

Programming Guide

VMware Infrastructure Perl Toolkit 1.5



Programming Guide
Revision: 20080110

You can find the most up-to-date technical documentation on our 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

© 2007, 2008 VMware, Inc. All rights reserved. Protected by one or more of U.S. Patent Nos. 6,397,242, 6,496,847, 6,704,925, 6,711,672, 6,725,289, 6,735,601, 6,785,886, 6,789,156, 6,795,966, 6,880,022, 6,944,699, 6,961,806, 6,961,941, 7,069,413, 7,082,598, 7,089,377, 7,111,086, 7,111,145, 7,117,481, 7,149,843, 7,155,558, 7,222,221, 7,260,815, 7,260,820, 7,269,683, 7,275,136, 7,277,998, 7,277,999, 7,278,030, 7,281,102, and 7,290,253; patents pending.

VMware, the VMware “boxes” logo and design, Virtual SMP and VMotion are registered trademarks or trademarks 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	5
1 Getting Started With VI Perl Toolkit	7
VI Perl Toolkit Architecture	7
Example Scenarios	8
VI Perl Toolkit Components	8
Common VI Perl Toolkit Tasks	9
VI Perl Toolkit Programming Conventions	9
VI Perl Toolkit Common Options	10
Specifying Options	10
Passing Parameters at the Command Line	10
Setting Environment Variables	10
Using a Configuration File	11
Using a Session File	11
Common Options Reference	12
Examples	12
Hello Host: Running Your First Script	13
2 Writing VI Perl Toolkit Scripts	15
Basic VI Perl Toolkit Script Pattern	15
Understanding Server-Side Objects	20
Using the Managed Object Browser to Explore Server-Side Objects	20
Types of Managed Objects and the Managed Object Hierarchy	21
Managed Object Hierarchy	22
Managed Entities in the Inventory	22
Accessing Server-Side Inventory Objects	23
Understanding Perl View Objects	24
Working With View Object Property Values	25
Accessing Property Values	25
Accessing Simple Property Values	25
Accessing Enumeration Property Values	25
Modifying Property Values	26
Creating Data Objects With Properties	26
Understanding Operations and Methods	26
Non-Blocking and Blocking Methods	27
Examples of Operations	27
Calling Methods	27
Omitting Optional Arguments in Method Calls	28
Updating View Objects	28
3 Refining VI Perl Toolkit Scripts	29
Creating and Using Filters	29
Using Filters with Vim::find_entity_view() or Vim::find_entity_views()	29
Using Filters on the Utility Application Command Line	30
Retrieving the ServiceInstance Object on the ESX Server Host	31
Saving and Using Sessions	31
Learning About Object Structure Using Data::Dumper	31

4 VI Perl Toolkit Subroutine Reference 35

- Packages 36
- Subroutines in the Opts Package 36
 - add_options 36
 - get_option 36
 - option_is_set 36
 - parse 37
 - validate 37
 - usage 37
- Subroutines in the Util Package 37
 - connect 37
 - disconnect 37
 - get_inventory_path 38
 - trace 38
- Subroutines in the Vim Package 38
 - clear_session 38
 - find_entity_view 38
 - find_entity_views 39
 - get_service_instance 39
 - get_service_content 40
 - get_session_id 40
 - get_view 40
 - get_views 40
 - load_session 40
 - login 41
 - logout 41
 - save_session 41
 - update_view_data 41

Glossary 43

About This Book

This book, the *Programming Guide*, provides information about writing and running VI Perl Toolkit scripts on VMware® Infrastructure (VI) hosts. Because toolkit subroutines allow you to manage VI hosts using VMware Infrastructure API (VI API) calls, a brief description of the server-side VI object model is included. This guide focuses on explaining how to access and modify server-side objects using the VMware Infrastructure (VI) Perl Toolkit and on discussing some programming techniques.

Revision History

This guide is revised with each release of the product or when necessary. A revised version can contain minor or major changes. [Table 1](#) summarizes the significant changes in each version of this guide.

Table 1. Revision History

Revision	Description
20070105	First version of the documentation for the VI Perl Toolkit 1.0.
20080110	Update for VI Perl Toolkit 1.5. Documentation bug fixes and some clarifications and additions.

To view the most current version of this guide, go to http://www.vmware.com/support/pubs/sdk_pubs.html.

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 VI Perl Toolkit to manage and monitor the hosts in the VMware Infrastructure 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 VI Perl Toolkit subroutines to access the objects on the VMware Infrastructure host and manipulate those objects using Perl. This document includes an in-depth discussion of the VMware Infrastructure object model and explains how you can preview and retrieve those objects and their attributes and methods.

Document Feedback

VMware welcomes your suggestions for improving our documentation. If you have comments, send your feedback to:

docfeedback@vmware.com

Technical Support and Education Resources

The following sections describe the technical support resources available to you. You can access the most current versions of other VMware manuals by going to:

<http://www.vmware.com/support/pubs>

Online Support

You can submit questions or post comments to the Developer Community: SDKs and APIs forum, which is monitored by VMware technical support and product teams. You can access the forum at:

<http://communities.vmware.com/community/developer>

Support Offerings

Find out how VMware support offerings can help meet your business needs. Go to:

<http://www.vmware.com/support/services>.

VMware Education Services

VMware courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. For more information about VMware Education Services, go to:

<http://mylearn1.vmware.com/mgrreg/index.cfm>.

Getting Started With VI Perl Toolkit

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

The chapter covers the following topics:

- [“VI Perl Toolkit Architecture”](#) on page 7
- [“Common VI Perl Toolkit Tasks”](#) on page 9
- [“VI Perl Toolkit Common Options”](#) on page 10
- [“Hello Host: Running Your First Script”](#) on page 13

VI Perl Toolkit Architecture

Before you start using the VI Perl Toolkit, it's important you understand how the toolkit and the VI API on the host interact. This interaction model directly affects how each script is structured, and is the basis for troubleshooting should things go wrong.

There is a limited number of Perl Toolkit subroutines, and they perform variations of these basic tasks:

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

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

Example Scenarios

This section looks at two example scenarios.

Assume you want to use the toolkit to retrieve performance information for a host. You might perform the following tasks:

- Check the *VI Perl Toolkit Utility Applications Reference* or the `/usr/lib/vmware-viperl/apps` directory (Linux) or `Program Files\VMware\VMware VI Perl Toolkit\Perl\apps` folder (Windows) for a script that retrieves performance information.

