OpenLMI Documentation

Release latest

OpenLMI authors

October 08, 2014

Contents

OpenLMI = Open Linux Management Infrastructure.

OpenLMI is open-source project aiming to improve management of Linux systems using WBEM standards. We reuse many already available open-source WBEM components, adding the missing ones and integrating them into one system management solution.

In short, WBEM can be described as a *remote API for system management*. See WBEM overview for details.

CHAPTER 1

Client components

There are many already existing tools and libraries to manage WBEM-enabled hosts. see WBEM overview for details.

OpenLMI project adds LMI metacommand and LMIShell.

1.1 *LMI metacommand*

A command line utility to perform discovery and operations on remote managed systems. For example, it can start a printing service on remote system:

```
/usr/bin/lmi -h my.server.org service start cups
```

LMI metacommand users do not need to know anything about WBEM, all the complexity is hidden inside.

1.2 *LMIShell*

A high-level python-based WBEM client, that can be used for scripting or as an interactive shell to manage remote systems. For example, one can write a script to start a service:

```
c = connect("my.server.org", "root", "opensesame")
cups = c.root.cimv2.LMI_Service.first_instance({"Name" : "cups.service"})
cups.StartService()
```

LMIShell users do not need to know anything about WBEM transport protocols, however some knowledge about the aforementioned remote API becomes necessary. In the example above, the script author must know that system services are exposed as instances of *LMI_Service* class with property *Name* (the service name) and method *StartService()* that starts the service.

CHAPTER 2

Server components

OpenLMI focuses on implementation of missing providers for networking, storage, system services, packages and so on.

CHAPTER 3

Table of Contents

3.1 OpenLMI client components

They consist of several client-side python utilities and libraries. We refer to them as *OpenLMI Tools* and ship them under obvious name *openlmi-tools*.

Currently they contain *LMI metacommand* and *LMIShell*.

3.1.1 LMI metacommand

Is a command line interface for OpenLMI Providers sitting on top of *LMIShell*. It provides an easy to use interface for system management through modular commands. These dynamically extend the functionality of *LMI metacommand*.

Short example:

```
$ lmi -h myhost.example.org storage fs create --label=opt-root ext4 /dev/vda5
```

Usage

LMI metacommand is a command line utility build on top of client-side libraries. It can not do much on its own. Its functionality is extended by commands that are installed separately. Each command operates on a set of providers that need to be installed on managed machine. Commands can be invoked directly from shell or within interactive mode.

Running from command line

It can run single command given on command line like this:

```
lmi -h ${hostname} service list --all
```

Getting help

For detailed help run:

```
lmi --help
```

To get a list of available commands with short descriptions:

```
lmi help
```

For help on a particular registered command:

```
lmi help service
```

Interactive mode

Or it can be run in interactive mode when command is omitted:

```
$ lmi -h ${hostname}
lmi> help
...
lmi> sw search django
...
lmi> sw install python-django
...
lmi> exit
```

`help` command is always your good friend. Following two lines gets you the same help message:

```
lmi> help storage raid
...
lmi> storage raid --help
...
```

Built-in commands Interactive mode comes with few special commands not available from command line. To get their list, type:

```
lmi> : help
```

They are prefixed with `:` and optional space. Currently only namespace nesting commands are supported. Those are `:cd`, `:..` and `:pwd`.

They work as expected:

```
lmi> :pwd                       # top-level namespace
/lmi
lmi> :cd storage                # you can do storage specific stuff here
>storage> :pwd
/lmi/storage
>storage> :cd raid              # we don't care about anything but raid
>>raid> :pwd
/lmi/storage/raid
>>raid> :cd /lmi/sw             # let's manage packages now
>sw> :..
lmi>
```

Static commands Aren't prepended with `:` and except for `help` are again available only in interactive mode.

EOF	Same as hitting `^D`. If some nested into some command's namespace, it will map to `:cd ..` and parent namespace will become active. If the top-level namespace is active, program will exit.
exit	Exits immediately. It accepts optional exit code as an argument.
help	Lists available commands. Accepts command path as an optional argument.

Extending *metacommand*

In order to make the *LMI metacommand* useful, you'll need to install some commands. If you run Fedora, the easiest way to get them is with your favorite package manager:

```
sudo dnf install 'openlmi-scripts-*'
```

Note: On *RHEL* you'll need to add EPEL to your repositories before installing them with *yum*.

They will be automatically discovered by *LMI metacommand*. You can ensure their presence with this simple test:

```
$ lmi help
Commands:
  file     - File and directory management functions.
  group    - POSIX group information and management.
  help     - Print the list of supported commands with short description.
  hwinfo   - Display hardware information.
  journald - Test for provider version requirements
  locale   - System locale management.
  net      - Networking service management.
  power    - System power state management.
  service  - System service management.
  sssd     - SSSD system service management.
  storage  - Basic storage device information.
  sw       - System software management.
  system   - Display general system information.
  user     - POSIX user information and management.

For more informations about particular command type:
    help <command>
```

As Python eggs They may be installed on any distribution. Go for them also if you want to be more up to date. They are available for download from PyPI. The easiest way to install them is with *pip* (shipped with *python-pip* package):

```
pip search openlmi-scripts
pip install --user openlmi-scripts-{hardware,system,service,storage}
```

Bleeding edge Commands are available from our git repository. Follow instructions there to install the most up to date versions.

Documentation Check out documentation of currently implemented commands.

- *Account command line reference*
- *Hardware command line reference*
- *Journald command line reference*
- *Locale command line reference*
- *Logical File command line reference*
- *Networking command line reference*
- *Power Management command line reference*
- *Realmd command line reference*

- *Service command line reference*
- *Software command line reference*
- *SSSD command line reference*
- *Storage command line reference*
- *System command line reference*

Configuration

LMI metacommand has the main configuration file located in:

`/etc/openlmi/scripts/lmi.conf`

User can have his own configuration file taking precedence over anything in global one above:

`$HOME/.lmirc`

Configuration is written in INI-like configuration files. Please refer to ConfigParser's documentation for details.

Follows a list of sections with their list of options. Most of the options listed here can be overridden with command line parameters.

See also:

`configuration`

Section [Main]

CommandNamespace [`string`] Python namespace, where command entry points will be searched for.

Defaults to `lmi.scripts.cmd`.

Trace [`boolean`] Whether the exceptions should be logged with tracebacks.

Defaults to `False`.

Can be overridden with `--trace` and `--notrace` options on command-line.

Note: For most exceptions generated by scripts a *Verbosity* option needs to be highest as well for tracebacks to be printed.

Verbosity: **`integer`** A number within range -1 to 2 saying, how verbose the output shall be. This differs from log_level, which controls the logging messages written to file. If logging to console is enabled it sets the minimum severity level. -1 Suppresses all messages except for errors. 0 shows warnings, 1 info messages and 2 enables debug messages. This option also affects the verbosity of commands, making them print more information to `stdout`.

Defaults to 0.

Can be overridden with `-v` and `-q` flags on command-line.

Section [CIM]

Namespace [`string`] Allows to override default CIM namespace, which will be passed to script library functions.

Defaults to `root/cimv2`.

Section [SSL]

VerifyServerCertificate [boolean] Whether to verify server-side certificate, when making secured connection over https.

Defaults to True.

Can be overridden with -n | --noverify flag on command-line.

Section [Format]

HumanFriendly [boolean] Whether to print values in human readable forms (e.g. with units).

Defaults to False.

Can be overridden with -H | --human-frienly flag on command-line.

ListerFormat [one of {csv, table}] What format to use, when listing tabular data. csv format allows for easy machine parsing, the second one is more human friendly.

Defaults to table.

Can be overridden with -L | --lister-format option on command line.

NoHeadings [boolean] Whether to suppress headings (column names) when printing tables.

Defaults to False.

Can be overridden with -N | --no-headings option on command line.

Section [Log]

Level [one of {DEBUG, INFO, WARNING, ERROR, CRITICAL}] Minimal severity level of messages to log. Affects only logging to a file. See the main_verbosity option controlling console logging level.

Defaults to ERROR.

LogToConsole [boolean] Whether the logging to console is enabled.

Defaults to True

On command-line the same could be achieved by redirecting stderr to /dev/null.

ConsoleFormat [string] Format string used when logging to a console. This applies to warnings and more severe messages. Refer to *Format String* in python's documentation for details.

Defaults to %(levelname)s: %(message)s.

ConsoleInfoFormat [string] Format string used when logging to a console. Applies to info and debug messages. Refer to *Format String* in python's documentation for details.

Defaults to %(message)s.

FileFormat [string] Format string used, when logging to a console. This applies only when *OutputFile* is set (see below). Refer to *Format String* in python's documentation for details.

