Set to 1 to require a value. Set to 0 if the value is optional.
variable	Allows you to specify the option in an environment variable or a configuration file. See "Specifying Options" on page 12.
type	Uses Perl `Getopt`-style syntax to specify option type and whether the option is required or optional. Use double quotes to indicate that option doesn't accept a value. The default numeric value is 0. The default string value is " " (empty string). You can use one of the following: ■ Equal sign (=) — mandatory ■ Colon (:) — optional ■ s — string ■ i — integer ■ f — float

2 The example adds the option to the default options using the `Opts::add_options()` subroutine:

```
Opts::add_options(%opts);
```

3 The example parses and validates the options before connecting to the server, as follows:

```
Opts::parse();
Opts::validate();
```

In Example 2-1, the `entity` option is required, so the script cannot run unless the user passes in the option name and value (see "Specifying Options" on page 12).

The `Vim::find_entity_views()` subroutine uses the value the user passes in later in the script. The value must be one of the managed-entity types listed as `view_type` parameter supported by `Vim::find_entity_views()`.

NOTE Your script must call `Opts::parse()` and `Opts::validate()` to process the options available for all scripts, even if you do not define script-specific command-line options.

When you attempt to run a script and do not supply all necessary options, the vSphere SDK for Perl displays usage help for the script, as in the following example:

```
perl simpleclient.pl
Required command option 'entity' not specified
Common VI options:
. . .

Command-specific options:
--entity (required)
ManagedEntity type: ClusterComputeResource, ComputeResource, Datacenter,
Folder, HostSystem, ResourcePool, VirtualMachine...
```

Step 3: Connect to the Server

The vSphere API is hosted as a secure Web service on a vCenter Server and ESX/ESXi system. By default, the Web service is available over HTTPS. Clients must provide valid credentials to connect to the service. Depending on the specifics of your server, you might have to enter only a user name and password. You might need other information. See Table 1-3, "Options Available for All SDK for Perl Commands," on page 16.

A call to `Util::connect()` connects to the server.

When a script reaches the call to `Util::connect()`, the vSphere SDK for Perl runtime checks the environment variables, configuration file contents, and command-line entries (in this order) for connection options. If the options are not defined, the runtime uses the defaults (`localhost` and no user name and password) to set up the connection.

Step 4: Obtain View Objects of Server-Side Managed Objects

When you call the subroutines in the `Vim` package to retrieve entities from the host, the vSphere SDK for Perl runtime creates the corresponding Perl objects (view objects) locally.

Example 2-1 uses the `Opts::get_option()` subroutine to assign to `$entity_type` the string value of the parameter that the user passes in when executing the script. Example 2-1 then uses `$entity_type` as the `view_type` parameter in the subsequent call to `Vim::find_entity_views()`.

```
# get all inventory objects of the specified type
my $entity_type = Opts::get_option('entity');
my $entity_views = Vim::find_entity_views(view_type => $entity_type);
```

The `Vim::find_entity_views()` subroutine creates a local Perl object (an array of references) from the server-side managed object of the specified entity type.

IMPORTANT This object is static and must be explicitly updated when the corresponding server-side object changes.

Step 5: Process Views and Report Results

The last part of the script processes the views. For this step, you must know the view objects' properties and methods, so you must understand the server-side objects. See "Understanding Server-Side Objects" on page 24 for an introduction. For in-depth information about server-side objects, see the *vSphere API Reference Guide* which is included on the vSphere SDK for Perl documentation page.

Because views are Perl objects, you use Perl object-oriented syntax to process the views. Example 2-1 loops through the array of entities returned (`@$entity_views`) and accesses the `name` property of each entity by calling `$entity_view->name`. The example then prints the name of each entity to the console.

See "Understanding Server-Side Objects" on page 24.

Step 6: Close the Server Connection

To log out and exit, use the `Util::disconnect()` subroutine. Example 2-1 shows the complete listing for `simpleclient.pl`.

Example 2-1. Sample Script (Commented Version)

```
#!/usr/bin/perl

# The simpleclient.pl script outputs a list of all the entities of the specified managed-entity
# type (ClusterComputeResource, ComputeResource, Datacenter, Datastore, Folder, HostSystem,
# Network, ResourcePool, VirtualMachine, or VirtualService) found on the target vCenter Server or
# ESX system. Script users must provide logon credentials and the managed entity type. The script
# leverages the Util::trace() subroutine to display the found entities of the specified type.

use strict;
use warnings;
use VMware::VIRuntime;

# Defining attributes for a required option named 'entity' that
# accepts a string.
#
my %opts = (
        entity => {
        type => "=s",
        variable => "VI_ENTITY",
        help => "ManagedEntity type: HostSystem, etc",
        required => 1,
    },
);
Opts::add_options(%opts);

# Parse all connection options (both built-in and custom), and then
# connect to the server
Opts::parse();
Opts::validate();
Util::connect();

# Obtain all inventory objects of the specified type
my $entity_type = Opts::get_option('entity');
my $entity_views = Vim::find_entity_views(view_type => $entity_type);

# Process the findings and output to the console

foreach my $entity_view (@$entity_views) {
    my $entity_name = $entity_view->name;
    Util::trace(0, "Found $entity_type: $entity_name\n");
}

# Disconnect from the server
Util::disconnect();
```

To run the simpleclient.pl script

1 Open a command prompt or console.

2 Change to the directory that contains the `simpleclient.pl` script.

3 Run the script using the following syntax:

```
perl simpleclient.pl <conn_params> --entity <EntityType>
```

For example:

```
perl simpleclient.pl --server aquarium.mycomp.com --username abalone --password tank --entity
               HostSystem
```

```
Found HostSystem: abcd-42.shellfish.vmware.com
```

Understanding Server-Side Objects

When you run a vSphere SDK for Perl script, your goal is to access and potentially analyze or modify server-side objects. You need the name of the vSphere API objects and often their properties and method names. For example, if you want to power off a virtual machine, you must know how to find the corresponding object, what the name of the power off method is, and how to run that method.

The *vSphere API Reference Guide* gives reference documentation for all vSphere API objects. Some users might also find the *vSphere Web Services SDK Programmer's Guide* helpful for understanding how the vSphere API objects interact. The guides are available from the VMware APIs and SDKs Documentation page.

This section first introduces the Managed Object Browser (MOB), which allows you to browse all objects on a remote host. The rest of the section discusses how to work with these server-side objects. You learn how to find the objects, access and modify properties, and how to run a method on the server.

Use the Managed Object Browser to Explore Server-Side Objects

The MOB is a Web-based server application hosted on all ESX/ESXi and vCenter Server systems. The MOB lets you explore the objects on the system and obtain information about available properties and methods. It is a useful tool for investigating server-side objects and for learning about the vSphere object model.

In ESXi 6.0 and later, the MOB is disabled by default on ESXi.

To enable the MOB on ESXi 6.0 and later systems

1 Select the host in the vSphere Web Client and go to Advanced System Settings.

2 Find **Config.HostAgent.plugins.solo.enableMob** and enable the MOB.

While a password is required to access the MOB, consider the security implications of enabling it.

To access the MOB on any ESXi or vCenter Server system

1 Start a Web browser.

2 Connect to the MOB using the fully-qualified domain name (or the IP address) of the ESX/ESXi or vCenter Server system, as follows:

```
https://<hostname.yourcompany.com>/mob
```

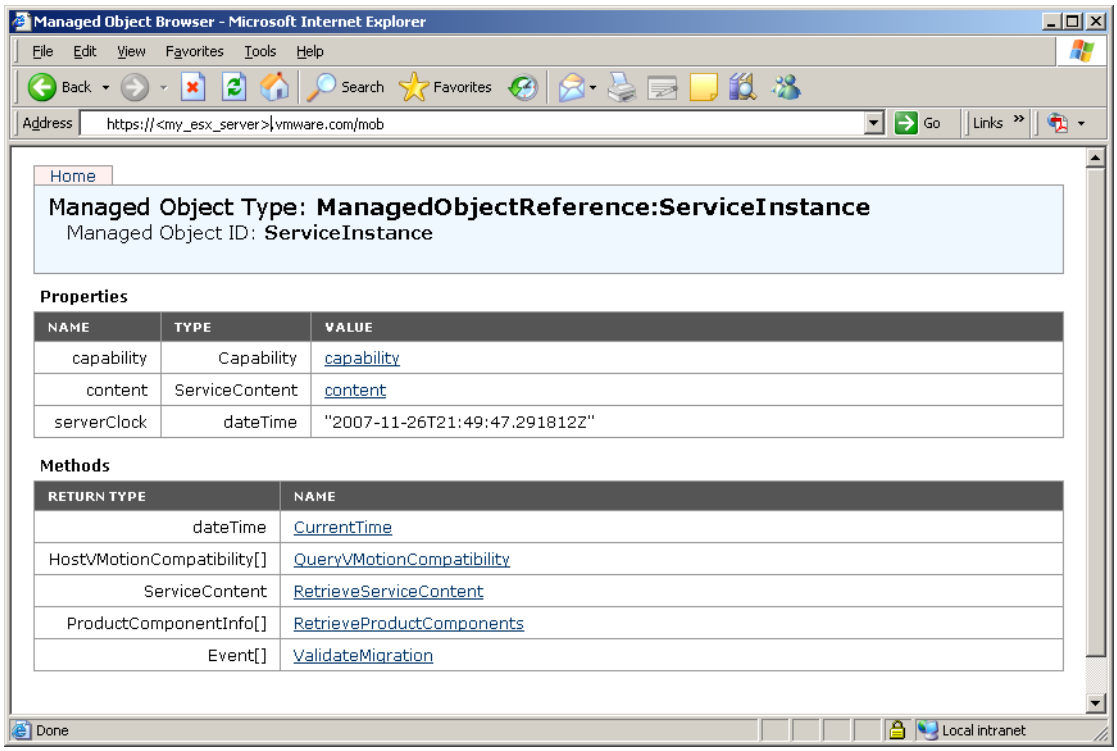
The browser prompts you for a user name and password for the host.

3 Enter the user name and password.

After you enter the user name and password, the host might display warning messages regarding the SSL certificate authority, such as `Website Certified by an Unknown Authority`. If VMware is the certificate authority, you can disregard such warnings and continue to log in to the MOB.

When you are successfully connected to the MOB, the browser displays the managed object reference for the service (`ManagedObjectReference:ServiceInstance`), available properties (with values), and methods, as shown in Figure 2-1.