NOTE All utility applications are fully supported. If you can't find a utility application, you can also use the scripts in the `./usr/share/doc/vmware-viperl/samples` or `Program Files\VMware\VMware VI Perl Toolkit\Perl\samples` folder. The `samples` scripts are not fully supported but meant as starting points for your own scripts.

You find the `viperformance.pl` script, which retrieves performance counters from the host.

- Call the script with the `--help` option or without any options to see its online documentation. For more detailed information than that provided by `--help`, see the *Utility Applications Reference* at http://www.vmware.com/support/developer/viperltoolkit/viperl15/doc/perl_toolkit_utilities_idx.html.
- To actually retrieve the information, call the `viperformance.pl` script with the name of the host, user name and password for log in, an output file name, and the counters you want to retrieve:

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

You must escape special characters in passwords on Linux. See [Table 1-2, “Common Options Reference,”](#) on page 12 for a complete list of connection parameters.

Assume you want to use the toolkit for a task that cannot be performed using one of the utility applications.

- Check the `/samples` folder for a sample script that performs a similar task. The scripts in the `samples` folder are available for you to customize.
- You can modify existing scripts or write your script using the VI Perl subroutines.
 - For an in-depth discussion of scripts that includes an example see [“Writing VI Perl Toolkit Scripts”](#) on page 15.
 - See [Chapter 4, “VI Perl Toolkit Subroutine Reference,”](#) on page 35.
 - For information about the server-side object your script interacts with, see the VI API Reference documentation.
- To execute a VI Perl Toolkit subroutine, use the parameter name followed by its value, 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>, );
```

- Any script you write can be called with the options listed in [Table 1-2, “Common Options Reference,”](#) on page 12. If you want to specify additional options, you can use the mechanism discussed in [“\(Optional\) Define Script-Specific Command-Line Options”](#) on page 16.

VI Perl Toolkit Components

The VI Perl Toolkit has these components:

- **VI Perl Toolkit Runtime** – Client-side runtime components that include:
 - A complete Perl binding of the VI API, which makes all server-side operations and data structures available. The toolkit handles the data type mapping between server-side and client-side objects transparently.
 - VMware Perl modules (`VIruntime.pl`, `VILib.pl`) that provide subroutines for basic functionality.

- **VI Perl Toolkit Utility Applications** – Management applications that you can run, as is, in your virtual data center. You execute each application with connection parameters and other, application-specific parameters. See the *VI Perl Toolkit Utility Applications Reference*.
- **Sample Scripts** – The VI Perl Toolkit includes sample scripts that you can quickly adapt to your own needs, and use to learn more about the VI Perl Toolkit's functionality. Unlike the utility applications, the sample scripts are not supported by VMware. You typically cannot use the sample scripts as is. You need to know Perl to customize the samples.

Common VI Perl Toolkit Tasks

The VI Perl Toolkit includes utility applications or samples for common administration tasks.

Table 1-1. Common Administration Tasks and Toolkit Utilities

Task	Script	Folder
Discovery (logging on)	<code>connect.pl</code>	<code>/apps/general</code>
Performance monitoring	<code>viperformance.pl</code> (retrieves performance counters from host)	<code>/apps/performance</code>
Virtual machine power operations	<code>vmcontrol.pl</code>	<code>/apps/vm</code>
Virtual machine snapshot and restore	<code>vmnapshot.pl</code> , <code>snapshotmanager.pl</code>	<code>/apps/vm</code>
Virtual machine migration	<code>vmmigrate.pl</code>	<code>/apps/vm</code>
Perform host operations, for example, adding a standalone host to a VirtualCenter Server, shutting down and rebooting a host, and so on.	<code>hostops.pl</code>	<code>/apps/host</code>
View or change allocation of CPU or memory shares on a virtual machine.	<code>sharesmanager.pl</code>	<code>/apps/vm</code>

Some tasks are more complex and require additional scripting. See [Chapter 2, “Writing VI Perl Toolkit Scripts,”](#) on page 15.

VI Perl Toolkit Programming Conventions

Because the toolkit interacts with a server using SOAP/WSDL, a number of common programming conventions are different than you might expect.

- **Boolean data types** – You send and receive Boolean values as follows:
 - **Input**, that is, sending from the client application:
 - false: Use 0, '0', or 'false' (capitalization ignored)
 - true: Use 1, '1', or 'true' (capitalization ignored)
 - **Output**, that is, receiving from the server:
 - false: Return value is 0
 - true: Return value is 1
- **Date/Time** – The server returns a SOAP `dateTime` value. You can use the `Date::Parse` Perl module to process these objects.

The VI Perl Toolkit 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 Perl Toolkit always returns `dateTime` values in this format.
- **SOAP error message** – When you see a SOAP error as a result of a call, there is most likely an error on the server, not an error with the communication to the server.

VI Perl Toolkit Common Options

A number of options are available for any VI Perl Toolkit 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>
```

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

Specifying Options

You can specify the common options in several different ways.

- [“Passing Parameters at the Command Line”](#) on page 10
- [“Setting Environment Variables”](#) on page 10
- [“Using a Configuration File”](#) on page 11
- [“Using a Session File”](#) on page 11



CAUTION Be sure to limit read access to a configuration file, especially if it contains user credentials.

The VI Perl Toolkit first processes any options that are set in the configuration file, next any environment variables, and finally command-line entries. This order of precedence always applies. That means, for example, that you cannot override an environment variable setting by using a configuration file.

Passing Parameters at the Command Line

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

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

For example, you can run `connect.pl` as follows:

```
connect.pl --server <server> --username <privileged_user> --password <password>
```

Enclose passwords and other text with special characters in quotation marks or escape each special character with a backslash (`\`). Special characters are characters that have special meaning to the shell, such as `'$'` in Linux environments.

On Linux, use single quotes (`' '`), on Windows, use double quotes (`" "`).

Setting Environment Variables

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

For example:

```
set VI_SERVER=<your_server_name>
```

The following example shows the contents of a `/root/.visdkrc` configuration file:

```
VI_SERVER = 10.17.211.138
VI_USERNAME = root
VI_PASSWORD = <root_password>
VI_PROTOCOL = https
VI_PORTNUMBER = 443
```

NOTE Do *not* escape special characters in the configuration file.