Defaults to %(asctime)s:%(levelname)-8s:%(name)s:%(lineno)d - %(message)s

OutputFile [string] Allows to set a path to file, where messages will be logged. No log file is written at default.

Defaults to empty string.

Can be overridden on command line with --log-file option.

Account command line reference

These commands allow to query and manage users and groups.

user

POSIX user information and management.

Usage:

lmi user list

lmi user show [<*user*> ...]

lmi user create <*name*> [options]

lmi user delete [**–no-delete-home**] [**–no-delete-group**] [**–force**] <*user*> ...

Commands:

list Prints a list of users.

show Show detailed information about user. If no users are provided, all of them are displayed.

create Creates a new user. See Create options below for options description.

delete Delete specified user (or user list). See Delete options below for options description.

Create options:

-c gecos, –gecos=gecos Set the GECOS field to gecos.

-d dir, –directory=dir Set the user's home directory to dir. If this option is not set, a default value is used.

-s shell, –shell=shell Set user's login shell to shell. If this option is not set, a default value is used.

-u uid, –uid=uid Use user ID uid for the newly created user. If this option is not set, a default value is used.

-g gid, –gid=gid Set user's primary group ID to gid. If this option is not set, a default value is used.

-r, –reserved The user is a system user. Implies the -M option.

-M, –no-user-home Don't create a home directory.

-n, –no-user-group Don't create a primary group for user.

-p, –password=pwd Set user's password to 'pwd'.

-P, –plain-password If set, the password set in '-p' parameter is plain text. Otherwise, it is already encrypted by supported hash algorithm. See crypt(3).

Delete options:

–no-delete-home Do not remove home directory.

–no-delete-group Do not remove users primary group.

–force Remove home directory even if the user is not owner.

group

POSIX group information and management.

Usage:

lmi group list [<*group*> ...]

lmi group create [**–reserved**] [**–gid**=gid] <*group*>

lmi group delete <*group*>

lmi group listuser [<*group*>] ...

lmi group adduser <*group*> <*user*> ...

lmi group removeuser <*group*> <*user*> ...

Commands:

list List groups. If no groups are given, all are listed.

create Creates a new group.

delete Deletes a group.

listuser List a users in a group or in a list of groups.

adduser Adds a user or a list of users to the group.

removeuser Removes a user or a list of users from the group.

Options:

-r, –reserved Create a system group.

-g, –gid=gid GID for a new group.

Hardware command line reference

This command can display various hardware information.

hwinfo

Display hardware information.

Usage:

lmi hwinfo [all]

lmi hwinfo system

lmi hwinfo motherboard

lmi hwinfo cpu

lmi hwinfo memory

lmi hwinfo disks

Commands:

all Display all available information.

system Display system information.

motherboard Display motherboard information.

cpu Display processor information.

memory Display memory information.

disks Display disks information.

Journald command line reference

This command allows to query and watch system logs through journald service. It can also log custom messages.

journald

Journald message log management.

Usage:

lmi journald list [(**–reverse** | **–tail**)]

lmi journald logger <*message*>

lmi journald watch

Commands:

list Lists messages logged in the journal

logger Logs a new message in the journal

watch Watch for newly logged messages

Options:

–reverse List messages from newest to oldest

–tail List only the last 50 messages

Locale command line reference

This command allows to display and set system locale.

locale

System locale management.

Usage:

lmi locale show [(**–locale** | **–vc-keyboard** | **–x11-keymap**)]

lmi locale set-locale (<*locale*> <*value*>) ...

lmi locale set-vc-keyboard [**–convert**] <*keymap*> [<*keymap-toggle*>]

lmi locale set-x11-keymap [**–convert**] <*layouts*> [<*model*> <*variant*> <*options*>]

Commands:

show Show detailed information about system locale cathegory (locale variables, key mapping on the virtual console, default key mapping of the X11 server). If no cathegory is provided via option, all locale information is displayed.

set-locale Set locale variables.

set-vc-keyboard Set the key mapping on the virtual console.

set-x11-keymap Set the default key mapping of the X11 server.

Show options:

–locale Display locale variables.

–vc-keyboard Display key mapping on the virtual console.

–x11-keymap Display default key mapping of the X11 server.

Set options:

–convert Try to set the nearest console keyboard/X11 keyboard setting for the chosen X11 keyboard/console keyboard setting.

Logical File command line reference

This command allows to query file system structure. It can also create and delete empty directories – mount points.

file

File and directory management functions.

Usage:

lmi file list <*directory*> [<*depth*>]

lmi file createdir <*directory*>

lmi file deletedir <*directory*>

lmi file show <*target*>

Commands:

list List a directory. When depth is specified, at most depth levels will be listed recursively.

The files and directories are listed in a tree-like structure.

Possible listed file types are:

- F : Regular data file.
- Dev : Device file. Can be either block or character device.
- Dir : Directory.
- P : Pipe file.
- L : Symbolic link.
- S : Unix socket.

createdir Create a directory. The parent directory must exist.

deletedir Delete a directory. The directory must be empty.

show Show detailed information about target. Target can be any file on the remote system.

Networking command line reference

This command allows to manage networking devices and their configuration.

net

Networking service management.

Usage:

lmi net device (**–help** | **show** [<*device_name*> ...] | **list** [<*device_name*> ...])

lmi net setting (**–help** | <*operation*> [<*args*>...])

lmi net activate <*caption*> [<*device_name*>]

lmi net deactivate <*caption*> [<*device_name*>]

lmi net enslave <*master_caption*> <*device_name*>

lmi net address (**–help** | <*operation*> [<*args*>...])

lmi net route (**–help** | <*operation*> [<*args*>...])

lmi net dns (**–help** | <*operation*> [<*args*>...])

Commands:

device Display information about network devices.

setting Manage the network settings.

activate Activate setting on given network device.

deactivate Deactivate the setting.

enslave Create new slave setting.

address Manipulate the list of IP addresses on given setting.

route Manipulate the list of static routes on given setting.

dns Manipulate the list of DNS servers on given setting.

Power Management command line reference

This command allows to display and control system power states.

power

System power state management.

Usage:

lmi power list

lmi power suspend

lmi power hibernate

lmi power reboot [**–force**]

lmi power poweroff [**–force**]

Commands:

list Prints a list of available power states.

suspend Suspend the system (suspend to RAM).

hibernate Hibernate the system (suspend to disk).

reboot Shutdown and reboot the system (–force will skip shutdown of running services).

poweroff Shutdown the system (–force will skip shutdown of running services).

Options:

–force Skip shutting down services first

Realmd command line reference

This command allows to join or leave AD or Kerberos domain.

realmd

Manage AD or Kerberos domain membership.

Usage:

lmi realmd [show]

lmi realmd join -u <*user*> [**-p** <*password*>] **-d** <*domain*>

lmi realmd leave -u <*user*> [**-p** <*password*>] **-d** <*domain*>

Commands:

show Show joined domain.

join Join the given domain.

leave Leave the given domain.

Options:

-u, –user The username to be used when authenticating to the domain.

-p, –password Optional password for the authentication. If omitted you will be prompted for one.

-d, –domain The domain to be joined/left.

Service command line reference

This command allows to list and manage system services.

service

System service management.

Usage:

lmi service list [(**–enabled** | **–disabled**)]

lmi service show <*service*>

lmi service start <*service*>

lmi service stop <*service*>

lmi service enable <*service*>

lmi service disable <*service*>

lmi service restart [**–try**] <*service*>

lmi service reload <*service*>

lmi service reload-or-restart [**–try**] <*service*>

Commands:

list Prints a list of services. Only enabled services are printed at default.

show Show detailed information about service.

start Starts a service.

stop Stops the service.

restart Restarts the service.

reload Ask the service to reload its configuration.

reload-or-restart

Reload the service if it supports it. If not, restart it instead.

Options:

–enabled List only enabled services.

–disabled List only disabled services.

–try Whether to abandon the operation if the service is not running.

Software command line reference

This command allows to list and manage rpm packages and repositories.

sw

System software management.

Usage:

lmi sw search [(**–repoid** <*repository*>)] [**–allow-duplicates**] <*package*>...

lmi sw list (**–help** | <*what*> [<*args*>...])

lmi sw show (**–help** | <*what*> [<*args*>...])

lmi sw install [**–force**] [**–repoid** <*repository*>] <*package*> ...

lmi sw install –uri <*uri*>

lmi sw update [**–force**] [**–repoid** <*repository*>] <*package*> ...

lmi sw remove <*package*> ...

lmi sw verify <*package*> ...

lmi sw enable <*repository*> ...

lmi sw disable <*repository*> ...