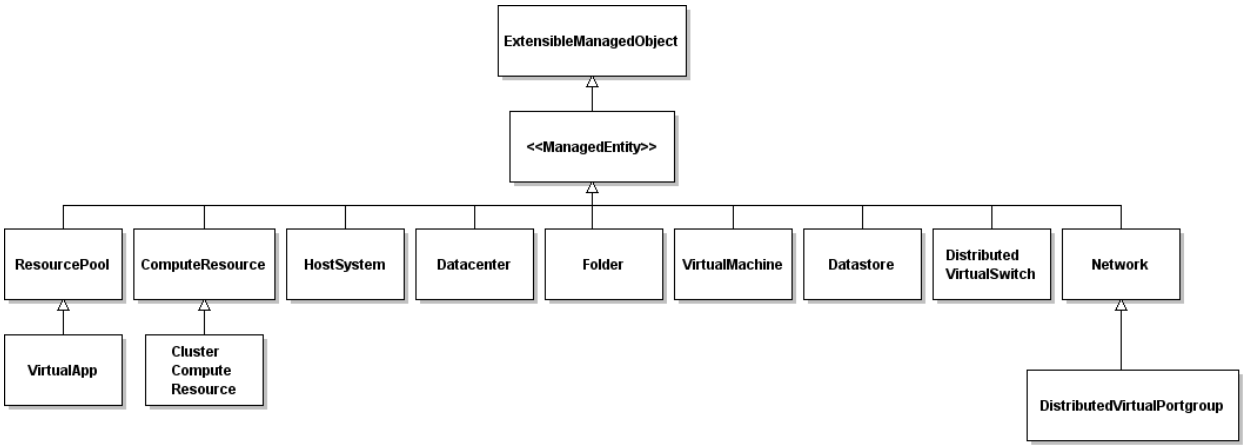
Figure 2-1. Managed Object Browser



Types of Managed Objects and the Managed Object Hierarchy

A *managed object* is the primary type of object in the vSphere object model. A managed object is a data type available on the server that consists of properties and operations. Each managed object has properties and provides various services (operations or methods). Figure 2-2 shows the `ExtensibleManagedObject` hierarchy as an example. See "Managed Entities in the Inventory" on page 26.

Figure 2-2. ExtensibleManagedObject Hierarchy



Managed objects define the entities in the inventory and also common administrative and management services such as managing performance (`PerformanceManager`), finding entities that exist in the inventory (`SearchIndex`), disseminating and controlling licenses (`LicenseManager`), and configuring alarms to respond to certain events (`AlarmManager`). See the *vSphere API Reference*.

A managed object reference (represented by a `ManagedObjectReference`) identifies a specific managed object on the server, encapsulates the state and methods of that server-side object, and makes the state and methods available to client applications. Clients run methods (operations) on the server by passing the appropriate managed object reference to the server as part of the method invocation.

Managed Object Hierarchy

The `ServiceContent` server-side object provides access to all other server-side objects. Each property of the `ServiceContent` object is a reference to a specific managed object. You must know those property names to access the other objects. You can use the MOB (see "Use the Managed Object Browser to Explore Server-Side Objects" on page 24) or use the API Reference documentation.

The *vSphere API Reference Guide* contains definitions of all server-side objects and their properties and methods. You can therefore use the *vSphere API Reference Guid*e to identify the list of parameters and operations that you can use with specific vSphere SDK for Perl views that you create and manipulate in your code.

To view documentation for server-side objects

1 Find the *vSphere API Reference Guide*, available from the VMware APIs and SDKs Documentation page.

2 Click **All Types** to see a list of all managed object types.

3 Find the `ServiceContent` object.

`ServiceContent` provides access services, such as `PerformanceManager`, and to inventory objects, which allow you to access the entities in the virtual datacenter such as hosts (`HostSystem`) and virtual machines (`VirtualMachine`). `ServiceContent` properties also allow access to other managed objects, for example:

- The `rootFolder` property is a `ManagedObjectReference` to a `Folder` managed object type.
- The `perfManager` property is a `ManagedObjectReference` to a specific instance of a `PerformanceManager` managed object type, and so on.

The vSphere Client displays the hierarchy of inventory objects. The vSphere Client uses the information about the objects (the properties and the relationships among them) for the display. For information about the vSphere Client and how to work with its display, see the documents in the vSphere online library

Managed Entities in the Inventory

The inventory consists of the managed entities on the server. A managed entity is a managed object that extends the `ManagedEntity` managed object type. `ManagedEntity` is an abstract class that defines the base properties and operations for vSphere managed objects such as datacenters and hosts. See Figure 2-2 for an overview. The following managed object types extend the `ManagedEntity` superclass:

- `Datacenter` – Contains other managed entities, including folders, virtual machines, and host systems. A vCenter Server instance can support multiple datacenters, but an ESX/ESXi host supports only one datacenter.
- `Datastore` – Represents logical storage volumes on which to store virtual machine files and other data.
- `Distributed Virtual Switch` – Interface for the VMware distributed virtual switch (DVS).
- `Folder` – Contains references to other entities, for example, other folders (`Folder`) or hosts (`HostSystem`).
- `HostSystem` – Provides access to a virtualization host platform.
- `Network` – Abstraction for a physical or virtual network (VLAN).
- `VirtualMachine` – Represents a single virtual machine.
- `ResourcePool` – Allows you to combine CPU and memory resources from multiple hosts and to establish rules for dividing those resources among all virtual machines associated with these hosts.
- `ClusterComputeResource` – Represents a cluster of `HostSystem` objects. Administrators create clusters to combine the CPU and memory resources of hosts and to set up VMware HA or VMware DRS for those clusters. See the *Resource Management Guide*, which is part of the vSphere documentation set.
- `ComputeResource` – Abstracts a host system's physical resources and allows you to associate those resources with the virtual machines that run on the host.
- `VirtualService` – Container for one or more virtual machines an associated object package using open virtual format (OVF).

Managed entities offer specific operations that vary depending on the entity type. For example, a `VirtualMachine` managed entity provides operations for creating, monitoring, and controlling virtual machines. You can power a virtual machine on or off (`PowerOnVM`, `PowerOffVM`) and you can capture state (`Snapshot`). A `HostSystem` entity provides operations for entering and exiting maintenance mode (`EnterMaintenanceMode_Task`, `ExitMaintenanceMode_Task`) and for rebooting the server (`RebootHost_Task`).

The `ManagedEntity` base class includes several properties that are inherited by each subclass, such as a `name` property, whose data type is a string. `ManagedEntity` also includes a few operations that are inherited by each subclass (`Destroy_Task`, and `Reload`, for example). `VirtualMachine` and `HostSystem` extend the `ManagedEntity` class, so each subclass has a `name` property inherited from `ManagedEntity`.

Accessing Server-Side Inventory Objects

The vSphere SDK for Perl provides subroutines for accessing server-side inventory objects and other managed objects that provide functionality to the server as a whole.

Example 2-1 obtains all entities of a specific type from the inventory. The entity type is passed as a parameter to the `Vim::find_entity_views()` subroutine, which returns an array of references to view objects that map to the corresponding server-side entities.

Example 2-2 starts at the level of the entire service and uses the `Vim::get_service_content()` subroutine to obtain an instance of the `ServiceContent` object:

```
my $content = Vim::get_service_content();
```

You can use the `ServiceContent` object to retrieve a local view of the services provided by the server, as in this example:

```
my $diagMgr = Vim::get_view(mo_ref => $content->diagnosticManager);
```

Example 2-2 shows how these two calls form the basis of a script that follows changes in the log file, which can be accessed as the `logfile` property of the `diagnosticManager`.

Example 2-2. Following Changes in a Log File

```
#!/usr/bin/perl
#
# Copyright 2007 VMware, Inc. All rights reserved.
#
# This script creates a Perl object reference to the ServiceContent data
# object, and then creates a reference to the diagnosticManager. The script
# follows ('tails') the log as it changes.
use strict;
use warnings;

use VMware::VIRuntime;

# read/validate options and connect to the server
Opts::parse();
Opts::validate();
Util::connect();

# get ServiceContent
my $content = Vim::get_service_content();
my $diagMgr = Vim::get_view(mo_ref => $content->diagnosticManager);
# Obtain the last line of the logfile by setting an arbitrarily large
# line number as the starting point
my $log = $diagMgr->BrowseDiagnosticLog(
   key => "hostd",
   start => "999999999");
my $lineEnd = $log->lineEnd;

# Get the last 5 lines of the log first, and then check every 2 seconds
# to see if the log size has increased.
my $start = $lineEnd - 5;
```

```
# Disconnect on receipt of an interrupt signal while in the infinite loop below.
$SIG{INT} = sub { Util::disconnect();
exit;
};

while (1) {
 $log = $diagMgr->BrowseDiagnosticLog(
   key => "hostd",
   start => $start);
 if ($log->lineStart != 0) {
  foreach my $line (@{$log->lineText}) {
  # next if ($line =~ /verbose\]/);
  print "$line\n";
  }
 }
 $start = $log->lineEnd + 1;
 sleep 2;
}
```

Understanding Perl View Objects

A view is a client-side Perl object populated with the state of one or more server-side managed objects by the vSphere SDK for Perl. A view object has the following characteristics:

- Is a static copy of a server-side managed object and includes properties and methods that correspond to the properties and operations of the server-side managed object.
- Must be explicitly updated when the object on the server changes. See "Updating View Objects" on page 32.
- Has properties that correspond to properties of server-side managed objects as follows:
  - For each simple property (string, Boolean, numeric data type), including inherited simple properties, the SDK creates an accessor method. The accessor method name is the same as the property name.
  - Arrays of properties become arrays of properties of the same name.
- Includes these methods:
  - An accessor method for each managed object property. The vSphere SDK for Perl provides accessors for any property, regardless of its depth inside a composite object structure.
  - A blocking and a non-blocking method for each (non-blocking) operation provided by the server-side managed object. See "Non-Blocking and Blocking Methods" on page 30.
  - A method that updates the state of any client-side view object with current data from the server. See "Updating View Objects" on page 32.

The vSphere SDK for Perl simplifies programming as follows:

- Provides a `_this` parameter to reference the object on which a method is run, if needed.
- Allows you to pass a view object as a parameter to methods that take managed object references as required parameter. The SDK converts the view object to the corresponding managed object.

Working with View Object Property Values

vSphere SDK for Perl view objects are Perl objects. You can retrieve a view, access and manipulate its properties, and call its methods using Perl's object-oriented syntax.

Accessing Property Values

Each property is defined as a specific data type and can be one of the following:

Table 2-3. Property Overview

Property	Example
Simple data type, such as a string, Boolean, numeric, or dateTime.	The `ManagedEntity` managed object has a `name` property of type string.
Array of simple data types or data objects.	A `HostSystem` managed object contains an array of virtual machines that are hosted by the corresponding physical machine.
Enumeration (enum) of predefined values. The values can be a collection of simple data types or data objects.	A virtual machine's power state can be a `poweredOn`, `poweredOff`, or `suspended` string value.
Complex data type called data object (part of the vSphere object model).	`AboutInfo`, `Action`, and `ServiceContent` are all data objects.

Accessing Simple Property Values

To access a simple property from a view, call the property's accessor on the view object. The accessor has the same name as the property itself, as follows:

```
$view_name->property_name
```

As shown in Example 2-1, you can access the `name` property of `entity_view` by calling its `name` method, as follows:

```
my $entity_name = $entity_view->name;
```

Accessing Enumeration Property Values

To retrieve the value of a property defined as an enumeration, you must dereference its value from within the containing object by qualifying the property with `->val`. For example, the power state of a virtual machine (`powerState`) is a member of the `runtime` data object.