Using a Configuration File

You use a text file that contains variable names and settings as a configuration file. Variables corresponding to the parameters are shown in [Table 1-2](#). You can then execute a script with the configuration file as in the following example:

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

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

NOTE Use `--config` if the configuration information is saved in a different file than `./visdkrc`. If you specify `--config`, the system ignores the `./visdkrc` settings.

Using a Session File

For non-interactive use or other applications where storing a clear text password is not desirable, you can create a session file using the `save_session.pl` script included in the `apps/session` directory.

To create a session file

- 1 Call `save_session.pl`.

You must supply connection parameters and the name of a session file in which the script can save an authentication cookie. The cookie does not contain a user name or password in a readable form. This cookie has a limited life time, it typically expires after 30 minutes of inactivity. On some servers, the session life time can be configured.

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

- 2 Call Remote CLI commands and pass in the session file using the `--sessionfile` parameter.

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

Common Options Reference

Table 1-2 lists options that are available for all VI Perl Toolkit scripts. Use the parameter on the command line and the variable in configuration files.

Table 1-2. Common Options Reference

Parameter	Variable	Description
<code>--config</code>	VI_CONFIG	Use the VI Perl Toolkit configuration file at the specified location. NOTE: You must specify a path that is readable from the current directory.
<code>--password</code>	VI_PASSWORD	Use the specified password (used in conjunction with <code>--username</code>) to log in to the server. <ul style="list-style-type: none"> ■ If <code>--server</code> specifies a VirtualCenter Server, the user name and password apply to that server. No passwords are then needed to execute on the ESX Server hosts that server manages. ■ If <code>--server</code> specifies an ESX Server host, the user name and password apply to that server. NOTE: Use the empty string (' ' on Linux and " " on Windows) to indicate no password. Enclose passwords with special characters in quotation marks (' ' on Linux and " " on Windows) or escape each special character with a backslash (\).
<code>--portnumber</code>	VI_PORTNUMBER	Use the specified port to connect to the ESX Server host. Default is 443.
<code>--protocol</code>	VI_PROTOCOL	Use the specified protocol to connect to the ESX Server host. Default is HTTPS.
<code>--save-session file</code>	VI_SAVESESSIONFILE	File for saving a session.
<code>--server</code>	VI_SERVER	Use the specified VI server. Default is localhost.
<code>--servicepath</code>	VI_SERVICEPATH	Use the specified service path to connect to the ESX Server host. Default is /sdk/webService.
<code>--sessionfile</code>	VI_SESSIONFILE	Use the specified session file to load a previously saved session. The session must be unexpired.
<code>--url</code>	VI_URL	Connect to the specified VMware Infrastructure SDK (VI SDK) URL.
<code>--username</code>	VI_USERNAME	Use the specified user name. <ul style="list-style-type: none"> ■ If <code>--server</code> specifies a VirtualCenter Server, the user name and password apply to that server. No passwords are then needed to execute on the ESX Server hosts that server manages. ■ If <code>--server</code> specifies an ESX Server host, the user name and password apply to that server.
<code>--verbose</code>	VI_VERBOSE	Display additional debugging information.
<code>--version</code>		Display version information.

Examples

The following examples illustrate passing in options. These are Linux-style example unless otherwise noted.

Save a session file, then call the `vminfo.pl` script from `/apps/vm` to list all properties of a specified virtual machine.

```
cd /usr/share/doc/vmware-viperl/samples/session
perl save_session.pl --save-sessionfile /tmp/vimsession --server <servername_or_address>
    --username <username> --password <password>
vminfo.pl --sessionfile /tmp/vimsession --vmname <name>
```

Connect to the server as user `snow-white` with password `dwarf$`. The first example escapes the special characters, the other two use single quotes (Linux) and double quotes (Windows).

```
vminfo.pl --server <server> --user snow\white --password dwarf\$ --vmname <name>
Linux: vminfo.pl --server <server> --user 'snow-white' --password 'dwarf$' --vmname <name>
Windows: vminfo.pl --server <server> --user "snow-white" --password "dwarf$" --vmname <name>
```

Hello Host: Running Your First Script

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

- Successful VI Perl Toolkit installation. See the *VI Perl Toolkit Installation Guide* for information.
- Access to one of the supported VMware Infrastructure hosts. Perform a connection check using the process described in [“Using the Managed Object Browser to Explore Server-Side Objects”](#) on page 20.

To execute the connect.pl script

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

2 Execute `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 invoke any utility application with `--help` to see information about its parameters.

Writing VI Perl Toolkit Scripts

This chapter illustrates how to write a VI Perl Toolkit script using a simple example script. The chapter also explores the basics of the VI API object model, in these sections:

- [“Basic VI Perl Toolkit Script Pattern”](#) on page 15
- [“Understanding Server-Side Objects”](#) on page 20
- [“Understanding Perl View Objects”](#) on page 24
- [“Working With View Object Property Values”](#) on page 25
- [“Understanding Operations and Methods”](#) on page 26
- [“Updating View Objects”](#) on page 28

NOTE This chapter does not discuss Perl basics. You are expected to know Perl and understand its programming conventions. For example, when you develop a VI Perl Toolkit script, you should 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).

Basic VI Perl Toolkit Script Pattern

In general, VI Perl Toolkit scripts retrieve objects, such as virtual machines, from the server and work with them. Many VI Perl Toolkit scripts follow the same basic pattern, shown in [Example 2-1](#).

Example 2-1. Basic VI Perl Toolkit Pattern (simpleclient.pl)

```
#!/usr/bin/perl
use strict;
use warnings;
use VMware::VIRuntime;
```

[Step 1, “Import the VI Perl Toolkit Modules,”](#)
on page 16

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

[Step 2, “\(Optional\) Define Script-Specific
Command-Line Options,”](#) on page 16,

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

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

[Step 3, “Connect to the Server,”](#) on page 18

```

# 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();

```

[Step 4, “Obtain View Objects of Server-Side Managed Objects,”](#) on page 18

[Step 5, “Process Views and Report Results,”](#) on page 18

[Step 6, “Close the Server Connection,”](#) on page 19

Step 1 Import the VI Perl Toolkit Modules

All VI Perl scripts should 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 you can use to connect to a VirtualCenter Server or ESX Server host, and to retrieve views, the client-side Perl objects that encapsulate the properties and operations of server-side managed objects. These subroutines are organized into three different packages:

- The `Opts` package (in the `VILib.pm` module) includes subroutines for handling built-in options and creating custom options.
- The `Util` package includes subroutines that facilitate routine tasks, such as setting up and closing connections to the server.
- The `Vim` package includes subroutines for accessing server-side managed objects, instantiating local proxy objects (views), updating properties, and invoking local methods to effect operations on remote operations.