Commands:

list List various information about packages, repositories or files.

show Show detailed informations about package or repository.

install Install packages on system. See below, how package can be specified. Installation from URI is also supported, it must be prefixed with –uri option.

update Update package.

remove Remove installed package.

verify Verify package. Files that did not pass the verification are listed prefixed with a sequence of characters, each representing particular attribute, that failed. Those are:

- S file Size differs
- M Mode differs (includes permissions and file type)
- 5 digest (formerly MD5 sum) differs
- D Device major/minor number mismatch
- L readLink(2) path mismatch
- U User ownership differs
- G Group ownership differs
- T mTime differs
- P caPabilities differ

enable Enable one or more repositories.

disable Disable one or more repositories.

Options:

–force Force installation. This allows to install package already installed – make a reinstallation or to downgrade package to older version.

–repoid <repository>

Select a repository, where the given package will be searched for.

–uri <uri> Operate upon an rpm package available on remote system through http or ftp service.

–installed Limit the query only on installed packages.

–help Get a detailed help for subcommand.

Specifying <package>:

Package can be given in one of following notations:

- <name>
- <name>.<arch>
- <name>-<version>-<release>.<arch> # nvra
- <name>-<epoch>:<version>-<release>.<arch> # nevra

- <epoch>:<name>-<version>-<release>.<arch> # envra

Bottom most notations allow to precisely identify particular package.

sw list

List packages, repositories or files.

Usage:

lmi sw list [all] [**–allow-duplicates**]

lmi sw list installed

lmi sw list available [**–repoid** <*repository*>] [**–allow-duplicates**]

lmi sw list repos [**–disabled** | **–all**]

lmi sw list files [**-t** <*file_type*>] <*package*>

Commands:

all

- List installed and available packages.

installed

- List installed packages.

available

- List available packages.

repos

- List repositories. Only enabled ones are listed by default.

files

- List files belonging to a package.

Options:

–allow-duplicates Print all possible versions of package found. Normally only the newest version is shown.

–repoid <repository> List just packages available in given <repository>.

–all List all repositories.

–disabled List only disabled repositories.

-t –type (file | directory | device | symlink | fifo)

List only particular file type.

sw show

Show details of package or repository.

Usage:

lmi sw show pkg [**–installed** | **–repoid** <*repository*>] <*package*>

lmi sw show repo <*repository*>

Options:

–installed Do not search available packages. This speeds up the operation when only installed packages shall be queried.

–repoid <repository> Search just this repository.

SSSD command line reference

This command allows to manage SSSD service.

sssd

SSSD system service management.

Usage:

lmi sssd status

lmi sssd restart [**–try**]

lmi sssd set-debug-level <*level*> [**–until-restart**] [options]

lmi sssd service

lmi sssd domain

Commands:

status Prints SSSD service's status.

restart Restarts the SSSD service.

set-debug-level Set debug level of selected (all by default) components.

service Manage supported services.

domain Manage SSSD domains.

Restart options:

–try Whether to abandon the operation if the service is not running.

Set-debug-level options:

–until-restart

Set the debug level but switch it to original value when SSSD is restarted.

–all Select all components (default)

–monitor Select the SSSD monitor.

–services=svc,...

Comma separated list of SSSD services.

–domains=dom,...

Comma separated list of SSSD domains.

Storage command line reference

`lmi storage` is a command for *LMI metacommand*, which allows listing and manipulation of storage on a remote host with installed *OpenLMI storage provider*.

Available commands:

lmi storage Generic information about storage devices.

lmi storage fs Filesystem and other data format management.

lmi storage luks LUKS management.

lmi storage lv Logical Volume management.

lmi storage mount Mount management.

lmi storage partition Partition management.

lmi storage partition-table Partition table management.

lmi storage raid MD RAID management.

lmi storage vg Volume Group management.

lmi storage thinpool Thin Pool management.

lmi storage thinlv Thin Logical Volume management.

Common options

- <*device*> can be specified as one of:
  - DeviceID of appropriate CIM_StorageExtent. This is internal OpenLMI ID of the device and it should be stable across system reboots.
  - Device name directly in /dev directory, such as */dev/sda*. This device name is available as *Name* property of CIM_StorageExtent.
  - Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as *ElementName* property of CIM_StorageExtent.
- <*vg*> represents name of a volume group, with or without */dev/* prefix.
- Any <*size*>, such as size of new partition or new logical volume, can be specified with 'T', 'G', 'M' or 'K' suffix, which represents appropriate unit (terabytes, gigabytes etc.) 1K (kilobyte) is 1024 of bytes. The suffix is case insensitive, i.e. 1g = 1G.

storage

Basic storage device information.

Usage:

lmi storage fs <*cmd*> [<*args*> ...]

lmi storage luks <*cmd*> [<*args*> ...]

lmi storage lv <*cmd*> [<*args*> ...]

lmi storage mount <*cmd*> [<*args*> ...]

lmi storage partition <*cmd*> [<*args*> ...]

lmi storage partition-table <*cmd*> [<*args*> ...]

lmi storage raid <*cmd*> [<*args*> ...]

lmi storage vg <*cmd*> [<*args*> ...]

lmi storage thinpool <*cmd*> [<*args*> ...]

lmi storage thinlv <*cmd*> [<*args*> ...]

lmi storage depends [**–deep**] [<*device*> ...]

lmi storage list [<*device*> ...]

lmi storage provides [**–deep**] [<*device*> ...]

lmi storage show [<*device*> ...]

lmi storage tree [<*device*>]

Commands:

fs Filesystem and other data format management.

luks LUKS management.

lv Logical Volume management.

mount Mount management.

partition Partition management.

partition-table Partition table management.

raid MD RAID management.

vg Volume Group management.

thinpool Thin Pool management.

thinlv Thin Logical Volume management.

list List short information about given device. If no devices are given, all devices are listed.

show Show detailed information about given devices. If no devices are provided, all of them are displayed.

provides Show devices, which are created from given devices (= show children of the devices).

For example, if a disk is provided, all partitions on it are returned. If 'deep' is used, all RAIDs, Volume Groups and Logical Volumes indirectly allocated from it are returned too.

depends Show devices, which are required by given devices to operate correctly (= show parents of the devices).

For example, if a Logical Volume is provided, its Volume Group is returned. If 'deep' is used, also all Physical Volumes and appropriate disk(s) are returned.

tree Show tree of devices, similar to lsblk.

If no device is provided, all devices are shown, starting with physical disks.

If a device is provided, tree starts with the device and all dependent devices are shown.

Options:

device Identifier of the device. Either one of:

- DeviceID of appropriate CIM_StorageExtent object. This is internal OpenLMI ID of the device and it should be stable across system reboots.

- Device name directly in /dev directory, such as '/dev/sda'. This device name is available as Name property of CIM_StorageExtent object.
- Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as ElementName property of CIM_StorageExtent object.

–deep Show all ancestors/children the device, not only the immediate ones.

storage fs

Filesystem and other data format management.

Usage:

lmi storage fs list [**–all**] [<*device*> ...]

lmi storage fs create [**–label**=<*label*>] <*fstype*> <*device*> ...

lmi storage fs delete <*device*> ...

lmi storage fs list-supported

Commands:

list List filesystems and other data formats (RAID metadata, ...) on given devices. If no devices are provided, all filesystems are listed. If –all option is set, all filesystem, including system ones like tmpfs, cgroups, procfs, sysfs etc are listed.

create Format device(s) with given filesystem. If more devices are given, the filesystem will span over these devices (currently supported only by btrfs).

For list of available filesystem types, see output of lmi storage fs list-supported.

delete Delete given filesystem or data format (like partition table, RAID metadata, LUKS, physical volume metadata etc) on given devices.

list-supported

List supported filesystems, which can be used as lmi storage fs create <fstype> option.

storage luks

LUKS management

Usage:

lmi storage luks list

lmi storage luks create [**-p** <*passphrase*>] <*device*>

lmi storage luks open [**-p** <*passphrase*>] <*device*> <*name*>

lmi storage luks close <*device*>

lmi storage luks addpass [**-p** <*passphrase*>] [**-n** <*new-passphrase*>] <*device*>

lmi storage luks deletepass [**-p** <*passphrase*>] <*device*>

Commands:

list List available LUKS formats and their clear-text devices (if any).

create Format given device with LUKS format. Any data on the device will be destroyed.

open Open given device formatted with LUKS and expose its clear-text data as a new block device.

close Close given device formatted with LUKS and destroy its clear-text block device.

addpass Add new passphrase to given LUKS-formatted device. Each device can have up to 8 separate passwords and any of them can be used to decrypt the device.

deletepass Remove given passphrase from LUKS-formatted device.