To retrieve the value of `powerState`, you must dereference the two containing objects (the `view` object and the `runtime` data object) and the value itself (`val`), as follows:

```
$vm_view->runtime->powerState->val
```

Because `powerState` is an enumeration, you use `runtime->powerState->val` to retrieve its string value.

```
foreach my $vm (@$vm_views) {
    if ($vm->runtime->powerState->val eq 'poweredOn') {
        print "Virtual machine " . $vm->name . " is powered on.\n";
        }
    else {
        print "Virtual machine " . $vm->name . " is not powered on.\n";
}
```

Modifying Property Values

You can modify a data object's property value by passing the new value, as follows:

```
$data_object-> <property> (<new value>);
```

`$data_object` is a blessed reference to a Perl object or class name, and `property` is a method call on the object.

For example, you can change the `force` property to `false`, as follows:

```
$host_connect_spec->force ('false');
```

To create an enumeration data object, use a string value as the argument to the enumeration type's constructor.

```
my $power_state = VirtualMachinePowerState->new('poweredOff');
```

Typically, enumerations are used as arguments to methods:

```
$vm->MigrateVM(
  host => $target_host,
  priority => VirtualMachineMovePriority->new('defaultPriority'),
  state => VirtualMachinePowerState->new('poweredOff'),
```

```
);
```

Creating Data Objects with Properties

You create data objects with constructors that have names corresponding to the classes of the data objects in the vSphere API. The constructor syntax follows common Perl conventions. The arguments supplied to the constructor are key-value pairs, where each key is the name of an object property, and the corresponding value is the value with which the property is initialized.

For example, creating a virtual machine requires the creation of a data structure that includes a number of nested data objects. One of those objects is a `VirtualMachineFieldInfo` data object, which can be constructed as follows:

```
my $files = VirtualMachineFileInfo->new
(
    logDirectory => undef,
    snapshotDirectory => undef,
    suspendDirectory => undef,
    vmPathName => $ds_path
);
```

The `VirtualMachineFileInfo` object is then contained within a `VirtualMachineConfigSpec` object:

```
my $vm_config_spec = VirtualMachineConfigSpec->new(
    name => $args{vmname},
    memoryMB => $args{memory},
    files => $files,           # <-- here
    numCPUs => $args{num_cpus},
    guestId => $args{guestid},
    deviceChange => \@vm_devices
);
```

This code is taken from the `apps/vm/vmcreate.pl` utility application. See the scripts in the `apps` and `samples` directories for examples of simple and complex uses of data objects.

To set the value of a property that is defined as an enumeration, you must pass the new value to the data object as follows:

```
$<ref> = new <enum_type> ('<val>');
```

For example, you can change the power state as follows:

```
$power_state = new VirtualMachinePowerState ('poweredOff');
```

Understanding Operations and Methods

The vSphere SDK for Perl runtime maps server-side operations to client-side Perl view object methods. For each operation defined on a `server` managed object, the vSphere SDK for Perl creates a corresponding view method when it creates the view object.

Non-Blocking and Blocking Methods

All server-side operations available in the vSphere API are non-blocking operations listed in the *vSphere API Reference Guide*. The vSphere SDK for Perl provides a non-blocking method corresponding to the server-side operation, and also provides a blocking (synchronous) method (`<opname>()` method).

- **Non-blocking methods** – Asynchronous methods that return control to the client immediately after invocation and return a task object to the calling program. Non-blocking methods allow you to monitor progress (of the `*_Task` object) outside the main processing logic of the script. This monitoring can be useful during long-running operations. These methods also allow you to interleave local (client-side) processing and server-side processing.
- **Blocking methods** – Synchronous methods that fully process the operation before returning control to the client script. Does not return a reference to a task object.If you use a blocking method, you do not have to handle a task object with additional code.

Examples of Operations

The following table lists some of the operations available for a `VirtualMachine` managed object.

Table 2-4. Examples for Asynchronous and Synchronous Methods

vSphere API Non-blocking (asynchronous)	vSphere SDK for Perl Non-blocking (asynchronous)	vSphere SDK for Perl Only Blocking (synchronous)
`PowerOnVM_Task()`	`PowerOnVM_Task()`	`PowerOnVM()`
`CloneVM_Task()`	`CloneVM_Task()`	`CloneVM()`
`SuspendVM_Task()`	`SuspendVM_Task()`	`SuspendVM()`

See the *vSphere API Reference Guide* for lists of all operations for each managed object.

Calling Methods

After you have retrieved the view object that corresponds to a managed object, you can run methods on that view to make use of the managed object's services. You run a method by specifying the method's `name` parameter, for example:

```
$vm->MigrateVM (name => 'productionVM');
```

The type of parameter required by the method depends on the operation defined in the vSphere API. It might be a simple type, data object, or managed object reference. For information about specific parameters and data types, see the *vSphere API Reference Guide*.

Blocking operations are run as methods on a view object. For example, to suspend a virtual machine, call:

```
$vm_view->SuspendVM();
```

You can execute any operation that is defined for a managed object as a method on a corresponding view object. Because the vSphere SDK for Perl creates an accessor and a mutator method (getter and setter method) for each property defined in the managed object, you can reference the name of any property as a method call of the view, for example:

```
my $network_name = $network_view->name
```

The vSphere SDK for Perl allows you to pass a view object to a method that requires a `ManagedObjectReference`. For example, if you have the view that represents a host (`$host`), you can pass the view to the `powerOn()` method as follows:

```
my $host = Vim::find_entity_view (view_type => 'HostSystem', name => 'my host');
my $vm = Vim::find_entity_view (view_type => 'VirtualMachine', name => 'my virtual machine');
$vm->powerOn (host => $host)
```

NOTE "Specifying Untyped Arguments in Scheduled Tasks and Callbacks" on page 39 discusses using the vSphere SDK for Perl `PrimType` structure in some calls.

Omitting Optional Arguments in Method Calls

When you call a vSphere API method using vSphere SDK for Perl, and want to omit an optional argument, you can do one of two things:

- You can omit the argument:

  ```
  $vm->PowerOnVM(host => $host); # with the optional host argument
  $vm->PowerOnVM(); # without the optional host argument
  ```

- You can supply `undef` as the value of the optional argument:

  ```
  $vm->PowerOnVM(host => undef);
  ```

Supplying `undef` as the value of the optional argument is useful when the value of an argument, which might or might not be `undef`, is contained in a variable, as in the following example:

```
my $host = Vim::find_entity_view(
```

```
  view_type => 'HostSystem',
  filter => { name => 'preferredHost' }
);
$vm->PowerOnVM(host => $host);
```

You cannot use the empty string or the value 0 to represent `undef` or an unset parameter.

Updating View Objects

In any view, the values of the view properties represent the state of the server-side objects at the time the view was created. These property values are not updated automatically. In a production environment, the state of managed objects on the server is likely to change frequently. If your client script depends on the server being in a particular state (`poweredOn` or `poweredOff`, for example), then you can refresh the view object's state. You can use the vSphere SDK for Perl `Vim::update_view_data()` subroutine to refresh the values of client-side views with server-side values. Example 2-3 uses `Vim::update_view_data()` to refresh view data.

Example 2-3. Updating the State of View Objects

```
#!/usr/bin/perl
use strict;
use warnings;
use VMware::VIRuntime;
. . .
# Get all VirtualMachine objects
my $vm_views = Vim::find_entity_views(view_type => 'VirtualMachine');

# Power off virtual machines.
foreach my $vm (@$vm_views) {
    # Refresh the state of each view
    $vm->update_view_data();
    if ($vm->runtime->powerState->val eq 'poweredOn') {
        $vm->PowerOffVM();
        print " Stopped virtual machine: " . $vm->name . "\n";
    } else {
        print " Virtual machine " . $vm->name .
    " power state is: " . $vm->runtime->powerState->val . "\n";
    }
}
```

Refining vSphere SDK for Perl Scripts 3

This chapter discusses some programming techniques for your vSphere SDK for Perl scripts.

The chapter includes these topics:

Creating and Using Filters

You can use the vSphere SDK for Perl to define and use filters that select objects based on property values. Filters can reduce a large result set to only those objects with characteristics of interest to you.

Using Filters with Vim::find_entity_view() or Vim::find_entity_views()

You can call `Vim::find_entity_view()` or `Vim::find_entity_views()` to retrieve objects from the ESX/ESXi host. `Vim::find_entity_view()` returns the first object it finds that matches the search criteria. `Vim::find_entity_views()` returns all objects.

When you call `Vim::find_entity_view()` the first object found might not be the one you are looking for. For example, you might want to retrieve only those virtual machine objects whose names begin with a certain prefix. When you call `Vim::find_entity_views()`, the command might return more objects than you want to work with, for example all virtual machines in a datacenter. You can apply one or more filters to `Vim::find_entity_view()` and `Vim::find_entity_views()` to select a subset of objects based on property values.

To apply a filter to the results of `Vim::find_entity_view()` or `Vim::find_entity_views()`, you supply an optional `filter` parameter. The value of the parameter is an anonymous hash reference containing one or more pairs of filter criteria. Each of the criteria is a property path and a match value. The match value can be either a string or a regular expression object. If the match value is a string, the value of the property must match the string exactly (including case). To match Boolean values, use the strings `true` and `false`.

The following filter parameter matches a virtual machine power state of `poweredOff`:

```
filter => { 'runtime.powerState' => 'poweredOff' }
```

You can also match using a regular expression object, generally known as a `qr//` (quoted regular expression) object. In this case, the value of the property must match the regular expression.

The following filter matches objects whose names begin with `Test`:

```
filter => { 'name' => qr/^Test/ }
filter => { 'name' => qr/^test/i } # make the match case-insensitive with the i option
```

For more information about the `qr//` operator, see the `perlre` (perl regular expressions) and `perlop` man pages in the standard Perl documentation.

The following example illustrates how you might use `Vim::find_entity_views()` in combination with a filter. It prints a list of virtual machine objects whose guest operating system names contain the string `Windows`.

Example 3-1. Filter that Creates Views of Windows-Based Virtual Machines Only

```
. . .
my $vm_views = Vim::find_entity_views(
    view_type => 'VirtualMachine',
    filter => {
        # True if string 'Windows' appears anywhere in guestFullName
        'config.guestFullName' => qr/Windows/
    }
);
# Print VM names
foreach my $vm (@$vm_views) {
    print "Name: " . $vm->name . "\n";
}
. . .
```

If you pass multiple filter criteria to `Vim::find_entity_view()` or `Vim::find_entity_views()`, the method returns only the managed objects for which all criteria match. The `filter` parameter specified in Example 3-2 includes two criteria. The example returns only virtual machines that fulfill both requirements:

- Guest operating system is Windows — the `config` property's `guestFullName` property includes the string `Windows`.
- Virtual machine is running. The power state is `poweredOn`.

Example 3-2. Example of Multiple Filter Specification

```
. . .
my $vm_views = Vim::find_entity_views(
    view_type => 'VirtualMachine',
    filter => {
        'config.guestFullName' => qr/Windows/,
        'runtime.powerState' => 'poweredOn'
    }
);
# Print VM names
foreach my $vm (@$vm_views) {
    print "Name: " . $vm->name . "\n";
}
. . .
```

IMPORTANT You can match only properties that have simple types like strings and numbers. Specifying a property with a complex type as an argument to a filter results in a fatal runtime error. For example, you cannot specify the `runtime` property of a `VirtualMachine` object, which is a complex object, not a string.