See [“VI Perl Toolkit Subroutine Reference”](#) on page 35.

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

When you invoke 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. VI Perl Toolkit lets you specify these options in a variety of ways. See [“Specifying Options”](#) on page 10.

A number of common command-line options, most of them connection options, are already defined for all utility applications (see [Table 1-2, “Common Options Reference,”](#) on page 12). In addition, most applications have application-specific options you pass to the script at execution time.

The VI Perl Toolkit has defined all common options using attributes and subroutines specified in the `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 seven entity types (for example `HostSystem`, `ResourcePool`, or `VirtualMachine`) is passed as a parameter to the `Vim::find_entity_views()` subroutine for further processing. The example creates new command-line options in two steps.

- First the example declares the option as a hash, where the key is the option name, and the value is a hashref containing `Getopt::Long`-style option attributes (See [Table 2-1](#) 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,
    },
);
```

- Then the example adds the option to the default options using the `Opts::add_options()` subroutine:

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

[Table 2-1](#) lists all attributes you can use to define command-line options. The code fragment in [Step 1](#) above uses only `type`, `variable`, `help`, and `required`. For related information, see the documentation for the `Getopt::Long` module.

Table 2-1. 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 <code>undef</code> to the calling script.										
func	Enables creating derived options. Set <code>func</code> to an external code reference to execute the code when the toolkit 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 value is optional.										
variable	Allows you to specify the option in an environment variable or a configuration file. See "Specifying Options" on page 10.										
type	Uses Perl <code>Getopt</code> -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).										
	<table border="1"> <thead> <tr> <th>Mandatory</th> <th>Optional</th> <th>String</th> <th>Integer</th> <th>Float</th> </tr> </thead> <tbody> <tr> <td>=</td> <td>:</td> <td>s</td> <td>i</td> <td>f</td> </tr> </tbody> </table>	Mandatory	Optional	String	Integer	Float	=	:	s	i	f
Mandatory	Optional	String	Integer	Float							
=	:	s	i	f							

- Finally, 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 10). The `Vim::find_entity_views()` subroutine uses the value the user passes in later in the script. The value must be one of the seven 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 VI Perl Toolkit displays usage help for the script, as in the following example:

```
C:\viperltoolkit>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, or VirtualMachine
```

Step 3 Connect to the Server

The VI API is hosted as a secure web service on VirtualCenter Server and ESX Server hosts. By default, the web service is available over HTTPS, so clients must provide valid credentials to connect to the service. If you reconfigured your host to use HTTP, you do not have to provide login credentials. Depending on the specifics of your server, you might need to enter only a user name and password. You might need other information. (See [Table 1-2, “Common Options Reference,”](#) on page 12).

To connect to the server, call `Util::connect()`.

When a script reaches the call to `Util::connect()`, the VI Perl Toolkit 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 Perl Toolkit Runtime automatically 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 `find_entity_views()` subroutine creates a local Perl object (an array of references) from the server-side managed object of the specified entity type.

NOTE This object is static and not automatically 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 object's properties and methods, so you must understand the server-side objects. See [“Understanding Server-Side Objects”](#) on page 20 for an introduction. For in-depth information about server-side objects, see the *VI API Reference Guide* which is included with the VI Perl Toolkit documentation.

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 (`@$entity_views`) and accessing the `name` property of each entity by calling `$entity_view->name`. The example then prints the name of each entity to the console.

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

See [“Understanding Server-Side Objects”](#) on page 20.

Step 6 Close the Server Connection

To log out and exit, use the `Util::disconnect()` subroutine.

[Example 2-2](#) shows the complete listing for `simpleclient.pl`.

Example 2-2. Sample script that uses basic pattern (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, Folder, HostSystem, ResourcePool, or VirtualMachine) found
# on the target VirtualCenter Server or ESX Server system.
# Script users must provide logon credentials and the managed entity type.
# The simpleclient.pl 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 Execute the script using the following syntax:

```
perl simpleclient.pl --server <my_server> --username <my_username> --password <password>
--entity <EntityType>
```

For example:

```
C:\viperltoolkit>perl simpleclient.pl --server aquarium.mycompany.com --username abalone
--password tank --entity HostSystem
Found HostSystem: lgto-1s-dhcp214.eng.vmware.com
```

Understanding Server-Side Objects

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

The *VI API Reference Guide* gives reference documentation for all VI API objects. Some users might also find the *VMware Infrastructure SDK Programmer's Guide* helpful. It is available from the SDK download site at <http://www.vmware.com/download/sdk/index.html>.

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 invoke a method on the server.

Using the Managed Object Browser to Explore Server-Side Objects

The MOB is a Web-based server application hosted on all VMware ESX Server hosts and VirtualCenter 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 VMware Infrastructure object model.

NOTE If the ESX Server host or VirtualCenter Server system uses HTTPS (the default), you need a user name and password to log into the MOB.

To access the MOB on any ESX Server or VirtualCenter Server system

- 1 Launch a Web browser on your development system.
- 2 Connect to the MOB using the fully-qualified domain name (or the IP address) of the ESX Server host or VirtualCenter Server, as follows:

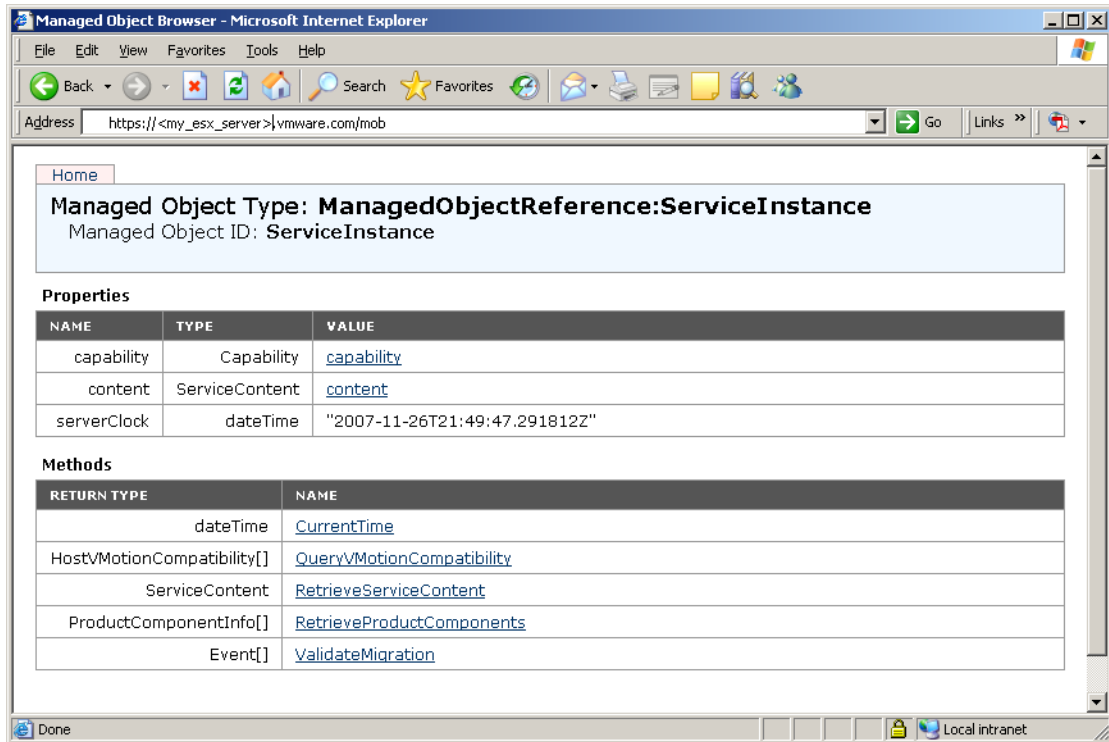
`https://<hostname.yourcompany.com>/mob`

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 a 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.

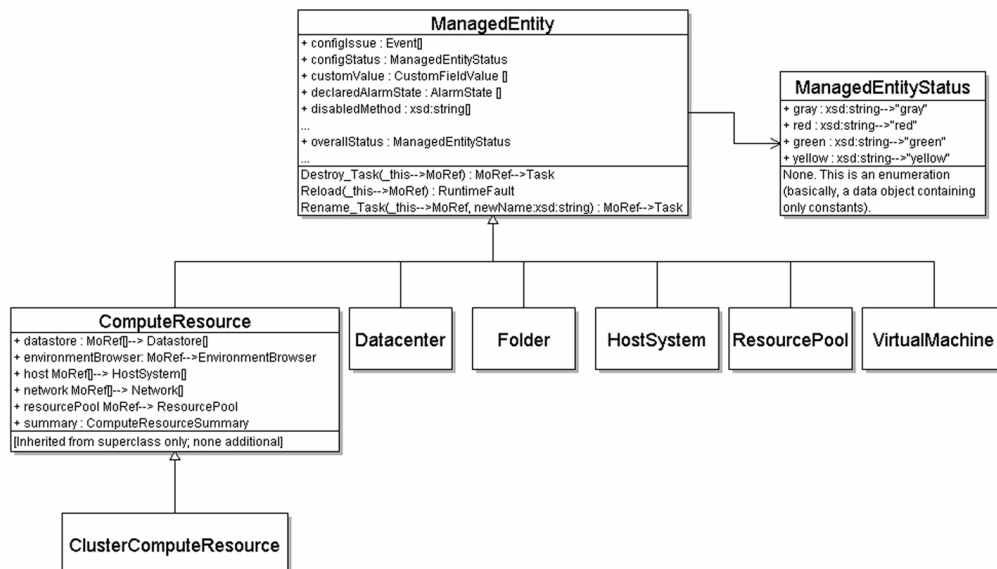
When you are successfully connected to the MOB, the browser displays the managed object reference for the service (ManagedObjectReference:ServiceInstance), available properties (with values), and methods, as shown in this example:



Types of Managed Objects and the Managed Object Hierarchy

A *managed object* is the primary type of object in the VMware Infrastructure 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 2-1 shows the ManagedEntity hierarchy as an example. See “Managed Entities in the Inventory” on page 22.

Figure 2-1. ManagedEntity Hierarchy



The different managed objects define the entities in the inventory as well as 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`). For a detailed discussion of managed objects, see the *VI 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 objects, and makes the state and methods available to client applications. Clients invoke 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 [“Using the Managed Object Browser to Explore Server-Side Objects”](#) on page 20) or use the API Reference documentation.

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

To view documentation for server-side objects

- 1 Find the *VI API Reference Guide*, which is included with the VI Perl Toolkit documentation.
- 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 also contains references to inventory objects, which allow you to access the entities such as hosts (`HostSystem`) and virtual machines (`VirtualMachine`) in the virtual datacenter. `ServiceContent` properties allow you 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.

NOTE The VI Client displays the hierarchy of inventory objects. The VI Client uses the information about the objects (the properties and the relationships among them) for the display. For information about the VI Client display, see the VMware Infrastructure 3 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 the VMware Infrastructure managed objects such as datacenters and hosts. See [Figure 2-1](#) 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 `VirtualCenter Server` instance can support multiple datacenters, but an `ESX Server` host supports only one datacenter. The single datacenter associated with each `ESX Server` host is not included in the VI Client display.
- `Folder` – Contains references to other entities, for example, other folders (`Folder`) or hosts (`HostSystem`).
- `HostSystem` – Provides access to a virtualization host platform.
- `VirtualMachine` – Represents a single virtual machine.

- **ResourcePool** – Allows you to combine CPU and memory resources from multiple hosts and 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 VMware Infrastructure documentation set, for more information.
- **ComputeResource** – Abstracts a host system’s physical resources and allows you to associate those resources with the virtual machines that run on the host.

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 has a **name** property that it inherits from **ManagedEntity**.

Accessing Server-Side Inventory Objects

The VI Perl Toolkit 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 2-3 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 2-3 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 2-3. 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 VI Perl Toolkit. A view object has the following characteristics:

- Is a Perl object.
- Includes properties and methods that correspond to the properties and operations of the server-side managed object.
- Is a static copy of a server-side managed object and is *not* automatically updated when the object on the server changes. See [“Updating View Objects”](#) on page 28.
- Includes these additional methods (beyond the operations offered in the server-side managed object):
 - An accessor method for each managed object property. The VI Perl Toolkit 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 27.
 - A method that updates the state of any client-side view object with current data from the server. See [“Updating View Objects”](#) on page 28.
- View object properties correspond to properties of server-side managed objects as follows:
 - Simple property types (strings, booleans, numeric datatypes, such as integers, floats, and so on) become view object properties of the same name.
 - Arrays of properties become arrays of properties of the same name.
- For each simple property (string, boolean, numeric data type), including inherited simple properties, the toolkit creates an accessor method. The accessor method name is the same as the property name.

The VI Perl Toolkit simplifies programming as follows:

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

Working With View Object Property Values

VI Perl Toolkit view objects are Perl objects. You can retrieve a view, manipulate its properties, and invoke its methods using Perl's object-oriented syntax.

This section explains how to work with properties in the following sections:

- [“Accessing Property Values”](#) on page 25
- [“Modifying Property Values”](#) on page 26
- [“Creating Data Objects With Properties”](#) on page 26

Accessing Property Values

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

Table 2-2. Property Overview

Property	Example
Simple data type, such as a string, boolean, numeric, or date <code>Time</code> .	The <code>ManagedEntity</code> managed object has a <code>name</code> property of type string.
Array of simple data types or data objects.	A <code>HostSystem</code> 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 one of only three possible string values such as <code>poweredOn</code> , <code>poweredOff</code> , or <code>suspended</code> .
Complex data types called <i>data objects</i> that are part of the VMware Infrastructure object model.	<code>AboutInfo</code> , <code>Action</code> , and <code>ServiceContent</code> 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` 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 (`val`) itself, 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 change the value of a property that is defined as an enumeration, you must pass the new value to the method defined for the property in the Perl object, for example:

```
$data_object->property = new PropertyType(new value)
$vm_view->runtime->powerState = new VirtualMachinePowerState("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 VI 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 will be 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 is a `VirtualMachineFileInfo` 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 `vm/vmcreate.pl` utility application. See the scripts in the `apps` and `samples` directories for examples of both 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 VI Perl Toolkit runtime maps server-side operations to client-side Perl view object methods. For each operation defined on a server managed object, the VI Perl Toolkit creates a corresponding view method when it creates the view object.

By default, all server-side operations available in the VI API are *non-blocking* operations listed in the *VI API Reference Guide* (`<opname>_Task()` method). The VI Perl Toolkit also provides a blocking (synchronous) method (`<opname>()` method) that provides the same functionality (as `<opname>_Task()`) but does not return a reference to a task object.

Thus, the VI Perl Toolkit provides both blocking and non-blocking methods. If you see an `<opname>_Task()` operation in the *VI API Reference Guide*, the toolkit supports both a non-blocking and a blocking `<opname>()` method that you can use in your script.

Non-Blocking and Blocking Methods

While the server-side VI API provides primarily non-blocking operations, the VI Perl Toolkit provides both non-blocking and blocking methods for each object:

- **Non-blocking methods** – Asynchronous methods that return control to the client immediately after invocation and also return a task object to the calling program.
- **Blocking methods** – Synchronous methods that fully process the invoked operation before returning control to the client script.

Each approach has advantages and disadvantages. For example, if you use a blocking methods, you don't have to handle a task object with additional code. Non-blocking methods allow you to do the following:

- Monitor progress (of the `*_Task` object) outside the main processing logic of the script, which may be especially useful during potentially long-running operations, such as creating virtual machines.
- Interleave local (client-side) processing and server-side processing.

Examples of Operations

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

Table 2-3. Examples for Asynchronous and Synchronous Methods

VI API, VI Perl Toolkit	VI Perl Toolkit Only
Non-blocking (asynchronous)	Blocking (synchronous)
<code>PowerOnVM_Task()</code>	<code>PowerOnVM()</code>
<code>CloneVM_Task()</code>	<code>CloneVM()</code>
<code>SuspendVM_Task()</code>	<code>SuspendVM()</code>

See the *VI API Reference Guide* for lists of all methods for each managed object.

Calling Methods

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

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

The type of parameter you need to pass to the method depends on the operation defined in the VI API. It might be a simple type, data object, or managed object reference.

For information about specific parameters and data types, see the *VI API Reference Guide*.

Blocking operations are invoked 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 VI Perl Toolkit 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 VI Perl Toolkit 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)
```

Omitting Optional Arguments in Method Calls

When you call a VI API Method using VI Perl Toolkit, 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 in cases where the value of an argument, which might or might not be `undef`, is contained in a variable:

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

NOTE You *cannot* use the empty string or the value 0.

Updating View Objects

In any view, the properties' values 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 be changing constantly. If your client script depends on the server being in a particular state (`poweredOn` or `poweredOff`, for example) then you can either check the state explicitly before you work with the corresponding view object, or you can refresh the view object's state. You can use the VI Perl Toolkit `Vim::update_view_data()` subroutine to refresh the values of client-side views with server-side values. [Example 2-4](#) uses `Vim::update_view_data()` to refresh a view's data.

Example 2-4. 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 VI Perl Toolkit Scripts

This chapter discusses some useful programming techniques for your VI Perl Toolkit script in the following sections:

- [“Creating and Using Filters”](#) on page 29
- [“Retrieving the ServiceInstance Object on the ESX Server Host”](#) on page 31
- [“Saving and Using Sessions”](#) on page 31
- [“Learning About Object Structure Using Data::Dumper”](#) on page 31

Creating and Using Filters

The VI Perl Toolkit allows you to define and use filters to select specific objects based on property values. Filters are helpful because they let you 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()`

When you call `Vim::find_entity_view()` or `Vim::find_entity_views()`, the toolkit might take a while to retrieve all desired objects. For example, retrieving all virtual machines in a datacenter might take a long time. You can save time if, for example, you obtain only those virtual machine objects whose names begin with a certain prefix. You can apply one or more filters to the `Vim::find_entity_view()` and `Vim::find_entity_views()` subroutines 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 it 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 regex, or `Regexp`) object. In this case, the value of the property must match the regular expression. The following filter parameter that 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 about the `qr//` operator, see the `perlre` (perl regular expressions) and `perllop` 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, so the example returns only virtual machines that fulfill both requirements:

- Guest operating system is Windows — the `config.guestFullName` property includes the string `Windows`.
- Virtual machine is running — 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";
}
. . .

```

NOTE 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 can't specify the `runtime` property of a `VirtualMachine` object, which is a complex object rather than a string.