Common options:

-p, –passphrase=passphrase Passphrase. It will be read from the terminal, if it is not provided on command line.

-n, –new-passphrase=passphrase New passphrase. It will be read from the terminal, if it is not provided on command line.

Open options:

<device> Device with LUKS format on it.

<name> Name of the clear-text block device to create.

Close options:

<device> Device with LUKS format on it, previously opened by 'lmi storage luks open'.

storage lv

Logical Volume management.

Usage:

lmi storage lv list [<*vg*> ...]

lmi storage lv create <*vg*> <*name*> <*size*>

lmi storage lv delete <*lv*> ...

lmi storage lv show [<*lv*> ...]

Commands:

list List available logical volumes on given volume groups. If no volume groups are provided, all logical volumes are listed.

create Create a logical volume on given volume group.

delete Delete given logical volume.

show Show detailed information about given Logical Volumes. If no Logical Volumes are provided, all of them are displayed.

Options:

vg Name of the volume group, with or without */dev/* prefix.

size Size of the new logical volume, by default in bytes. 'T', 'G', 'M' or 'K' suffix can be used to specify other units (TiB, GiB, MiB and KiB) - '1K' specifies 1 KiB (= 1024 bytes). The suffix is case insensitive, i.e. 1g = 1G = 1073741824 bytes.

'E' suffix can be used to specify number of volume group extents, '100e' means 100 extents.

storage mount

Mount management.

Usage:

lmi storage mount list [**–all**] [<*target*> ...]

lmi storage mount create <*device*> <*mountpoint*> [(**-t** <*fs_type*>) (**-o** <*options*>)]

lmi storage mount delete <*target*>

lmi storage mount show [**–all**] [<*target*> ...]

Commands:

list List mounted filesystems with a device attached to them. <target> can be specified either as device names or mountpoints.

create Mount a specified device on the path given by mountpoint. Optionally, filesystem type, common options (filesystem independent) and filesystem specific options can be provided. If no filesystem type is specified, it is automatically detected.

Options can be provided as a comma-separated string of 'option_name:value' items. Possible option names are:

AllowExecution AllowMandatoryLock AllowSUID AllowUserMount AllowWrite Auto Dump FileSystemCheckOrder InterpretDevices Silent SynchronousDirectoryUpdates SynchronousIO UpdateAccessTimes UpdateDirectoryAccessTimes UpdateFullAccessTimes UpdateRelativeAccessTimes

Possible option values for all of the options except for FileSystemCheckOrder are 't', 'true', 'f', 'false'. All of them are case insensitive. The FileSystemCheckOrder option's value is a number.

In case an option is not recognized as being one of the possible options listed above, it's used as a filesystem dependent option.

Examples:

create /dev/vda1 /mnt -t ext4 -o 'AllowWrite:F,InterpretDevices:false'

create /dev/vda2 /mnt -o 'FileSystemCheckOrder:2'

create /dev/vda3 /mnt -o 'user_xattr,barrier=0'

create /dev/vda4 /mnt -o 'Dump:t, AllowMandatoryLock:t, acl'

delete Unmount a mounted filesystem. Can be specified either as a device path or a mountpoint.

show Show detailed information about mounted filesystems with a device attached to them. <target> can be specified either as device names or mountpoints. <spec>. Optionally, show all mounted filesystems.

storage partition

Partition management.

Usage:

lmi storage partition list [<*device*> ...]

lmi storage partition create [**–logical** | **–extended**] <*device*> [<*size*>]

lmi storage partition delete <*partition*> ...

lmi storage partition show [<*partition*> ...]

Commands:

list List available partitions on given devices. If no devices are provided, all partitions are listed.

create Create a partition on given device.

If no size is given, the resulting partition will occupy the largest available space on disk.

The command automatically creates extended and logical partitions using these rules:

- If no partition type (logical or extended) is provided and MS-DOS partition is requested and there is extended partition already on the device, a logical partition is created.
- If there is no extended partition on the device and there are at most two primary partitions on the device, primary partition is created.
- If there is no extended partition and three primary partitions already exist, new extended partition with all remaining space is created and a logical partition with requested size is created.

delete Delete given partitions.

show Show detailed information about given partitions. If no partitions are provided, all of them are displayed.

Options:

size Size of the new partition volume, by default in sectors. 'T', 'G', 'M' or 'K' suffix can be used to specify other units (TiB, GiB, MiB and KiB) - '1K' specifies 1 KiB (= 1024 bytes). The suffix is case insensitive, i.e. 1g = 1G = 1073741824 bytes.

device,

partition Identifier of the device/partition. Either one of:

- DeviceID of appropriate CIM_StorageExtent object. This is internal OpenLMI ID of the device and it should be stable across system reboots.
- Device name directly in /dev directory, such as '/dev/sda'. This device name is available as Name property of CIM_StorageExtent object.
- Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as ElementName property of CIM_StorageExtent object.

–logical Override the automatic behavior and request logical partition.

–extended Override the automatic behavior and request extended partition.

storage partition-table

Partition table management.

Usage:

lmi storage partition-table list [<*device*> ...]

lmi storage partition-table create [**–gpt** | **–msdos**] <*device*> ...

lmi storage partition-table show [<*device*> ...]

Commands:

list List partition tables on given device. If no devices are provided, all partition tables are listed.

create Create a partition table on given devices. The devices must be empty, i.e. must not have any partitions on them. GPT partition table is created by default.

show Show detailed information about partition table on given devices. If no devices are provided, all of them are displayed.

Options:

device Identifier of the device. Either one of:

- DeviceID of appropriate CIM_StorageExtent object. This is internal OpenLMI ID of the device and it should be stable across system reboots.
- Device name directly in /dev directory, such as '/dev/sda'. This device name is available as Name property of CIM_StorageExtent object.
- Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as ElementName property of CIM_StorageExtent object.

–gpt Create GPT partition table (default).

–msdos Create MS-DOS partition table.

storage raid

MD RAID management.

Usage:

lmi storage raid list

lmi storage raid create [**–name**=<*name*>] <*level*> <*device*> ...

lmi storage raid delete <*device*> ...

lmi storage raid show [<*device*> ...]

Commands:

list List all MD RAID devices on the system.

create Create MD RAID array with given RAID level from list of devices.

delete Delete given MD RAID devices.

show Show detailed information about given MD RAID devices. If no devices are provided, all MD RAID devices are displayed.

Options:

device Identifier of the device. Either one of:

- DeviceID of appropriate CIM_StorageExtent object. This is internal OpenLMI ID of the device and it should be stable across system reboots.
- Device name directly in /dev directory, such as '/dev/sda'. This device name is available as Name property of CIM_StorageExtent object.

- Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as ElementName property of CIM_StorageExtent object.

level RAID level. Supported levels are: 0, 1, 4, 5, 6, 10.

storage vg

Volume Group management.

Usage:

lmi storage vg list

lmi storage vg create [**–extent-size**=<*size*>] <*name*> <*device*> ...

lmi storage vg delete <*vg*> ...

lmi storage vg show [<*vg*> ...]

lmi storage vg modify <*vg*> [**–add**=<*device*>] ... [**–remove**=<*device*>] ...

Commands:

list List all volume groups on the system.

create Create Volume Group with given name from list of devices.

delete Delete given Volume Groups.

show Show detailed information about given Volume Groups. If no Volume Groups are provided, all of them are displayed.

modify Add or remove Physical Volumes to/from given Volume Group.

Options:

device Identifier of the device. Either one of:

- DeviceID of appropriate CIM_StorageExtent object. This is internal OpenLMI ID of the device and it should be stable across system reboots.
- Device name directly in /dev directory, such as '/dev/sda'. This device name is available as Name property of CIM_StorageExtent object.
- Name of MD RAID or logical volume. This method cannot be used when the name is not unique, for example when there are two logical volumes with the same name, allocated from different volume groups. This name is available as ElementName property of CIM_StorageExtent object.

vg Name of the volume group, with or without */dev/* prefix.

size Requested extent size of the new volume group, by default in bytes. 'T', 'G', 'M' or 'K' suffix can be used to specify other units (TiB, GiB, MiB and KiB) - '1K' specifies 1 KiB (=1024 bytes). The suffix is case insensitive, i.e. 1g = 1G = 1073741824 bytes.

-a <device> , –add=<device> Device to add to a Volume Group.

-r <device> , –remove=<device> Device to remove from a Volume Group.

storage thinpool

Thin Pool management.

Usage:

lmi storage thinpool list

lmi storage thinpool create <*name*> <*vg*> <*size*>

lmi storage thinpool delete <*tp*> ...

lmi storage thinpool show [<*tp*> ...]