Filtering Views Selectively Using Properties

Each Perl view object has properties that correspond to properties of server-side managed objects as follows:

- For each simple property (string, Boolean, numeric data type), including inherited simple properties, the SDK creates an accessor method. The accessor method name is the same as the property name.
- Arrays of properties become arrays of properties of the same name.

Because many of the server-side managed objects have a large number of properties, accessing only a small number of objects can potentially result in noticeable performance degradation. You use a properties filter to populate the view object only with properties you are interested in to avoid that problem.

Using View Subroutines with a Properties Argument

The view subroutines—`get_view()`, `get_views()`, `find_entity_view()`, and `find_entity_views()`—can accept a `properties` argument that consists of a list of property paths for retrieval from the server. Go to the *vSphere Web Services SDK Reference* for a list of properties for each server-side managed object. Property paths can be full paths, and can include nested properties. Properties do not have to be top-level managed object properties.

The following example illustrates filtering by property.

1 Populate a virtual machine view with power-state information only, as follows:

```
my $vm_view = Vim::find_entity_view(
        view_type => 'VirtualMachine',
        filter => { 'name' => 'foo' },
        properties => [ 'runtime.powerState' ]
        );
```

2 Use the view object's `get_property()` method. Note that `$vm_view` is an array reference, not a scalar.

```
my $state = $vm_view->get_property('runtime.powerState');
```

3 If you are interested in subproperties of the retrieved properties, you can retrieve them like this:

```
my $vm_view = Vim::find_entity_view(
        view_type => 'VirtualMachine',
        filter => { 'name' => 'foo' },
        properties => [ 'config.hardware' ]);
my $memsize = $vm_view->get_property('config.hardware.memoryMB');
```

`get_property()` works with fully-populated views as well. The following code fragments uses `get_property` to retrieve a property from a virtual machine.

```
my $vm_view = Vim::find_entity_view(
            view_type => 'VirtualMachine',
            filter => { 'name' => 'foo' });
my $memsize = $vm_view->get_property('config.hardware.memoryMB');
```

The following code fragment, which retrieves the same property by traversing the tree, has the same result.

```
my $vm_view = Vim::find_entity_view(
            view_type => 'VirtualMachine',
            filter => { 'name' => 'foo' });
my $memsize = $vm_view->config->hardware->memoryMB;
```

When you use a filtered view and attempt to read a property that was not retrieved from the server, the result is the same as if the property were unset.

Using Filters on the Utility Application Command Line

When you run a utility application that takes arguments specifying names for virtual machines, host systems, and so on, you must supply the exact name on the command line. Regular expressions are not accepted.

When you run a utility application, there are some restrictions on special characters:

- In virtual machine names, you must represent the character forward slash (/) as `%2f`, backward slash (\) as `%5c`, and percent (%) as `%25` when they appear in virtual machine names.
- On UNIX-like command lines, surround names that contain special characters with single-quotes, and use percent (%) as the escape character.

For example, to search for the virtual machine `San Jose`, run this command:

```
perl vminfo.pl --username Administrator --password 'secret' --server myserver --vmname 'San Jose'
```

To search for the virtual machine `San-Jose/5`, run this command:

```
perl vminfo.pl --username Administrator --password 'secret' --server myserver --vmname
               'San-Jose%2f5'
```

Retrieving the ServiceInstance Object on a vSphere Host

You can retrieve the `ServiceInstance` object to access the `ServiceContent` or to retrieve the current time on an ESX/ESXi or vCenter Server system.

If you want to retrieve the current time, you must retrieve a `ServiceInstance` object and call its `CurrentTime()` method. You can use the `Vim::get_service_instance()` subroutine to retrieve the object.

To retrieve the current VMware host time

1 Connect to the VMware host:

```
Util::connect();
```

2 Retrieve the `ServiceInstance` object:

```
my $service_instance = Vim::get_service_instance();
```

3 Retrieve the current host time:

```
$service_instance->CurrentTime();
```

Saving and Using Sessions

The vSphere SDK for Perl library includes several subroutines that save and reuse sessions, so you can maintain sessions across scripts. Using sessions can enhance security: Instead of storing passwords in scripts, you can run the `Vim::login()` subroutine in your script using the name of the session file. The session file does not expose password information.

Saving Sessions

You can save a session using the subroutine call syntax as follows:

```
Vim::save_session():

...
# usual login procedure with connect()
Util::connect();
...
# save the global session in file .mysession
Vim::save_session(session_file => '.mysession');
```

Alternatively, you can use `save_session()` with the object-oriented syntax (see "Using Multiple Sessions" on page 37):

```
...
# object-oriented login
my $service_url = "https://$server/sdk/vimService";
my $vim = Vim->new(service_url => $service_url);
$vim->login(user_name => $username, password => $password); ...
# save session $vim in file .mysession
$vim->save_session(session_file => '.mysession');
```

The session remains active until the program runs a log out or disconnect operation, or until the program times out. Time out is 30 minutes after the last operation was performed.

Loading Sessions

You can use `load_session()` to load a saved session into the global session as follows:

```
Vim::load_session(session_file => '.visession');
```

Alternatively, you can load a session using the object-oriented syntax as follows:

```
my $service_url = "https://$server/sdk/vimService";
my $vim = Vim–>new(service_url => $service_url);
$vim = $vim–>load_session(session_file => '.visession');
```

Using Multiple Sessions

In some cases, you might want to create sessions on several vSphere servers at once, or create more than one session on the same server.

Each time an application connects to a server in the vSphere environment, a session between the application and the server is created. The vSphere SDK for Perl represents the session as a vSphere SDK for Perl object. When you use single sessions, one global object is implicit for the sessions.

For multiple objects, you cannot use the implicit global vSphere object. Instead, you must create and use vSphere objects explicitly, and use the object-oriented syntax for calling vSphere SDK for Perl methods.

You create an open session in two stages.

1 Create a `vSphere object` using the `new()` constructor.

2 Log in by calling the object-oriented `login()` method. The arguments to the object-oriented `login()` method are the same as for the procedural `Vim::login()` subroutine.

Most procedural `Vim::` methods have an object-oriented counterpart. The procedural methods operate on an implicitly specified global vSphere object. Object-oriented methods operate on the explicitly supplied vSphere object.

The following code fragment from `/samples/sessions/multisession.pl` illustrates how to use multiple sessions, using the object-oriented programming style in vSphere SDK for Perl.

Example 3-3. Using Multiple Sessions

```
use VMware::VIRuntime;
...
# create object for each host
my @vim_objs;
my $url;
$url = Opts::get_option('url');;
push @vim_objs, Vim–>new(service_url => $url);
$url = Opts::get_option('url2');
push @vim_objs, Vim–>new(service_url => $url);

# login to all hosts
my $username = Opts::get_option('username');
my $password = Opts::get_option('password');
$vim_objs[0]–>login(user_name => $username, password => $password);


if (Opts::option_is_set('username2')) {
   $username = Opts::get_option('username2');
}
if (Opts::option_is_set('password2')) {
   $password = Opts::get_option('password2');
}
$vim_objs[1]–>login(user_name => $username, password => $password);

# list VM's for all hosts
foreach my $vim_obj (@vim_objs) {
   print "List of virtual machines:\n";
   my $vm_views = $vim_obj–>find_entity_views(view_type => 'VirtualMachine');
   foreach my $vm (@$vm_views) {
      print $vm–>name . "\n";
   }
   print "\n";
```

```
}

# logout
foreach my $vim_obj (@vim_objs) {
   $vim_obj->logout();
}
```

Learning About Object Structure Using Data::Dumper

The vSphere SDK for Perl transparently uses the `Data::Dumper` Perl module (a standard library) to create the client-side view objects. Example 3-4 illustrates how you can use `Data::Dumper` to explore view objects and corresponding vSphere objects.

Lines 12 through 14 set several parameters of `Data::Dumper`, as follows:

- `Sortkeys` orders the name-value pairs alphabetically by name.
- `Deepcopy` enables deep copying of structures. Deep copying ensures that the output is straightforward and tree-like.
- `Indent` set to 2 causes `Data::Dumper` to take hash key length into account in the output. The indent results in a more readable format.

Example 3-4. Using Data::Dumper to Output Perl Object Structures

```
01 use strict;
02 use warnings;
03
04 use VMware::VIRuntime;
05 use VMware::VILib;
06
07 # Parse connection options and connect to the server

08 Opts::parse();
09 Opts::validate();
10 Util::connect();
11
12 $Data::Dumper::Sortkeys = 1; #Sort the keys in the output
13 $Data::Dumper::Deepcopy = 1; #Enable deep copies of structures
14 $Data::Dumper::Indent = 2; #Output in a reasonable style (but no array indexes)
15
16
17
18 # Get the view for the target host
19 my $host_view = Vim::find_entity_view(view_type => 'HostSystem');
20
21 print "The name of this host is ", $host_view->name . "\n\n";
22
23 print Dumper ($host_view->summary->config->product) . "\n\n\n";
24
25 print Dumper ($host_view->summary->config) . "\n\n\n";
26
27 print Dumper ($host_view->summary) . "\n\n\n";
28
29 # logout
30 Vim::logout();
```

When you run the entire program, it produces detailed output. The output from line 23 looks as follows:

```
$VAR1 = bless( {
    'apiType' => 'HostAgent',
    'apiVersion' => '4.0.0',
    'build' => '31178',
    'fullName' => 'VMware ESX Server 3.0.1 build-31178',
    'localeBuild' => '000',
```

```
    'localeVersion' => 'INTL',
    'name' => 'VMware ESX Server',
    'osType' => 'vmnix-x86',
    'productLineId' => 'esx',
    'vendor' => 'VMware, Inc.',
    'version' => '3.0.1'
    }, 'AboutInfo' );
```

The output above shows the content of the `summary.config.product` property of a `HostSystem` managed object. The type (or more properly class) of `summary.config.product` property is `AboutInfo`. Perl's `Data::Dumper` module writes the object in a form that can be used with `eval` to get back a copy of the original structure. The `bless` keyword indicates the data is a Perl object, and the last argument to `bless` is the class of the object, `AboutInfo.`

Line 19 (in Example 3-4) retrieves the `HostSystem` view object and line 21 prints the name associated with the corresponding host.

The `config` property has more values than those printed by line 23. Line 25 prints the entire `config` object. Inside the `config` object printed by line 25 (in Example 3-4), the `product` property is an object. The `bless` function returns a reference to the `product` object, which is itself nested inside the `config` object.

```
$VAR1 = bless( {
    'name' => 'test-system.eng.vmware.com',
    'port' => 'nnn',
    'product' => bless( {
        'apiType' => 'HostAgent',
        'apiVersion' => '4.0.0',
        'build' => '31178',
        'fullName' => 'VMware ESX Server 3.0.1 build-31178',
        'localeBuild' => '000',
        'localeVersion' => 'INTL',
        'name' => 'VMware ESX Server',
        'osType' => 'vmnix-x86',
        'productLineId' => 'esx',
        'vendor' => 'VMware, Inc.',
        'version' => '3.0.1'
        }, 'AboutInfo' ),
    'vmotionEnabled' => 'false'
    }, 'HostConfigSummary' );
```

The output from line 27 of Example 3-4 prints the structure of the entire `summary` object of the host view. The output shows a number of nested objects, including two objects that are nested two levels deep. The `product` object is nested inside the `config` object, and the `connectionState` object is nested inside the `runtime` object.