Using Filters on the Utility Application Command Line

When you invoke 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 no longer accepted and regular expression metacharacters do not need to be escaped. For example:

```
perl hostinfo.pl --username Administrator --password 'secret' --server myserver --hostname
    santa.clara
```

Matches only the host `santa.clara`, not the hosts `santa-clara` or `santa$clara`.

When you invoke a utility applications, you must escape the characters forward slash (/), backward slash (\), and percent (%) as `%2f`, `%5c`, and `%25` respectively when they appear in virtual machine names. Elsewhere, you do not have to escape these characters. You should single-quote percent (%) on Unix-like command lines. For example, to search for the virtual machine `San-Jose/5`, execute this command:

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

Retrieving the ServiceInstance Object on the ESX Server Host

You can retrieve the `ServiceInstance` object to access the `ServiceContent` or to retrieve the current time on an ESX Server host.

If you want to retrieve the current time on an ESX Server host, 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 ESX Server host time

- 1 Connect to the ESX Server 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 VI Perl Toolkit library includes several subroutines that let you save and reuse sessions, enabling you to maintain sessions across scripts. Using sessions can also enhance security: Instead of storing passwords in scripts, you can invoke the `Util::connect()` subroutine in your script using the name of the session file. The session file does not expose password information.

To save a session, call `Vim::save_session()`, passing in a filename as a hash. You can then use the session file in another script.

```
$session1->login(
    user_name => 'user',
    password => 'secret');
$session1 = Vim::save_session (session_file => $filename);

$session2 = Vim->new(service_url => 'https://<hostname2>/sdk');
$session2->login(
    user_name => 'user',
    password => 'secret');
```

You can then use the session object to reference all functions.

```
$session1->find_entity_views
```

To use sessions, you must first log in to the server using the `Util::connect()` subroutines.

The session remains active until either a log out or disconnect operation is invoked, or until it times out (after 30 minutes, by default).

Learning About Object Structure Using `Data::Dumper`

The VI Perl Toolkit transparently uses the `Data::Dumper` Perl module (a standard library) to create the client-side view objects. [Example 3-3](#) illustrates how you can use `Data::Dumper` to learn more about these Perl objects and how they relate to the VI object model.

Lines 12 through 14 set the following parameters of `Data::Dumper` to make it easier to understand the output:

- `Sortkeys` orders the name-value pairs alphabetically by name.
- `Deepcopy` enables deep copying of structures. In general, 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 3-3. 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 execute the entire program, it produces copious output. The output from line 23 looks as follows:

```

$VAR1 = bless( {
    'apiType' => 'HostAgent',
    'apiVersion' => '2.0.0',
    '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 out 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`.

The output above also shows the content of the properties of object referred to by the `product` property of the `config` property of the `summary` property of the `HostSystem` managed object of this server. Perl's `Dumper` module writes out the object in a form that can be used with `eval` to retrieve a copy of the original structure. The `bless` keyword indicates that a Perl object follows (in parentheses).

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

The `config` property has more values than just those printed by line 23. Line 25 prints out the entire `config` object. Inside the `config` object printed by line 25 (in [Example 3-3](#)), 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' => '2.0.0',
    '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 3-3](#) prints the structure of the entire `summary` object of the host view. There are 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.

VI Perl Toolkit Subroutine Reference

This chapter lists subroutines available in the VI Perl Toolkit runtime modules and libraries. Click any link for more information.

	Subroutine	Description
Opts	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 the 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, environment variable, or configuration file and transforms into appropriate data structures for validation.
	validate	Ensures that input values are complete, consistent, and valid.
	usage	Displays a help text message associated with the option.
Util	connect	Establishes a session 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.
	clear_session	Terminates the current session loaded by the <code>load_session()</code> subroutine.
Vim	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 <code>ServiceInstance</code> object, which can be used to query the server time or to retrieve the <code>ServiceContent</code> 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 VMware Infrastructure host.
	logout	Disconnects the client from the server and closes the connection to the Web service.
	save_session	Save a session cookie, which is a text file.
	update_view_data	Refreshes the property values of a view object.

Packages

The subroutines are available in these three packages:

- The `Opts` package includes subroutines for handling built-in options and creating custom options. See [“Subroutines in the Opts Package”](#) on page 36.
- The `Util` package includes subroutines to facilitate routine tasks, such as setting up and closing connections to the server. See [“Subroutines in the Util Package”](#) on page 37.
- The `Vim` package includes subroutines for accessing server-side managed objects, instantiating local view objects, updating properties, and invoking local methods to effect operations on remote operations.

Subroutines in the Opts Package

The `Opts` package includes the following subroutines:

- [“add_options”](#) on page 36
- [“get_option”](#) on page 36
- [“option_is_set”](#) on page 36
- [“parse”](#) on page 37
- [“validate”](#) on page 37
- [“usage”](#) on page 37

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.

Parameters

`%opts` – Name of the hash variable that comprises the option name and its attributes.

This subroutine returns nothing.

get_option

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

Parameters

`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 command line or whether the option has a default or computed value (that is, the return value of a `func`).

Parameters

`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 [“VI Perl Toolkit Programming Conventions”](#) on page 9.

parse

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

This subroutine takes no parameters.

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

validate

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

This subroutine takes no parameters.

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

usage

Displays a help text message associated with the option.

This subroutine takes no parameters and returns nothing.

Subroutines in the Util Package

The Util package includes the following subroutines:

- [“connect”](#) on page 37
- [“disconnect”](#) on page 37
- [“get_inventory_path”](#) on page 38
- [“trace”](#) on page 38

connect

Establishes a session with the VirtualCenter Server or ESX Server 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.

This subroutine returns nothing.

Parameters

- `user_name` - User account on the ESX Server or VirtualCenter 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.

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.

This subroutine has no parameters and returns nothing.

get_inventory_path

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

Parameter

`view` - Managed entity view.

Returns

String that identifies the inventory path of the managed entity.

trace

General-purpose logging function used in conjunction with the `--verbose` command-line option. If not specified, the default log level is 0. Passing the `--verbose` flag without a value sets the level to 1.

This subroutine returns nothing.

- `logLevel` - Numeric value. If not specified, default is 0.
- `message` - A string that will be printed for the associated loglevel value.

Subroutines in the Vim Package

The Vim package includes the following subroutines:

- `"clear_session"` on page 38
- `"find_entity_view"` on page 38
- `"find_entity_views"` on page 39
- `"get_service_instance"` on page 39
- `"get_service_content"` on page 40
- `"get_session_id"` on page 40
- `"get_view"` on page 40
- `"get_views"` on page 40
- `"load_session"` on page 40
- `"login"` on page 41
- `"logout"` on page 41
- `"save_session"` on page 41
- `"update_view_data"` on page 41

clear_session

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

This subroutine takes no parameters and 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.

Parameters

- `view_type` - Managed entity type specified as one of these strings:
 - “ClusterComputeResource”
 - “ComputeResource”
 - “Datacenter”
 - “Folder”
 - “HostSystem”
 - “ResourcePool”
 - “VirtualMachine”
- `begin_entity` (optional) – Managed object reference that specifies the starting point for the search (in the context of the inventory). This helps you narrow the scope.
- `filter` (optional) – A hash of one or more name-value pairs in which the name represents the property value to test and the value represents a pattern the property must match. If more than one pair is present, all the patterns must match.

Returns

Reference to a view object containing the same properties as the managed entity. If there is more than one managed entity that 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.

Parameters

- `view_type` - Managed entity type specified as one of these strings:
 - “ClusterComputeResource”
 - “ComputeResource”
 - “Datacenter”
 - “Folder”
 - “HostSystem”
 - “ResourcePool”
 - “VirtualMachine”
- `begin_entity` (optional) – Managed object reference that specifies starting point for search (in context of inventory).
- `filter` (optional) – A hash of one or more name-value pairs, in which the name represents the property value to test and the value represents a pattern the property must match. If more than one pair is present, all the patterns must match.

Returns

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

get_service_instance

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

This subroutine takes no parameters.

Returns

Returns a `ServiceInstance` object.

get_service_content

Retrieves properties of the service instance enabling access to the service's managed objects. 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 need to navigate to the object of interest.

This subroutine has no parameters.

Returns

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

get_session_id

Allows you to retrieve the session ID corresponding to the current session.

This subroutine has no parameters.

Returns

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

get_view

Retrieves the properties of a single managed object.

Parameters

- `mo_ref` – A managed object reference, obtained from a property of another managed object or a view.
- `view_type` (optional) – The 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

A view object containing static copies of the property values for a managed object.

get_views

Retrieves the properties of a set of managed objects.

Parameters

- `mo_ref_array` – A reference to an array of managed object references.
- `view_type` (optional) – The 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

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

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.

Returns

Returns the `Vim` object instance.

Parameters

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

Example

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

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

login

Establishes a session with the Web service running on the VirtualCenter Server or ESX Server host using the credentials (user name, password) provided via command-line, environment variables, or configuration file.

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

Returns

Returns the `Vim` object instance.

Parameters

- `service_url` - The URL of the server to which the client connects.
- `user_name` - User account on the ESX Server or VirtualCenter Server system.
- `password` - Password for the user account.

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()`.

This subroutine takes no parameters and returns nothing.

save_session

Allows you to save a session cookie, which is a text file. If you call a VI Perl Script and pass the session cookie another application reads and uses that cookie, the application is considered authenticated using the credentials that were used during session cookie creation.

This subroutine returns nothing.

Parameters

`session_file` - Full path and filename where the token should be saved. Times out after 30 minutes.

NOTE You pass in the filename as a hash.

Example

`save_session (session_file => $filename);`

update_view_data

Refreshes the property values of a view object.

This subroutines takes no parameters and returns nothing.

Glossary

A **appliance**

See virtual appliance.

D **data object**

Complex data type that consists of properties and values only (no operations or methods). Data objects are used throughout the VI API to capture or reflect the state of various properties of managed objects. As implemented in the VI API, data objects are analogous to structures (structs) in C, C++, and other programming languages.

H **hash**

One or more key-value pairs that define the attributes and their values.

I **inventory**

The collection of all managed entities on the server, that is, of all instances of `HostSystem`, `Datacenter`, `VirtualMachine`, `ResourcePool`, `ComputeResource`, `ClusterComputeResource`, and `Folder`.

M **managed entity**

One of the seven managed object types that extends the `ManagedEntity` managed object.

The `ManagedEntity` managed object type is an abstract class that defines the base properties and methods for VMware Infrastructure objects, the same kinds of manageable components found in a physical IT infrastructure, such as datacenters and hosts.

managed object

A server-side type that encapsulates properties and operations available on the server. Different managed objects offer different services (operations, methods). From the highest level, the various managed object types on the server define common administrative and management services one would expect to use in a typical datacenter, 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`).

managed object reference

A type of data object that enables distributed computing for the VMware Infrastructure environment. A managed object reference identifies a specific managed object on the server, and encapsulates the state and methods of server-side objects, making them available to client applications. Clients invoke methods (operations) on the server by passing the appropriate managed object reference (`mo_ref`) to the server, in the method invocation.

MOB (Managed Object Browser)

A Web-based application hosted on all VMware ESX Server hosts and VirtualCenter Server systems. The MOB lets you explore the objects on the system and obtain information about each object's properties and methods.

V VI API

A set of Web services, hosted on ESX Server and VirtualCenter Server host systems, that provides interfaces to VMware Infrastructure components such as hosts, virtual machines, and data centers and operations on these components.

view

A client-side Perl object that the VI Perl Toolkit has populated with the state of one or more server-side managed objects. Client applications and scripts work with view objects rather than with the managed entities that exist on the server. To create a view, call the appropriate VI Perl Toolkit subroutine (`Vim::get_view`, `Vim::get_views`, and so on) with the managed object reference for the entity of interest.

virtual appliance

A virtual appliance is a virtual machine that is prepackaged with an operating system and a set of applications.

virtualization

Separation of a resource- or service-request from the underlying physical delivery of that service.

Virtualization provides an abstraction layer between computing resources, physical storage, networking hardware and the applications that use these resources. Virtualization can greatly enhance the computing environment, optimizing the use of available physical components. For example, virtual memory enables computer software to use more memory than is physically installed, via the background swapping of data to disk storage. Virtualization techniques can be applied to all layers of an IT infrastructure such as networks, storage, laptop or server hardware, operating systems, and applications.

VI SDK

The package of components (WSDL, sample code, and other artifacts) required for developing Java, C#, or other Web-services-enabled client applications that invoke operations on the Web-services-based VI API.