Commands:

list List all thin pools on the system.

create Create Thin Pool with given name and size from a Volume Group.

delete Delete given Thin Pools.

show Show detailed information about given Thin Pools. If no Thin Pools are provided, all of them are displayed.

Options:

vg Name of the volume group, with or without */dev/* prefix.

tp Name of the thin pool, with or without */dev/* prefix.

size Requested extent size of the new volume group, by default in bytes. 'T', 'G', 'M' or 'K' suffix can be used to specify other units (TiB, GiB, MiB and KiB) - '1K' specifies 1 KiB (=1024 bytes). The suffix is case insensitive, i.e. 1g = 1G = 1073741824 bytes.

storage thinlv

Thin Logical Volume management.

Usage:

lmi storage thinlv list [<*tp*> ...]

lmi storage thinlv create <*tp*> <*name*> <*size*>

lmi storage thinlv delete <*tlv*> ...

lmi storage thinlv show [<*tlv*> ...]

Commands:

list List available thin logical volumes on given thin pools. If no thin pools are provided, all thin logical volumes are listed.

create Create a thin logical volume on given thin pool.

delete Delete given thin logical volume.

show Show detailed information about given Thin Logical Volumes. If no Thin Logical Volumes are provided, all of them are displayed.

Options:

tp Name of the thin pool, with or without */dev/* prefix.

size Size of the new logical volume, by default in bytes. 'T', 'G', 'M' or 'K' suffix can be used to specify other units (TiB, GiB, MiB and KiB) - '1K' specifies 1 KiB (= 1024 bytes). The suffix is case insensitive, i.e. 1g = 1G = 1073741824 bytes.

System command line reference

This command can display general system information.

system

Display general system information.

Usage:

lmi system

Command development

Do you want to write your own command? You are at the right place. As a newcomer, you should start with *Tutorial*.

Once you have your command ready, don't forget to make it public in our *repository*.

Command versus Script:

Until now we've been using *command* as a term for a subcommand of *LMI metacommand* and a package (rpm/python egg) containing it. In this documentation you'll encounter another words seemingly meaning the same. Following dictionary tries to clear out any confusion:

Term	Description
command	Either a subcommand of *LMI metacommand* or a software package containing a *script*. It may have several *subcommands*.
script	Python library utilizing *LMIShell* for instrumenting CIM providers through a CIMOM broker comming with one or more *commands* for *LMI metacommand*.
subcommand	Same as *command* used in relation to either *metacommand* or another *command*.
command wrapper	Implementation of a *command* in a *script* as a subclass of `LmiBaseCommand`.
top-level command	Direct *subcommand* of *LMI metacommand*. It appers in its help message.
end-point command	*command* without any *subcommand*. It handles command-line arguments and renders output.
command multiplexer	*command* with one or more *subcommands*. They do not handle command line arguments.
command name	Is a single word denoting *command* on a command line.
command's full name	All *command names* leading up to the *command* optionally including the `lmi`. For example in statement `lmi -h myhost.example.org storage fs create ext4 /dev/vda5` the full name of *command* `create` is `lmi storage fs create`.

Tutorial

This is a step-by-step tutorial on developing a *script* for *OpenLMI* providers. It explains how to create simple library for instrumenting *OpenLMI LogicalFile Provider*, wrap its functionality with *command wrapper* and register it as a subcommand of *LMI metacommand*.

Required knowledge You should be familiar with terms like *CIM*, *cimom*, *schema*, *provider*, *DMTF* profile. *This short tutorial* should be enough to get you started.

You should also be familiar with scripting in *python* and LMIShell which we use heavily in snippets below.

Preparation You need `tog-pegasus` *cimom* up and running with `openlmi-logicalfile` providers installed and registered on managed machine. There is a Quick Start Guide to assist you with setting it up. We will connect to it from a client which needs the following installed:

- `openlmi-python-base`
- `openlmi-tools`

Note: *RHEL* clients will also need `openlmi-scripts` installed because *LMI metacommand* is not part of *OpenLMI Tools* there.

Installing python dependencies For the first two items you may use standard rpms build for Fedora:

```
yum install openlmi-tools
```

Or you may install them to your user directory as python eggs with `pip`:

```
pip install openlmi-tools
```

Dependencies are solved for you automatically in both cases.

Note: On *RHEL* there are several possible scenarios:

1. install `openlmi-tools` as a python egg (see above)
2. install `openlmi-tools` from git (see below)
3. install both `openlmi-tools` and `openlmi-scripts` as rpms with EPEL repository enabled (for the latter package)

Make sure you don't mix above options.

Or directly from git repository. Please follow steps described there.

Setting up environment We'll stick to the process described here that lets us develop quickly without the need to reinstall anything while making changes.

First let's check out our `openlmi-scripts` repository:

```
git clone https://github.com/openlmi/openlmi-scripts.git
cd openlmi-scripts
```

Now let's set up our workspace:

```
WSP=~/.python_workspace
mkdir $WSP
# may be added to '$HOME/.profile' or '$HOME/.bashrc'
export PYTHONPATH=$WSP:$PYTHONPATH
export PATH="$PATH:$WSP"
```

Making script structure We'll use provided `commands/make_new.py` script to create the basic structure and `setup.py.skel` file:

```
cd commands
# this will ask us additional questions used to create setup.py.skel file
./make_new.py mylf
```

Because a script implementation for `OpenLMI LogicalFile` profile is already present in upstream repository (in `commands/logicalfile`), we need to name our library distinctly (e.g. `mylf`).

Following structure should be created:

[dirtree] mylf child node doc child node _build child node cmdline.rst child node conf.py.skel child node index.rst child node Makefile child node python.rst child node _static child node _templates child node lmi child node __init__.py child node scripts child node __init__.py child node mylf child node __init__.py child node Makefile child node README.md child node setup.cfg child node setup.py.skel ;

We should check that everything matches in `mylf/setup.py.skel` and correct any shortcomings.

`setup.py` is generated out of `setup.py.skel` template by running:

```
make setup
```

OpenLMI LogicalFile introduction *OpenLMI LogicalFile* is a CIM provider which provides a way to read information about files and directories. The provider also allows to traverse the file hierarchy, create and remove empty directories.

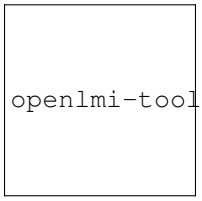

Figure 3.1: LogicalFile model

It consists mainly of few specializations of `CIM_LogicalFile` representing any type of file on filesystem, `LMI_UnixFile` holding unix specific information for each such file and association classes between them. `CIM_LogicalFile` has following key properties inherited by `LMI_*` subclasses above:

- **Name**
- **CSName**
- **CSCreationClassName**
- `FSCreationClassName`
- `CreationClassName`
- `FSName`

Only those shown in **bold** are mandatory. Others are ignored when requesting an instance of `CIM_LogicalFile`. This applies also to `LMI_UnixFile` with **Name** being replaced with **LFName**. None of the presented classes supports enumeration of instances. Only references can be obtained.

With `CreateInstance()` and `DeleteInstance()` calls issued on class/object of `LMI_UnixDirectory` we are able to create and delete directories.

Let's write some code Before writing code that actually does anything useful, we start by specifying usage string. It is a command line API. Writing it will give you a clear picture of what you're going to implement and how it will be used. Once done, all the subcommands can be implemented one by one in a straightforward way.

Writing usage string Usage string is a module's documentation, help message and a prescription for command line parser, all-in-one. Writing it is pretty straightforward. Let's put it to `mylf/lmi/scripts/mylf/cmd.py`:

```
"""
Read informations about file system structure.

Usage:
    %(cmd)s list [options] <directory>
    %(cmd)s show [-L] <file>
    %(cmd)s create <directory>
    %(cmd)s delete <directory>

Options:
    -t --type <type>   Filter listed files by their type. One of:
                       any, file, device, directory, fifo, symlink, socket.
                       Defaults to any.
    -L --dereference   Causes symlink to be followed.
"""
```

The first line provides a short description that will be shown with

```
lmi help
```

after the command is registered. Text under `Usage:` and `Options:` are parsed by docopt. It is very well readable but writing it may pose quite a challenge for the first time developer. Please refer to its documentation for more information.

Note the `%(cmd)s` string which needs to be present instead of `lmi mylf` or similar command names.

Note also spaces that separate options from their descriptions. There must be a column of spaces at least 2 characters wide. Otherwise *docopt* will treat description as a continuation of option specification.

Let's add one more snippet so we can test it:

```
from lmi.scripts.common import command

MyLF = command.register_subcommands('MyLF', __doc__, {})
```

This creates a command multiplexer without any children (we'll add them later).