Specifying Untyped Arguments in Scheduled Tasks and Callbacks

Because of the way vSphere SDK for Perl maps the vSphere API into Perl, you have to specify arguments to callback methods differently from the way you specify arguments to other methods. You can use `PrimType` to specify untyped arguments in scheduled tasks and callbacks.

- You must specify the arguments positionally, in the order defined in the bindings for other languages like Java.
- You must indicate the type of each argument using the `PrimType` constructor.

For example, consider a scheduled task that periodically creates a snapshot. The `CreateSnapshot()` method takes four arguments, `name`, `description`, `memory`, and `quiesce`.

You must define the arguments before you use them by creating four `MethodActionArgument` objects with `PrimType` values, as follows:

```
my $name = MethodActionArgument->new(
  value => PrimType->new('Sample snapshot task', 'string')
);
my $description = MethodActionArgument->new(
  value => PrimType->new('Created from a sample script', 'string')
);
```

```
my $memory = MethodActionArgument->new(
  value => PrimType->new(0, 'boolean')
);
my $quiesce = MethodActionArgument->new(
  value => PrimType->new(0, 'boolean')
);
```

You use the `MethodActionArgument` objects in the order defined in the positional API, not with the usual `name => $value` syntax. You can then supply the four values defined above as arguments to `CreateSnapshot()`.

```
my $snapshot_action = MethodAction->new(
  name => "CreateSnapshot",
  argument => [
    $name,
    $description,
    $memory,
    $quiesce
  ]
);
```

The complete example is in `/samples/scheduled_task/vm_snapshot_schedule.pl` (Linux) and in `VMware vSphere CLI\Perl\samples\scheduled_task\vm_snapshot_schedule.pl` (Windows).

Using Advanced Subroutines

vSphere SDK for Perl includes one subroutine, `Opts::get_config()`.

Opts::get_config()

Determines whether a configuration file was read when vSphere SDK for Perl executed `Opts::parse()`.

This subroutine has no parameters.

Returns

If a configuration file was successfully opened, `Opts::get_config()` returns the path to it. If no configuration file was found, or if it could not be opened, `Opts::get_config()` returns `undef`.

vSphere SDK for Perl Subroutine Reference

4

The vSphere SDK for Perl are available in three packages:

- The `Opts` package includes subroutines for handling built-in options and creating custom options. See "Subroutines in the Opts Package" on page 42.
- The `Util` package includes subroutines for facilitating routine tasks such as setting up and closing connections to the server. See "Subroutines in the Util Package" on page 43.
- The `Vim` package includes subroutines for accessing server-side managed objects, instantiating local view objects, updating properties, and running local methods to run operations on remote servers.

Table 4-1. Subroutines in the Opts Package

Subroutine	Description
add_options	Enables custom options to be parsed and validated for execution in the context of the script to which the options have been added.
get_option	Retrieves the value of a specified built-in or custom option.
option_is_set	Checks whether an option has been explicitly set by a script or from the command line or whether the option has a default or computed value (that is, the return value of a func).
parse	Reads options from the command line, an environment variable, or a configuration file and transforms them into appropriate data structures for validation.
validate	Ensures that input values are complete, consistent, and valid.
usage	Displays a help text message.

Table 4-2. Subroutines in the Vim Package

Subroutine	Description
clear_session	Terminates the current session loaded by the `load_session()` subroutine.
find_entity_view	Searches the inventory tree for a managed object that matches the specified entity type.
find_entity_views	Searches the inventory tree for managed objects that match the specified entity type.
get_service_instance	Retrieves a `ServiceInstance` object, which can be used to query the server time or to retrieve the `ServiceContent` object.
get_service_content	Retrieves properties of the service instance, enabling access to the service's managed objects.
get_session_id	Retrieves a session ID.
get_view	Retrieves the properties of a single managed object.
get_views	Retrieves the properties of a set of managed objects.
load_session	Uses a saved session file for connecting to a server.
login	Establishes a session with the Web service running on the vSphere host.

Table 4-2. Subroutines in the Vim Package (Continued)

Subroutine	Description
logout	Disconnects the client from the server and closes the connection to the Web service.
save_session	Saves a session cookie, which is a text file.
update_view_data	Refreshes the property values of a view object.

Table 4-3. Subroutines in the Util Package

Subroutine	Description
connect	Establishes a session by using the token provided in a previously-saved session file, or by using the user name and password provided on the command line, in environment variables, or in a configuration file.
disconnect	If used in conjunction with connect (and a session file), does nothing. If used in conjunction with a user name and password, logs out and closes the session.
get_inventory_path	Returns the inventory path for the specified managed entity.
trace	General-purpose logging function used in conjunction with the `--verbose` command-line option.

Subroutines in the Opts Package

The `Opts` package includes the following subroutines:

add_options

Adds custom options so that they can be submitted to parsing and validation. After the script has validated the options, the script can use them at run time.

Parameters

Parameter	Description
`%opts`	Name of the hash variable that consists of the option name and its attributes.

Returns

Returns nothing.

get_option

Retrieves the value of the specified built-in or custom option.

Parameters

Parameter	Description
`option_name`	String value of the built-in or custom option.

Returns

Returns one of the following, depending upon the attributes defined for the option:

- Return value of `func` (after execution) if a function is associated with the option
- Default value, if one is specified for the option

- Value of the option, as passed to the script
- `Undef` if none of the above are specified

option_is_set

Checks whether an option has been explicitly set by a script or from the command line or whether the option has a default value or computed value (return value of a `func`).

Parameters

Parameter	Description
`option_name`	String value of the built-in or custom option.

Returns

Boolean. Returns `1` (true) if the option value has been explicitly set. Returns 0 (false) if the option value is a default value, is null, or has not been explicitly set. For a discussion of Boolean, see "vSphere SDK for Perl Programming Conventions" on page 11.

parse

Reads options from the command line, an environment variable, or a configuration file and transforms the option into appropriate data structures for validation.

Parameters

No parameters.

Returns

Returns nothing. Displays an error message and quits if the `parse` operation is not successful. If you want to use a configuration file, call `Opts::get_config()` to make sure the file can be opened. See "Opts::get_config()" on page 40.

validate

Ensures that input values (from the command line, an environment variable, or a configuration file) are complete, consistent, and valid.

Parameters

No parameters.

Returns

Returns nothing. It displays an error message and quits if the parse operation is not successful.

usage

Displays the help text message.

Parameters

No parameters.

Returns

Returns nothing.

Subroutines in the Util Package

The `Util` package includes the following subroutines:

connect

Establishes a session with the vCenter Server or ESX/ESXi Web service by using the token provided in a previously saved session file, or by using the user name and password provided using the command line, environment variables, or a configuration file.

Parameters

Parameter	Description
`user_name`	User account on the ESX/ESXi or vCenter Server system.
`password`	Password for the user account.
`session_file`	Full path and filename for the token saved from a previous successful connection. Use `session_file` (instead of `user_name` and `password`) to reestablish a session to the same server or to establish a new connection to a different server.

Returns

Returns nothing.

disconnect

If used in conjunction with `connect` and a session file, does nothing. If used in conjunction with a user name and password, logs out and closes the session.

Parameters

No parameters.

Returns

Returns nothing.

get_inventory_path

Returns the inventory path for the specified managed entity, for example, `Folder`, `Datacenter`, `HostSystem`, `VirtualMachine`, `ComputeResource`, `ClusterComputeResource`, or `ResourcePool`. The resulting inventory path can later be passed to the SOAP operation `FindByInventory` path to retrieve the `ManagedObjectReference` for a managed entity (from which a view can be created).

Parameters

Parameter	Description
`view`	Managed entity view.
`vim_instance`	Managed object.

Returns

Returns a string that identifies the inventory path of the managed entity.

trace

General-purpose logging function used in conjunction with the `--verbose` command-line option. Default log level is 0. Passing the `--verbose` flag without a value sets the level to 1.

Parameters

Parameter	Description
`loglevel`	Numeric value that specifies the log level. Default is 0.
`message`	String that specifies the associated `loglevel` value.

Returns

Returns nothing.

Subroutines in the Vim Package

The `Vim` package includes the following subroutines:

clear_session

Terminates the current session loaded by the `load_session()` subroutine.

Parameters

No parameters.

Returns

Returns nothing.

find_entity_view

Searches the inventory tree for a managed entity that matches the specified entity type. The search begins with the root folder unless the `begin_entity` parameter is specified.

In most cases, you specify a filter or property when using this command to avoid performance problems. See "Creating and Using Filters" on page 33 and "Filtering Views Selectively Using Properties" on page 34.

Parameters

Parameter	Description
`view_type`	Managed entity type specified as one of these strings: ■ "`ClusterComputeResource`" ■ "`ComputeResource`" ■ "`Datacenter`" ■ "Datastore" ■ "DistributedVirtualSwitch" ■ "`Folder`" ■ "`HostSystem`" ■ "Network" ■ "`ResourcePool`" ■ "VirtualApp" ■ "`VirtualMachine`"
`begin_entity` (optional)	Managed object reference that specifies the starting point for the search in the inventory. This parameter helps you narrow the scope.
`filter`	Hash of one or more name-value pairs. The name represents the property value to test and the value represents a pattern that the property must match. If more than one pair is specified, all the patterns must match. Use filters to avoid performance problems. See "Creating and Using Filters" on page 33 and "Filtering Views Selectively Using Properties" on page 34.

Returns

Reference to a view object containing the same properties as the managed entity. If more than one managed entity matches the specified entity type, the subroutine returns only the first managed entity found. If no matching managed entities are found, the subroutine returns `undef`.

find_entity_views

Searches the inventory tree for managed objects that match the specified entity type.

To avoid performance problems, use this command with a filter or specify the `properties` argument. By default, this subroutine retrieves all properties of an entity. See "Creating and Using Filters" on page 33 and "Filtering Views Selectively Using Properties" on page 34.

See the vSphere SDK for Perl API Reference for a list of properties. You can specify properties inherited from `ManagedEntity` or local to a specific entity type.

Parameters

Parameter	Description
`view_type`	Managed entity type specified as one of these strings: ■ "`ClusterComputeResource`" ■ "`ComputeResource`" ■ "`Datacenter`" ■ "Datastore" ■ "DistributedVirtualSwitch" ■ "`Folder`" ■ "`HostSystem`" ■ "Network" ■ "`ResourcePool`" ■ "VirtualApp" ■ "`VirtualMachine`"
`begin_entity` (optional)	Managed object reference that specifies the starting point for search in the inventory. This parameter helps you narrow the scope.

Parameter	Description
`filter` (optional)	Hash of one or more name-value pairs. The name represents the property value to test and the value represents a pattern that the property must match. If more than one pair is specified, all the patterns must match.
properties	Properties to retrieve. Default is all properties. Use a filter or properties to avoid performance problems. See "Filtering Views Selectively Using Properties" on page 34.