And finally let's modify our `mylf/setup.py.skel` by adding entry point:

```
entry_points={
    'lmi.scripts.cmd': [
        'mylf = lmi.scripts.mylf.cmd:MyLF',
        ],
    }
```

Now we can install it and test it:

```
cd mylf
make setup      # make setup.py out of template
# make sure the $WSP is in $PYTHONPATH
python setup.py develop --install-dir=$WSP
lmi help
lmi help mylf
```

We should be able to see the usage string we've written.

Implementing `show` command Now let's implement the easiest command. Let's start with appending following snippet to *mylf/lmi/scripts/mylf/__init__.py*.

```
import os

from lmi.shell import LMIInstance, LMIInstanceName
from lmi.scripts.common import errors
from lmi.scripts.common import get_computer_system
from lmi.scripts.common import get_logger

LOG = get_logger(__name__)

def logical_file_type_name(file_identity):
    """
    Get a name of file type for supplied instance of ``CIM_LogicalFile``.
    """
    namemap = {
            'lmi_datafile'       : 'file',
            'lmi_unixdevicefile' : 'device',
            'lmi_unixdirectory'  : 'directory',
            'lmi_fifopipefile'   : 'fifo',
            'lmi_symboliclink'   : 'symlink',
            'lmi_unixsocket'     : 'socket'
    }
    try:
        return namemap[file_identity.classname.lower()]
    except KeyError:
        LOG().warn('Unhandled logical file class "%s".',
                file_identity.classname)
        return 'unknown'

def permission_string(file_identity):
    """
    Make an ls-like permission string for supplied instance of
    ``CIM_LogicalFile``.
    """
    return ''.join(l if getattr(file_identity, a) else '-'
              for l, a in zip('rwx', ('Readable', 'Writeable', 'Executable')))

def get_logical_file_instance(ns, file_ident, dereference=False):
    """
    Get an instance of ``CIM_LogicalFile`` corresponding to given file
    identity.

    :param file_ident: Either a file path or an instance of ``LMI_UnixFile``.
    :param boolean dereference: Whether to follow symbolic links
    """
```

```
    if isinstance(file_ident, basestring):
        uf = get_unix_file_instance(ns, file_ident, dereference)
    elif isinstance(file_ident, LMIInstanceName):
        uf = file_ident.to_instance()
    else:
        uf = file_ident
    return uf.first_associator(AssocClass='LMI_FileIdentity')

def get_unix_file_instance(ns, path, dereference=False):
    """
    :param boolean dereference: Whether to follow symbolic links
    :returns: Instance of ``LMI_UnixFile`` corresponding to given *path*.
    """
    cs = get_computer_system(ns)
    uf_name = ns.LMI_UnixFile.new_instance_name({
        'CSCreationClassName' : cs.classname,
        'CSName'              : cs.name,
        'LFName'              : path,
        'LFCreationClassName' : 'ignored',
        'FSCreationClassName' : 'ignored',
        'FSName'              : 'ignored',
    })
    try:
        uf = uf_name.to_instance()
        if dereference:
            lf = get_logical_file_instance(ns, uf, False)
            if logical_file_type_name(lf) == 'symlink':
                try:
                    target = lf.TargetFile
                    if not os.path.isabs(target):
                        target = os.path.abspath(
                                os.path.join(os.path.dirname(lf.Name), target))
                    # recursively try to dereference
                    uf = get_unix_file_instance(ns, target, dereference)
                except Exception as err:
                    LOG.warn('failed to get link target "%s": %s',
                            lf.TargetLink, err)
        return uf
    except:
        raise errors.LmiFailed('No such file or directory: "%s".' % path)
```

First two functions turn their argument to a human readable form. The other two are somewhat special. They actually interact with a broker. Each such function takes as a first argument a *namespace object*, *LMIShell's* abstraction, which acts as a liaison. All our communication is done through this object. We always name it `ns`. These are *getters* we will need in our *Show* command. Getters usually return one or several instances of `LMIInstanceName`.

Now let's place following into `mylf/lmi/scripts/mylf/cmd.py`.

```
from lmi.scripts import mylf
from lmi.scripts.common import command
from lmi.scripts.common import errors

class Show(command.LmiLister):
    COLUMNS = ('Attribute', 'Value')

    def transform_options(self, options):
        options['<path>'] = options.pop('<file>')

    def execute(self, ns, path, _dereference):
```

```
        uf = mylf.get_unix_file_instance(ns, path, _dereference)
        lf = mylf.get_logical_file_instance(ns, uf, _dereference)
        return [
            ('Path'        , lf.Name),
            ('Type'        , mylf.logical_file_type_name(lf)),
            ('User ID'     , uf.UserID),
            ('Group ID'    , uf.GroupID),
            ('Size'        , lf.FileSize),
            ('Permissions' , mylf.permission_string(lf))
        ]
```

And change `MyLF` command there like this:

```
MyLF = command.register_subcommands('MyLF', __doc__,
        { 'show' : Show })
```

All is set up. To try it out:

```
$ lmi -h $HOST mylf show /root
Attribute   Value
Path        /root
Type        directory
User ID     0
Group ID    0
Size        4096
Permissions r-x
```

Our `Show` command inherits from `LmiLister` which renderes a table. In order to do that it needs to know number of columns and their headings which specifies `COLUMNS` property.

Most of the work is done in its `execute()` method. All parameters following namespace object come from command line. First it collects the data, make them readable and then returns them as a list of rows.

Command line options need to be modified before passing them to object method. *Several rules* apply. We can see that `--dereference` option is turned to `_dereference` parameter name. Replacing leading dashes with single underscore is a default behaviour that you may customize.

Sometimes you may want to rename an option. This is a case of `<file>` argument that would be passed as a `file` which is python's *built-in*. Here comes `transform_options()` method into play. Any possible option manipulation is allowed here. It may be used also to convert values to your liking.

Implementing `list` Most of necessary functionality has been implemented in previous snippet for the `show` command. Following snippet is enough to generate all the files in directory. Put it again to `mylf/lmi/scripts/mylf/__init__.py`.

```
def make_directory_instance_name(ns, directory):
    """
    Retrieve object path of a directory.

    :type directory: string
    :param directory: Full path to the directory.
    :rtype: :py:class:`lmi.shell.LMIInstanceName.LMIInstanceName`
    """
    if directory != '/':
        directory = directory.rstrip('/')
    cs = get_computer_system(ns)
    return ns.LMI_UnixDirectory.new_instance_name(
            { 'CSCreationClassName' : cs.classname
            , 'CSName'              : cs.name
```

```
            , 'CreationClassName'   : 'LMI_UnixDirectory'
            , 'FSCreationClassName' : 'LMI_LocalFileSystem'
            , 'FSName'              : ''
            , 'Name'                : directory})

def get_directory_instance(ns, directory):
    """
    Retrieve instance of 'LMI_UnixDirectory'.

    :type directory: string of :py:class:'lmi.shell.LMIInstanceName.LMIInstanceName'
    :param directory: Full path to the directory or its instance name.
    :rtype: :py:class:'lmi.shell.LMIInstance.LMIInstance'
    """
    if isinstance(directory, basestring):
        directory = make_directory_instance_name(ns, directory)
    if isinstance(directory, LMIInstanceName):
        directory = directory.to_instance()
    return directory

def list_directory(ns, directory, file_type='any'):
    """
    Yields instances of ''CIM_LogicalFile'' representing direct children of the
    given directory.

    :param directory: Either a file path or an instance of
        ''LMI_UnixDirectory''.
    :param file_type: Filter of files made by checking their type. One of: ::

        {'any', 'file', 'device', 'directory', 'fifo', 'symlink', 'socket'}
    """
    def _generate_children():
        for child in get_directory_instance(ns, directory).associators(
                AssocClass='LMI_DirectoryContainsFile',
                Role='GroupComponent',
                ResultRole='PartComponent'):
            if (   file_type and file_type != 'any'
               and logical_file_type_name(child) != file_type):
                continue
            yield child
    return sorted(_generate_children(), key=lambda i: i.Name)
```

Note the `associators()` call on `LMI_UnixDirectory` instance. It enumerates all `CIM_LogicalFile` instances that are referenced by `LMI_DirectoryContainsFile` associations. These represent a relation of parent directory and its direct children. Parent directory is referenced with `GroupComponent` role while the children with `PartComponent`. It's advisable to always provide as much information to calls like:

- `associators()`
- `associator_names()`
- `references()`
- `reference_names()`

as possible. Without the `AssocClass` parameter given, broker would try to enumerate all instrumented association classes possible, resulting in very poor performance. Both `Role` and `ResultRole` parameters need to be given here, otherwise a parent directory of the one being enumerated would also appear in output.

Following subclass of `LmiInstanceLister` needs to be added to `mylf/lmi/scripts/mylf/cmd.py` and added to `MyLF` subcommands dictionary (omitted for now).

```
class List(command.LmiInstanceLister):
    CALLABLE = mylf.list_directory
    PROPERTIES = (
            'Name',
            ('Type', mylf.logical_file_type_name),
            ('Permissions', mylf.permission_string),
            ('Size', 'FileSize'))

    def verify_options(self, options):
        if options['--type'] is not None \
               and not options['--type'].lower() in {
                   'any', 'file', 'directory', 'symlink', 'dev', 'socket',
                   'fifo'}):
            raise errors.LmiInvalidOptions(
                    'Unsupported type: %s' % options['--type'])

    def transform_options(self, options):
        file_type = options.pop('--type')
        if file_type is None:
            file_type = 'any'
        options['file-type'] = file_type
```

Instead of defining our own `execute()` method, we just associate `list_directory()` function with `List` command using `CALLABLE` property. Thanks to the ability to transform option names in any way, we are not limited to the use of arguments as listed in usage string. Apart from renaming options, we also check the value of `--type` option. Overriding `verify_options()` to check for validity of options is the more preferred approach compared to delayed checking in associated function.

Implementing `create` and `delete` Let's again start with content of `mylf/lmi/scripts/mylf/__init__.py` module.

```
def create_directory(ns, directory):
    """
    Create a directory.

    :type directory: string
    :param directory: Full path to the directory.
    """
    ns.LMI_UnixDirectory.create_instance(
            make_directory_instance_name(ns, directory).path.keybindings)

def delete_directory(ns, directory):
    """
    Delete an empty directory.

    :param directory: Either a file path or an instance of
        ``LMI_UnixDirectory``.
    """
    get_directory_instance(ns, directory).delete()
```

`create_instance()` call of any `LMIClass` creates a new instance, in this case we create an instance of `LMI_UnixDirectory`. If it exists already, an exception will be raised. On the other hand, `delete_directory()` operates on an `LMIInstance` which must exists. If directory does not exist or it's not empty, an exception will be raised.

Now let's move on to `mylf/lmi/scripts/mylf/cmd.py`:

```
class Create(command.LmiCheckResult):
    EXPECT = None
    CALLABLE = mylf.create_directory

class Delete(command.LmiCheckResult):
    EXPECT = None
    CALLABLE = mylf.delete_directory
```

`LmiCheckResult` is a special command that prints no useful information. It allows us to check, whether the associated function returns expected result and prints an error if not. Here we expect `None`. Associated functions in this case throw an exception upon any error which have the same effect.

Test it

```
lmi -h $HOST mylf create /root/some_directory
# try it for the second time (it will fail)
lmi -h $HOST mylf create /root/some_directory
# now let's delete it
lmi -h $HOST mylf delete /root/some_directory
# try it for the second time (it will fail)
lmi -h $HOST mylf delete /root/some_directory
```

Summary Now that the script is ready and tested, we may commit it, push it, do a *pull request* and host it on PyPI:

```
python setup.py register
python setup.py sdist upload
```

Source code of this example is available as a `tarball`.

Basics

This provides a general overview on what script is, how is it written and is interfaced with.

Prerequisities Reader should be familiar with a CIM (Common Information Model). He should have a general idea about, what OpenLMI is and what it does. He should get familiar with LMIShell, which is a python binary shipped with *OpenLMI client components*.

Also user should be familiar with standard *nix command line utilities [1].

Introduction By a *script* in this document we mean:

- Python library utilizing LMIShell for instrumenting CIM providers through a CIMOM broker. It resides in `lmi.scripts.<script_name>` package. Where `<script_name>` usually corresponds to some *LMI* profile name.
- Command wrappers for this library as a set of classes inheriting from `LmiBaseCommand`. These may create a tree-like hierarchy of commands. They are the entry points of *LMI metacommand* to the wrapped functionality of library.

Command wrappers are part of the library usually grouped in a single module named after the `lmi` command or `cmd`:

```
lmi.scripts.<script_name>.cmd
```

[1] Described by a POSIX.

Writing a library Library shall consist of a set of functions taking a *namespace* or connection object as a first argument. There are no special requirements on how to divide these functions into submodules. Use common sense. Smaller scripts can have all functionality in `lmi/scripts/<script_name>/__init__.py` module. With wrappers usually contained in `lmi/scripts/<script_name>/cmd.py`.

Library should be written with an ease of use in mind. Functions should represent possible use cases of what can be done with particular providers instead of wrapping 1-to-1 a CIM class's methods in python functions.

Any function that shall be called by a command wrapper and communicates with a CIMOM must accept a namespace object named as `ns`. It's an instance of `LMINamespace` providing quick access to represented CIM namespace [2] and its classes. It's also possible to specify that function shall be passed a raw `LMIConnection` object. For details see *Function invocation*.

Service example Suppose we have a service profile in need of ython interface. Real provider implementation can be found at `src/service` directory in upstream git [3]. For more information please refer to service description.

As you may see, this implements single CIM class `LMI_Service` with a few useful methods such as:

- `StartService()`
- `StopService()`

We'd like to provide a way how to list system services, get a details for one of them and allow to start, stop and restart them.

Simplified [4] version of some of these functions may look like this:

```
def list_services(ns, kind='enabled'):
    for service in sorted(ns.LMI_Service.instances(),
                key=lambda i: i.Name):
        if kind == 'disabled' and service.EnabledDefault != \
                ns.LMI_Service.EnabledDefaultValues.Disabled:
            continue
        if kind == 'enabled' and service.EnabledDefault != \
                ns.LMI_Service.EnabledDefaultValues.Enabled:
            # list only enabled
            continue
        yield service
```

It yields instances of `LMI_Service` cim class. We prefer to use `yield` instead of `return` when enumerating instances because of memory usage reduction. For example when the user limits the number of instances listed. With `yield` the number of iterations will be reduced automatically.

```
from lmi.shell import LMIInstanceName
from lmi.scripts.common import get_logger
from lmi.scripts.common.errors import LmiFailed

LOG = get_logger(__name__)

def start_service(ns, service):
    if isinstance(service, basestring):
        # let's accept service as a string
        inst = ns.LMI_Service.first_instance(key="Name", value=service)
        name = service
    else:   # or as LMIInstance or LMIInstanceName
        inst = service
```

[2] Default namespace is `"root/cimv2"`.
[3] view: https://fedorahosted.org/openlmi/browser/openlmi-providers git: `ssh://git.fedorahosted.org/git/openlmi-providers.git/`
[4] Simplified here means that there are no documentation strings and no type checking.

```
        name = inst.path['Name']
    if inst is None:
        raise LmiFailed('No such service "%s".' % name)
    if isinstance(inst, LMIInstanceName):
        # we need LMIInstance
        inst = inst.to_instance()
    res = inst.StartService()
    if res == 0:
        LOG().debug('Started service "%s" on hostname "%s".',
                    name, ns.hostname)
    return res
```

In similar fashion, `stop_service`, `restart_service` and others could be written.

`ns` argument typically represents `root/cimv2` namespace which is the main implementation namespace for `OpenLMI` providers. One could also make these functions act upon a connection object like this:

```
def get_instance(c, service):
    inst = c.root.cimv2.LMI_Service.first_instance(
                key="Name", value=service)
    if inst is None:
        raise LmiFailed('No such service "%s".' % service)
    return inst
```

User can then easily access any other namespace he may need. Command classes need to be informed about an object type the wrapped function expects (see *Function invocation*).

The `LOG` variable provides access to the logger of this module. Messages logged in this way end up in a log file [5] and console. Implicitly only warnings and higher priority messages are logged into a console. This can be changed with metacommand's parameteres.

If operation fails due to some unexpected error, please raise `LmiFailed` exception with human readable description.

See also:

Exceptions for conventions on using exceptions.

Upstream git for more *real world* examples.

Command wrappers overview They are a set of command classes wrapping up library's functionality. They are structured in a tree-like hierarchy where the root [6] command appears in a help message of *LMI metacommand*. All commands are subclasses of `LmiBaseCommand`.

Behaviour of commands is controlled by class properties such as these:

```
class Show(command.LmiShowInstance):
    CALLABLE = 'lmi.scripts.service:get_instance'
    PROPERTIES = (
            'Name',
            'Caption',
            ('Enabled', lambda i: i.EnabledDefault == 2),
            ('Active', 'Started'),
            'Status')
```

Example above contains definition of `Show` command wrapper for instances of `LMI_Service`. Its associated function is `get_instance()` located in `lmi.scripts.service` module [7]. Properties used will be described in detail *later*. Let's just say, that `PROPERTIES` specify a way how the instance is rendered.