Returns

Reference to an array of view objects containing static copies of property values for the matching inventory objects. If no matching entities are found, the array is empty.

Example

The following example, originally published in VMware Communities in post #1272780, retrieves the `name` property from each inventory object. Note that `$entity_views` extracted from the server-side managed object is an array reference, not a scalar.

```
...
my %opts = (
      entity => {
      type => "=s",
      variable => "VI_ENTITY",
      help => "ManagedEntity type: HostSystem, etc",
      required => 1, },
);
Opts::add_options(%opts);
Opts::parse();
Opts::validate();
Util::connect();

# Obtain all inventory objects of the specified type
my $entity_type = Opts::get_option('entity');
my $entity_views = Vim::find_entity_views();
      view_type => $entity_type,
      properties => [ 'name' ]);
...
```

get_service_instance

Retrieves a `ServiceInstance` object, which can be used to query the server time or to retrieve the `ServiceContent` object.

Parameters

No parameters.

Returns

Returns a `ServiceInstance` object.

get_service_content

Retrieves properties of the service instance enabling access to the managed objects of the service. Alternatively, you can use `get_views()`, `get_view()`, and other subroutines to access the objects more directly. If you start with the service content to work with the Web service, you can navigate to the object of interest.

Parameters

No parameters.

Returns

Reference to `ServiceContent` object, which contains managed object references to all inventory content, including the root folder.

get_session_id

Retrieves the session ID corresponding to the current session.

Parameters

No parameters.

Returns

Session ID cookie for use by `load_session()`.

get_view

Retrieves the properties of a single managed object.

Parameters

Parameter	Description
`mo_ref`	Managed object reference obtained from a property of another managed object or a view.
`view_type` (optional)	Type of view to construct from the managed object. If the parameter is absent, the subroutine constructs a view with a type that matches the managed object type name.

Returns

View object containing static copies of a managed object's property values.

get_views

Retrieves the properties of a set of managed objects.

Parameters

Parameter	Description
`mo_ref_array`	Reference to an array of managed object references.
`view_type` (optional)	Type of view to construct from the managed object. If the parameter is absent, the subroutine constructs a view with a type that matches the name of the managed object type.

Returns

Reference to an array of view objects containing copies of property values for multiple managed objects.

Notes

The `Vim::get_views()` subroutine takes a reference to an array of managed object references and returns a reference to an array of view objects. Although the array can contain multiple managed object types, objects of only one type can be obtained at the same time.

load_session

Uses a saved session file or session cookie for connecting to a server. Use `Util::connect()` instead of `Vim::login()` after loading the session.

You can use `save_session()` to get a session file or `get_session_id()` to get a session ID.

Parameters

Parameter	Description
`service_url`	URL of the server to which the client connects (optional if using `session_file`).
`session_file`	Full path and filename for a session file returned by `save_session()`. You must specify either `session_file` or `session_id`. You must pass in the filename as a hash.
`session_id`	Session ID returned by `get_session_id()`. You must specify either `session_file` or `session_id`.

Returns

Returns the vSphere object instance.

Example

To load a session using a session file: `load_session(session_file => $filename);`

To load a session using a session ID: `load_session(service_url => $url, session_id => $sessionid);`

login

Establishes a session with the Web service running on the vCenter Server or ESX/ESXi system using the user name and password credentials provided using the command-line, environment variables, or configuration file.

NOTE In most cases, you use `Util::connect()` instead to establish a connection.

Parameters

Parameter	Description
`service_url`	URL of the server to which the client connects.
`user_name`	User account on the ESX/ESXi or vCenter Server system.
`password`	Password for the user account.

Returns

Returns the vSphere object instance.

logout

Disconnects the client from the server and closes the connection to the Web service. Use this subroutine if you connected using `Vim::login()`. Otherwise, use `Util::disconnect()`.

Parameters

No parameters.

Returns

Returns nothing.

save_session

Saves a session cookie, which is a text file. See "Using a Session File" on page 13.

Parameter

Parameter	Description
`session_file`	Full path and filename where the token should be saved. The session times out after 30 minutes. You pass in the filename as a hash.

Returns

Returns nothing.

Example

```
save_session (session_file => $filename);
```

update_view_data

Refreshes the property values of a view object.

Parameters

No parameters.

Returns

Returns nothing.

Web Services for Management Perl Library

A

Web Services for Management (WS-Management) provides a common way for systems to access and exchange management information across the IT infrastructure.

ESX/ESX version 3.5 and later supports WS-Management by implementing over a dozen CIM (Common Information Model) profiles. CIM profiles are a set of object-oriented schemas defined by the DTMF (Distributed Management Task Force). CIM defines how managed elements in a networked environment are represented as a common set of objects and relationships that users can view, share, and control. For example, system management client applications might be able to check the status of server components such as CPU, fans, power supplies, and so on.

The WS-Management Perl library allows you to write scripts that retrieve CIM data from the ESX/ESXi host using CIMOM, a service that provides standard CIM management functions over a WBEM (Web-Based Enterprise Management). WBEM is a standard protocol for passing CIM-XML messages over HTTP.

Although you can use the WS-Management library with other available WS-Management-enabled CIMOMs, this appendix limits discussion to using the library with the CIMOM available on ESX/ESXi version 3.5 and later and VirtualCenter 2.5 and later.

This appendix includes these topics:

Web Services for Management Overview

The SMASH (Systems Management Architecture for Server Hardware) initiative is one of several related standards initiatives of the DMTF. The SMASH profiles build on other DMTF standards, including the Common Information Model (CIM), an object-oriented approach to modeling managed resources throughout the distributed computing environment. CIM Schemas define classes and associations among the classes in several key areas. CIM Schemas build around a core schema, including devices, applications, network, and the system itself. A CIM object manager brokers requests for data from any of the managed elements.

Clients can use the CIM-XML protocol for CIMOM access. Clients can also use Web Services for Management (WS-Management), a SOAP-based protocol for accessing CIM data. The Perl library discussed in this appendix is an implementation of WS-Management client artifacts (stubs, bindings) for connecting to a WS-Management server and obtaining CIM data.

For information about CIM, SMASH, and WS-Management, visit the dmtf.org Web site. See the *CIM SMASH/Server Management API Programming Guide* for information on CIM/SMASH and ESX/ESXi.

Required Perl Modules

The WS-Management Perl library requires these Perl modules:

- `SOAP::Lite` – Version 0.67 - version 0.69 are supported. Versions before 0.67 or 0.7 or later are not supported.
- `UUID` – Version 0.02 and later.
- `Data::Dump` – Version 1.07 and later.

If the system you are using is behind a firewall, make sure that the `http_proxy` and `ftp_proxy` environment variables are set to match your Proxy server before you proceed with the following instructions for Windows or Linux.

NOTE If you use one of the supported Linux distributions, the required modules are included with the vSphere SDK for Perl and you do not have to install them.

To install required Perl modules on a Windows system

1 Determine which version of Perl you are using by running the **`perl –v`** command.

   - For version 5.6, type the following at the command prompt:

     ```
     C:\>ppm install http://theoryx5.uwinnipeg.ca/ppmpackages/SOAP–Lite.ppd
     ```

   - For version 5.8, type the following at the command prompt:

     ```
     C:\>ppm install http://theoryx5.uwinnipeg.ca/ppms/SOAP–Lite.ppd
     ```

2 Run the following command to install `UUID`:

   ```
   C:>ppm install UUID
   ```

3 Run the following command to install `Data::Dump`:

   ```
   C:>ppm install Data–Dump
   ```

To install required Perl modules on a Linux system

1 Enter the following commands for remote access to CPAN (comprehensive Perl archive network) in a terminal window:

   ```
   $ sudo –s
   # perl –MCPAN –e shell
   ```

2 Run the following command at the CPAN prompt to install `SOAP::Lite`:

   ```
   cpan> install SOAP::Lite
   ```

3 Respond to the questions that appear.

   The module installs.

4 Go to the cpan.org Web site, search for UUID, and download the latest source.

5 Untar the downloaded file, open a terminal window.

6 Go to the untarred directory and run the following commands to install UUID:

   ```
   # make
   # make test
   # make install
   ```

   If running `make` results in errors about missing items, install `uuid–dev` with `apt–get` on Debian-based systems or `e2fsprogs–dev[el]` on other systems.

7 Enter run the following commands to install `Data::Dump`:

   ```
   # perl –MCPAN –e shell
   cpan> install Data::Dump
   ```

After you have installed vSphere SDK for Perl, the following artifacts and samples are in the vSphere SDK for Perl installation directory:

Table A-1. Components and Locations

Path	Description
`Perl/samples/WSMan`	Sample Perl scripts that use the WS-Management library to obtain information through the CIMOM of an ESX/ESXi or vCenter Server system. Sample programs let you check sensor health, obtain firmware revision levels, list field-replaceable units, and list power supply details. See Table A-2, "WSMan Sample Scripts," on page 53.
`Perl/lib/WSMan`	WS-Management interface Perl modules.

Sample Scripts

You can run the sample scripts as is. You can also use the scripts as the starting point for writing your own Perl scripts to obtain CIM data from the server. If you accepted the defaults during vSphere SDK for Perl installation, the samples are in the following location on a Windows system:

```
C:\Program Files\VMware\VMware VI CLI\Perl\samples\WSMan
```

When you run the samples, you must specify connection options. See "vSphere SDK for Perl Common Options" on page 12. For example, you can specify connection options on the command line as follows:

```
perl <scriptname.pl> --server <servername> --username <username> --password <password>
```

For example:

```
perl firmwarerevisions.pl --server my.FQDN.esx35server.com --username root --password root_pass
```

If `--server` is not specified, it defaults to `localhost`. If you are connecting to a remote host and do not specify a user name and password, you are prompted.

The CIMOM service listens for requests on port 80.

Table A-2. WSMan Sample Scripts

Script	Description
`checksensorhealth.pl`	Returns a list of sensors associated with all system devices.
`firmwarerevisions.pl`	Obtains a list of firmware revisions on the system.
`listfrus.pl`	Returns a list of all field-replaceable units on the system.
`listpowersupplies.pl`	Obtains status of discrete sensors associated with all power supplies. Demonstrates traversing associations and using `GetInstance`.

The WS-Management library consists of the `WSMan::Basic` and `WSMan::GenericOps` classes, and the `StubOps.pm` object-oriented wrapper for generic operations. The following sections discuss each library component.

SOAP Message Construction with WSMan::WSBasic

You can use the `WSMan::WSBasic` class to construct SOAP messages for communicating with the WS-Management server. The Perl module is located in `Perl/lib/WSMan/WSBasic.pm`. All operations in this class return deserialized `SOAP::SOM` objects from which you can extract the fault code or the SOAP replies.

You usually do not use this module directly. Instead, you use the `GenericOps` module built on top of `WSBasic`. `GenericOps` supports generic operations as defined by the DMTF standards. See "Generic CIM Operations with WSMan::GenericOps" on page 57. If you want to use the `SOAP::SOM` library directly, see the CPAN documentation for `SOAP::SOM`.

The `WSMan::WSBasic` module requires the following Perl modules:

- `SOAP::Lite` – `WSMan::WSBasic` requires Version 0.65 or later to form SOAP messages and to parse XML replies that are received from the WS-Management server.

- `UUID` – Generates UUIDs for the SOAP messages.

Table A-3 lists the methods the `WSBasic` class provides, which are discussed in more detail below.

Table A-3. Methods in WSMan::WSBasic

Method	Description
WSMan::WSBasic->new	Constructor.
register_xml_ns	Registers extra XML namespaces that might be required for proprietary tags in the SOAP message
register_class_ns	Registers extra CIM namespaces that the WS-Management server might require.
Identify	Performs the `wsmid:Identify` operation, which causes the WS-Management server to identify itself.
Enumerate	Filters results differently depending on the arguments you pass in.
PullRelease	Performs a Pull or a Release operation (overloaded method).
Get	Retrieves an instance of a class.

WSMan::WSBasic->new

Constructor that takes a hash argument containing key-value pairs.

Arguments

The constructor takes the following arguments:

Argument	Description
`address`	WS-Management server URL. Specify the transport protocol by adding `http` (basic user-password authentication) or `https` (HTTP with SSL encryption).
`port`	Port on which the WS-Management server listens for requests.
`path`	Path to the WS-Management server. The path is combined with the address and port arguments to form the complete URL of the WS-Management server. The resulting URL is `http://address:port/path`.
`username`	User name for the WS-Management server.
`password`	Password for the WS-Management server.
`namespace`	CIM namespace. Default is `root/cimv2`.If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.
`timeout` (optional)	Timeout for the HTTP request.

Example

```
$client = WSMan::WSBasic->new( address => 'http://www.abc.com/',
                               port => '80',
                               path => 'wsman',
                               username => 'wsman',
                               password => 'secret',
                               namespace => 'root/cimv2',    #optional
                               timeout => '60'               #optional
                             );
```

register_xml_ns

Registers extra XML namespaces that might be required for proprietary tags in the SOAP message. Calling `register_xml_ns` is not usually required.

Arguments

A hash. Keys are the prefixes, values are the relative URLs as values.

Example

```
$client->register_xml_ns((wsen => 'http://www.dmtf.org/wsen'));
```

Declares a prefix `wsen` with the URL `http://www.dmtf.org/wsen` in the global XML namespace.

register_class_ns

Registers extra ResourceURIs that the WS-Management server might require. By default, the constructor provides a set of ResourceURIs only for classes in the CIM schema. Classes with other schema names, such as `VMware_*` classes, require a different ResourceURI when enumerated using the vSphere SDK for Perl.

You can find the ResourceURIs corresponding to other supported schemas in the OpenWSMan configuration file, which is located in the server's file system at `/etc/openwsman/openwsman.conf`. The ResourceURIs are listed in the value of the `vendor_namespaces` configuration parameter.

Arguments

A hash. Keys are the prefixes, values are the relative URLs as values.

Example

```
$client->register_class_ns((OMC => 'http://schema.omc-project.org/wbem/wscim/1/cim-schema/2',
VMware => 'http://schemas.vmware.com/wbem/wscim/1/cim-schema/2'));
```

Registers the ResourceURIs needed to enumerate classes in the OMC and VMware schemas.

Identify

Performs the `wsmid:Identify` operation, which causes the WS-Management server to identify itself. Helps you determine whether the server is up and running.

Arguments

No arguments.

Returns

Returns a `SOAP::SOM` object, which you can use to parse the results or do error correction.

Enumerate

Filters results depending on the arguments you pass in. Several arguments perform generic operations that are implemented in another class, as described in "Generic CIM Operations with WSMan::GenericOps" on page 57. Other arguments implement enumeration for non-standard-compliant servers. This document discusses the most common arguments. Look at the Perl code for information on other arguments.

Arguments

Accepts the following arguments:

Argument	Description
`class_name`	Specifies the class that you want to enumerate. This argument is passed as a string.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.
`enummode` (optional)	Specifies an enumeration mode such as `EnumerateEPR` or `EnumerateEPRandObject`. This argument is passed as a string.
`polymorphism` (optional)	Specifies polymorphism modes, passed in as a string. For example `IncludeSubClassProperties`, `ExcludeSubClassProperties`, and `None`.

Returns

Returns a `SOAP::SOM` object that can be used to either check for errors (`$result->fault`) or to parse the results (`$result->result`). The SOAP object includes a header and data in XML format.

PullRelease

An overloaded method that performs a `Pull` operation or a `Release` operation.

Arguments

Accepts the following arguments:

Argument	Description
`enumid`	Enumeration ID that the `Pull` or `Release` operation should use. This argument is passed as a string.
`action`	Specifies the operation to perform, `Pull` or `Release`. This argument is passed as a string.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.

Get

Retrieves an instance of a class.

Arguments

Accepts the following named arguments:

Argument	Description
`class_name`	The class whose instance you want to retrieve. This argument is passed as a string.
`options`	Passes keys for the particular instance on which you want to perform a `Get` operation. Passed as a reference to a hash containing the keys in name-value pairs.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.

WSMan::WSBasic Examples

This section shows a few code examples for `WSMan::WSBasic`.

Using Enumeration Modes

To use one of the enumeration modes like `EnumerateEPR` or `EnumerateEPRandObject`, call the `Enumerate` operation with `EnumerationMode` enabled. You can also specify the enumeration mode in the constructor.

```
$result = $client->Enumerate(class_name => 'CIM_Processor',
                                #namespace => 'root/cimv2', #if needed.
                                enummode => 'EnumerateEPR'
                                );
```

Registering Classes

To perform operations on vendor-specific classes, you must register them first with the client. The actual URL depends on your WS-Management software.

```
$client->register_class_ns(Linux => 'http://www.dmtf.org/linux');
```

Using Enumerate and Pull Operations

```
#!/usr/bin/perl -w
use strict;
use WSMan::WSBasic;     #Import the module.

my ($enumid, result, $client); #declaring variables.
```

```
#Construct the client.
$client = WSMan::WSBasic->new(
                          path => 'wsman',
                          username => 'wsman',
                          password => 'secret',
                          port => '8889',
                          address => 'http://something.somewhere.com'
                         );
#Execute the Enumerate method.
$result = $client->Enumerate(class_name => 'CIM_Processor',
                                  #namespace => 'root/cimv2'
                                  );
if($result->fault){
#If a fault occurred, then print the faultstring
die $result->faultstring;
}
else{
   #If no fault occurred then get the enumid.
   $enumid = $result->result;
}
$result = $client->PullRelease(
                                  class_name => 'CIM_Processor',
                                  enumid => $enumid,
                                  action => 'Pull',
                                  #namespace => 'root/cimv2'
                                   );
if($result->fault){
   #If a fault occurred, then print the faultstring
   die $result->faultstring;
}
else{
# Do stuff with $result, which is a SOAP::SOM object containing a deserialized XML reply.
# It is better to use the Generic Operations module, built on top of this module.
}
```

Generic CIM Operations with WSMan::GenericOps

The `GenericOps` module implements some of the generic operations specified in the WS-Management CIM bindings published by the DMTF. Not all generic operations are implemented. The Perl module is located at `Perl/lib/WSMan/GenericOps.pm`.

The `WSBasic` module discussed in "SOAP Message Construction with WSMan::WSBasic" on page 53 provides more primitive intrinsic WS-Management operations. The `GenericOps` module requires the `WSMan::WSBasic` module.

Table A-4 lists the methods the `GenericOps` class provides, which are discussed in more detail below.

Table A-4. Methods in WSMan::GenericOps

Method	Description
WSMan::GenericOps->new	Constructor.
register_xml_ns	Registers extra XML namespaces that might be required for proprietary tags in the SOAP message.
register_class_ns	Registers extra CIM namespaces that the WS-Management server might require.
Identify	Performs the `wsmid:Identify` operation, which causes the WS-Management server to identify itself.
EnumerateInstances	Enumerates the instances of a given class.
EnumerateInstanceNames	Enumerates only the key values of the instances of a given class.
EnumerateAssociatedInstances	Returns the instances related to the source object through an association.
EnumerateAssociatedInstanceNames	Returns objects with only the key values of the associated instance populated.
EnumerateAssociationInstances	Returns objects containing association instances of which the class is a part.

Table A-4. Methods in WSMan::GenericOps (Continued)

Method	Description
EnumerateAssociationInstanceNames	Returns objects containing key values of the association instances of which the class is a part.
GetInstance	Retrieves a particular instance of a class.

WSMan::GenericOps->new

Constructor that takes a hash argument containing key-value pairs in the following form:

```
$client = WSMan::GenericOps->new(( address => 'http://www.abc.com/',
                port => '80',
                path => 'wsman',
                username => 'wsman',
                password => 'secret',
                namespace => 'root/cimv2',     #optional
                timeout => '60'                #optional
                ));
```

Arguments

The constructor has the following arguments:

Argument	Description
`address`	URL of the WS-Management server. Specify the transport protocol by adding the `http` prefix for HTTP (basic user-password authentication) or the `https` prefix for HTTP with SSL encryption.
`port`	Port on which WS-Management listens for requests.
`path`	Path to the WS-Management server. The path is combined with the address and port arguments to form the complete URL of the WS-Management server in `http://address:port/path` order.
`username`	User name for the WS-Management server.
`password`	Password for the WS-Management server.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.
`timeout` (optional)	Timeout for the HTTP request, in case of slow servers.

register_xml_ns

Registers extra XML namespaces that might be required for proprietary tags in the SOAP message. Calling `register_xml_ns` is not required unless you are trying to extend the class itself.

Arguments

A hash. Keys are the prefixes, values are the relative URLs as values.

Example

```
$client->register_xml_ns((wsen => 'http://www.dmtf.org/wsen'));
```

Declares a prefix `wsen` with the URL `http://www.dmtf.org/wsen` in the global XML namespace.

register_class_ns

Registers extra ResourceURIs that the WS-Management server might require. By default, the constructor provides a set of ResourceURIs only for classes in the CIM schema. Classes with other schema names, such as `VMware_*` classes, require a different ResourceURI when enumerated using the vSphere SDK for Perl.

You can find the ResourceURIs corresponding to other supported schemas in the OpenWSMan configuration file, which is located in the server's file system at `/etc/openwsman/openwsman.conf`. The ResourceURIs are listed in the value of the `vendor_namespaces` configuration parameter.

Arguments

A hash. Keys are the prefixes, values are the relative URLs as values.

Example

```
$client->register_class_ns((OMC => 'http://schema.omc-project.org/wbem/wscim/1/cim-schema/2',
VMware => 'http://schemas.vmware.com/wbem/wscim/1/cim-schema/2'));
```

Registers the ResourceURIs needed to enumerate classes in the OMC and VMware schemas.

Identify

Performs the `wsmid:Identify` operation, which causes the WS-Management server to identify itself. Helps you determine whether the server is running.

Arguments

No arguments.

Returns

Prints a fault string if a fault occurs, or returns the reply sent by the server. The reply is a hash reference containing the parsed reply in key-value pairs.

EnumerateInstances

Enumerates the instances of a given class.

Returns

Returns a list of hashes containing the parsed reply from the server, or prints a fault string from the server if an error occurs.

Example

```
$client->EnumerateInstances(
                class_name => 'CIM_Processor',
                namespace => 'root/cimv2'        #optional
                )
```

EnumerateInstanceNames

Enumerates only the key values of the instances of a given class. Similar to `EnumerateInstances`.

Returns

Like `EnumerateInstances`, either returns a list of hashes containing the parsed reply from the server (keys only), or prints a fault string if an error occurs.

EnumerateAssociatedInstances

Returns the instances related to the source object through an association. Results are filtered based on the argument you pass in.

Arguments

Accepts the following arguments:

Argument	Description
`class_name`	Name of the class for which you want to get the associated instances.
`selectors`	Set keys as a reference to a hash. Used to identify the instance of the class mentioned in the `class_name` argument.
`associationclassname` (optional)	Name of the association class for the instance.

Argument	Description
`role` (optional)	Role that the object plays in the association class. The method filters the results according to the role.
`resultclassname` (optional)	Result class name, which must be present in the association. The method returns only those instances.
`resultrole` (optional)	Role that the result class plays in this instance. The method returns the results based on `resultrole`.
`includeresult` (optional)	Further filters query results based on properties of the instances. You can pass in properties as a name-value hash, then pass in a reference to this hash in the `includeresult` named argument.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.

Example

```
$client->EnumerateAssociatedInstances(
                                        class_name => 'CIM_Foo',
                                        selectors => \%hash;
                                        associationclassname => 'CIM_Bar',        #optional
                                        role => 'CIM_Baz',                        #optional
                                        resultclassname => 'CIM_Bat',             #optional
                                        resultrole => 'CIM_Quux',                 #optional
                                        includeresult => \%hash,                  #optional
                                        namespace => 'root/cimv2'                 #optional
```

EnumerateAssociatedInstanceNames

Returns objects with only the key values of the associated instance populated. The usage is the same as for `EnumerateAssociatedInstances`.

EnumerateAssociationInstances

Returns objects containing association instances of which the class is a part. The usage is the same as for `EnumerateAssociatedInstances`.

EnumerateAssociationInstanceNames

Returns objects containing key values of the association instances of which the class is a part. The usage is the same as for `EnumerateAssociatedInstances`.

GetInstance

Retrieves a particular instance of a class.

Arguments

Accepts the following named arguments:

Argument	Description
`class_name`	Name of the class whose instance you want to retrieve, passed as a string
`options`	Keys for the instance on which you want to perform the `GetInstance` operation. The argument is passed as a reference to a hash containing the keys in name-value pairs.
`namespace`	Default CIM namespace. Default is `root/cimv2`. If the namespace is not `root/cimv2`, you must pass in the namespace of the class in this argument.

Returns

Prints a fault string or returns the result in a hash.

Credential Store Perl Library

B

The vSphere SDK for Perl credential store library can be used to automate the logon process for non-interactive client applications by storing the password in a secured local credential cache that the application can access at runtime. You can manage the vSphere credential store using the credential store library included in the vSphere SDK for Perl and discussed in this appendix.

If an application authenticates itself to a vCenter Server system, it requires no additional authentication to access any of the ESX/ESXi systems managed by that vCenter Server system.

Authentication can occur as follows:

- Specifying the authentication information explicitly using one of the command-line parameters (user name and password, url, and so on) or configuration file parameters. See "vSphere SDK for Perl Common Options" on page 12.
- Using a session file. See "Using a Session File" on page 13.
- Using Microsoft SSPI, discussed in "Using Microsoft Windows Security Support Provider Interface (SSPI)" on page 15.
- Using the credential store Perl Library, which is included in the vSphere SDK for Perl and discussed in this appendix.

See "vSphere SDK for Perl Common Options" on page 12 for a discussion of the order of precedence.

This appendix explains how to set up and use the credential store and includes a reference to credential store subroutines. The appendix includes the following topics:

Credential Store Overview

Client applications that launch automatically for unattended operations, such as cron jobs and software agents, must be able to log in to the ESX/ESXi hosts without user assistance. The vSphere Web Services SDK provides client-side credential store libraries and tools for automating the login process in a more secure manner. After the credential store has been set up, system administrators are no longer required to keep passwords in local scripts. The credential store can be set up for an ESX/ESXi system, or for a vCenter Server system. If an application authenticates itself to a vCenter Server system, it requires no additional authentication to access any of the ESX/ESXi systems managed by that vCenter Server system.

The credential store consists of:

- A persistence file used to store authentication credentials. Currently, only passwords are supported. The persistence file maps a remote user account from an ESX/ESXi host to that user's password on the host.

  IMPORTANT The passwords in the file are obfuscated but not encrypted. You must protect the file by other means and carefully control who can access it.

- vSphere Web Services SDK (C# and Java) and vSphere SDK for Perl libraries for programmatically managing the file. vSphere Web Services SDK and vSphere SDK for Perl access the same credential store.

Credential Store Components

The vSphere SDK for Perl includes two credential store files in its installation package:

- `VICredStore.pm` – The Perl package for the credential store library located in:

  ```
  Windows: \Program Files\VMware\VMware vSphere CLI\Perl\lib\VMware\VICredStore.pm
  Linux: /usr/lib/perl5/site_perl/5.8.8/VMware/VICredStore.pm
  ```

  Perl applications can use this package to add, retrieve, delete, update, and list the entries stored in the credential store. The `apps/general/credstore_admin.pl` file is an example for credential store use.

  Each entry in the credential store is a tuple of host name, user name, and password. The password is stored in an obfuscated manner in the credential store.

- `credstore_admin.pl` — A Perl application that uses `VICredStore.pm` for accessing the credential store. You can use `credstore_admin.pl`, which is an example, as a command-line interface to the credential store. `credstore_admin.pl` is located in:

  ```
  Windows: \Program Files\VMware\VMware vSphere CLI\Perl\apps\general\credstore_admin.pl
  Linux: /usr/lib/vmware-viperl/apps/general/credstore_admin.pl
  ```

Managing the Credential Store

You can use Perl to manage the credential store in one of two ways:

- Use the subroutines in `VICredStore.pm` in your Perl script. See "vSphere Credential Store Subroutine Reference" on page 63 for reference documentation. The `credstore_admin.pl` script illustrates how to use the subroutines.
- Use the `\apps\general\credstore_admin.pl` commands to manage the store interactively.

IMPORTANT Create a user with appropriate privileges and store the corresponding user name and password in the credential store. Do not use the root or administrator user and the corresponding password.

Using the Credential Store

After you have set up the credential store with users and passwords, you can use the credentials as follows:

- In your own Perl scripts, you can retrieve passwords or other information as needed using the library subroutine.
- When you run an existing vSphere SDK for Perl or vSphere CLI script, you can specify the host and user name either from the command line or in an environment variable. When that host and user name has an entry with a valid password in the credential store, the script is run.
- If you run a script that includes a host name but no user, and if the credential store contains exactly one entry for that host, the script takes the user from that credential store entry and not prompt for a user.
- When you call an existing vSphere SDK for Perl or vSphere CLI script, and you specify only the host name, the authentication mechanism prompts for a user name. If no entry exists for that user, the authentication mechanism also prompts for a password.

vSphere Credential Store Subroutine Reference

The `ViCredStore` package includes the following subroutines:

init

Initializes the credential store. Call this subroutine once, before any of the other credential store subroutines. The credential store is not created until your program calls `add_password`.

This subroutine accepts the location of the credential store file. If you do not provide a credential store filename `VICredStore::init()` looks in the default location.

- Linux: `$HOME/.vmware/credstore/vicredentials.xml`
- Windows: `%APPDATA%\VMware\credstore\vicredentials.xml`

If no credential store exists at the default location, the initialization process:

- Checks that the `credstore` directory exists, and creates one if it does not.
- Creates the `vicredentials.xml` file and parent directory.

If you provide a non-default credential store filename to `VICredStore::init()`, the credential store at that location is used. If there is no credential store at that location and the directory you specify exists, the initialization process creates the file. If the directory you specify does not exist, the initialization process fails.

Parameters

Parameter	Description
`filename`	Name of credential store file.

Returns

Returns 1 if initialization is successful; otherwise, returns 0.

get_password

Retrieves the password for a specified server and user name.

Parameters

Parameter	Description
`server`	Server for which you want to retrieve the password for the specified user. Can be an ESX/ESXi or vCenter Server system.
`username`	User for whom you want to retrieve the password.

Returns

Returns the password, or `undef` if no password is found.

add_password

Creates a credential store file if none exists and stores the password for a given server and user name.

If a password already exists for that server and user name, `add_password` overwrites that password.

Parameters

Parameter	Description
`server`	Server for the new entry. Can be an ESX/ESXi or vCenter Server system.
`username`	User name for the new entry. VMware recommends that you create a user with appropriate privileges and store the corresponding user name and password in the credential store. Do not use the root or administrator user and the corresponding password.
`password`	Password for the new entry.

Returns

Returns 1 if a password for this server and user does not exists; otherwise, returns zero.

remove_password

Removes the password for a given server and user name. If no password exists, this method has no effect.

Parameters

Parameter	Description
`server`	Server from which the password for the specified user is removed. Can be an ESX/ESXi or vCenter Server system.
`username`	User name for which the associated password is removed.

Returns

Returns 1 if the password existed and was successfully removed; otherwise, returns zero.

clear_passwords

Removes all passwords.

Parameters

No parameters.

Returns

Returns nothing.

get_hosts

Returns a list of all servers that have entries in the credential store.

Parameters

No parameters.

Returns

Returns a list of all servers in the credential store.

get_usernames

For a given server, returns all user names that have an associated password stored in the credential store.

Parameters

`server` – Server for which all user names are listed.

Returns

Returns a list of all users belonging to the specified server.

close

Closes the credential store, and frees all resources associated with it. If you want to run additional credential store subroutines, you must run `init` again to reinitialize the credential store.

Call `close` only once for each credential store initialized by a call to `init`.

Parameters

No parameters.

Returns

Returns nothing.

credstore_admin.pl Utility Application

`credstore_admin.pl` is a utility application you can use for credential store administration. At the same time, the utility serves as sample code if you want to write your own script.

In addition to the options listed in "Common Options Reference" on page 16, the utility supports the following options:

Table B-1. Command-line Options for credstore_admin.pl

Option	Description

`add`
`-s|--server <server>`
`-u|--username <username>`
`-p|--password <password>`
`-t|--thumbprint <thumbprint>`
	Adds a new user name and password entry into the credential store for the specified user.
	Alternatively, adds the thumbprint for a specified server io the credential store.
	Note: You cannot add the user name, password, and thumbprint for a server with one command. Add first the username and password, and run the command again to add the thumbprint.

`get`
`-s|--server <server>`
`-u|username <username>`
`-t|--thumbprint <thumbprint>`
	Retrieves the password or the thumbprint for the specified user from the credential store.

`remove`
`-s|--server <server>`
`-u|--username <username>`
`-t|--thumbprint <thumbprint>`
	Removes an existing password or an existing thumbprint for the specified user from the credential store.

`list` `[-s|--server <server>]`	Lists existing entries.
`clear`	Deletes all entries from the credential store.

Index