[5] If logging to a file is enabled in configuration.
[6] Also called a top-level command.
[7] Precisely in an `__init__.py` module of this package.

Top-level commands Are entry points of a script library. They are direct subcommands of `lmi`. For example:

```
$ lmi help
$ lmi service list
$ lmi sw show openlmi-providers
```

`help`, `service` and `sw` are top-level commands. One script (such as `service` above) can provide one or more of them. They need to be listed in a `setup.py` script in `entry_points` argument of `setup()` function. More details will be noted later in Setup script.

They contain usage string which is a documentation and prescription of command-line arguments in one string. This string is printed when user requests command's help:

```
$ lmi help service
```

Usage string looks like this:

```
"""
System service management.

Usage:
    %(cmd)s list [--all | --disabled]
    %(cmd)s start <service>

Options:
    --all       List all services available.
    --disabled  List only disabled services.
"""
```

Format of this string is very important. It's parsed by a docopt command line parser which uses it for parsing command-line arguments. Please refer to its documentation for details.

Note: There is one deviation to *common* usage string. It's the use of `%(cmd)s` formatting mark. It is replaced with full command's name. Full name means that all subcommands and binary name prefixing current command on command line are part of it. So for example full name of command **list** in a following string passed to command line:

```
lmi sw list pkgs
```

is `lmi sw list`.

If parsing **sw** usage, it is just `lmi sw`.

The formatting mark is mandatory.

Options and arguments given on command-line are *pre-processed* before they are passed to *end-point command*. You should get familier with it before writing your own usage strings.

End-point commands Are associated with one or more function of script library. They handle the following:

1. call docopt parser on command line arguments
2. make some name pre-processing on them (see *Options pre-processing*)
3. verify them (see *End-point commands*)
4. transform them (see *End-point commands*)
5. pass them to associated function
6. collect results

7. render them and print them

Developper of command wrappers needs to be familiar with each step. We will describe them later in details.

There are following end-point commands available for subclassing:

- `LmiCheckResult` (see *LmiCheckResult*)
- `LmiLister` (see *LmiLister*)
- `LmiInstanceLister` (see *LmiInstanceLister*)
- `LmiShowInstance` (see *LmiShowInstance*)

They differ in how they render the result obtained from associated function.

These are documented in depth in *End-point commands*.

Command multiplexers Provide a way how to group multiple commands under one. Suppose you want to list packages, repositories and files. All of these use cases need different arguments, and render different information thus they should be represented by independent end-point commands. What binds them together is the user's intent to *list* something. He may wish to do other operation like *show*, *add*, *remove* etc. with the same subject. Having all combination of these intents and subjects would generate a lot of commands under the top-level one. Let's instead group them under particular *intent* like this:

- `sw list packages`
- `sw list repositories`
- `sw list files`
- `sw show package`

To reflect it in our commands hierarchy, we need to use `LmiCommandMultiplexer` command.

```
class Lister(command.LmiCommandMultiplexer):
    """ List information about packages, repositories or files. """
    COMMANDS = {
            'packages'     : PkgLister,
            'repositories' : RepoLister,
            'files'        : FileLister
    }
```

Where `COMMANDS` property maps command classes to their names. Each command multiplexer consumes one command argument from command line, denoting its direct subcommand and passes the rest of options to it. In this way we can create arbitrarily tall command trees.

Top-level command is nothing else than a subclass of `LmiCommandMultiplexer`.

Specifying profile and class requirements Most commands require some provider installed on managed machine to work properly. Each such provider should be represented by an instance of `CIM_RegisteredProfile` on remote broker. This instance looks like this (in MOF syntax):

```
instance of CIM_RegisteredProfile {
    InstanceID = "OpenLMI+OpenLMI-Software+0.4.2";
    RegisteredOrganization = 1;
    OtherRegisteredOrganization = "OpenLMI";
    RegisteredVersion = "0.4.2";
    AdvertiseTypes = [2];
    RegisteredName = "OpenLMI-Software";
};
```

We are interested just in `RegisteredName` and `RegisteredVersion` properties that we'll use for requirement specification.

Requirement is written in *LMIReSpL* language. For its formal definition refer to documentation of `parser`. Since the language is quite simple, few examples should suffice:

`'OpenLMI-Software < 0.4.2'` Requires OpenLMI Software provider to be installed in version lower than `0.4.2`.

`'OpenLMI-Hardware == 0.4.2 & Openlmi-Software >= 0.4.2'` Requires both hardware and software providers to be installed in particular version. Short-circuit evaluation is utilized here. It means that in this example OpenLMI Software won't be queried unless OpenLMI Hardware is installed and having desired version.

`'profile "OpenLMI-Logical File" > 0.4.2'` If you have spaces in the name of profile, surround it in double quotes. `profile` keyword is optional. It could be also present in previous examples.

Version requirements are not limited to profiles only. CIM classes may be specified as well:

`'class LMI_SoftwareIdentity >= 0.3.0 & OpenLMI-LogicalFile'` In case of class requirements the `class` keyword is mandatory. As you can see, version requirement is optional.

`'! (class LMI_SoftwareIdentity | class LMI_UnixFile)'` Complex expressions can be created with the use of brackets and other operators.

One requirement is evaluated in these steps:

Profile requirement

1. Query `CIM_RegisteredProfile` for instances with `RegisteredName` matching given name. If found, go to 2. Otherwise query `CIM_RegisteredSubProfile` [8] for instances with `RegisteredName` matching given name. If not found return `False`.
2. Select the (sub)profile with highest version and go to 3.
3. If the requirement has version specification then compare it to the value of `RegisteredVersion` using given operator. If the relation does not apply, return `False`.
4. Return `True`.

Class requirement

1. Get specified class. If not found, return `False`.
2. If the requirement has version specification then compare it to the value of `Version` [9] qualifier of obtained class using given operator. And if the relation does not apply, return `False`.
3. Return `True`.

Now let's take a look, where these requirements can be specified. There is a special select command used to specify which command to load for particular version on remote broker. It can be written like this:

```
from lmi.scripts.common.command import LmiSelectCommand

class SoftwareCMD(LmiSelectCommand):

    SELECT = [
          ( 'OpenLMI-Software >= 0.4.2 & OpenLMI-LogicalFile'
          , 'lmi.scripts.software.current.SwLFCmd')
```

[8] This is a subclass of `CIM_RegisteredProfile` thus it has the same properties.
[9] If the Version qualifier is missing, -1 will be used for comparison instead of empty string.

```
        , ( 'OpenLMI-Software >= 0.4.2'
          , 'lmi.scripts.software.current.SwCmd')
        , ('OpenLMI-Software', 'lmi.scripts.software.pre042.SwCmd')
    ]
```

It says to load `SwLFCmd` command in case both OpenLMI Software and OpenLMI LogicalFile providers are installed. If not, load the `SwCMD` from `current` module for OpenLMI Software with recent version and fallback to `SwCmd` for anything else. If the OpenLMI Software provider is not available at all, no command will be loaded and exception will be raised.

Previous command could be used as an entry point in your `setup.py` script (see the *Entry points*). There is also a utility that makes it look better:

```
from lmi.scripts.common.command import select_command

SoftwareCMD = select_command('SoftwareCMD',
      ( 'OpenLMI-Software >= 0.4.2 & OpenLMI-LogicalFile'
      , 'lmi.scripts.software.current.SwLFCmd'),
      ( 'OpenLMI-Software >= 0.4.2', 'lmi.scripts.software.current.SwCmd'),
      ('OpenLMI-Software', 'lmi.scripts.software.pre042.SwCmd')
)
```

See also:

Documentation of `LmiSelectCommand` and `select_command`.

And also notes on related *LmiSelectCommand properties*.

Command wrappers module Usually consists of:

1. license header
2. usage dostring - parseable by docopt
3. end-point command wrappers
4. single top-level command

The top-level command is usally defined like this:

```
Service = command.register_subcommands(
        'Service', __doc__,
        { 'list'    : Lister
        , 'show'    : Show
        , 'start'   : Start
        , 'stop'    : Stop
        , 'restart' : Restart
        },
    )
```

Where the `__doc__` is a usage string and module's doc string at the same time. It's mentioned in point 2. `Service` is a name, which will be listed in `entry_points` dictionary described *below*. The global variable's name we assign to should be the same as the value of the first argument to `register_subcommands()`. The last argument here is the dictionary mapping all subcommands of **service** to their names [10].

Egg structure Script library is distributed as a python egg, making it easy to distribute and install either to system or user directory.

[10] Taken from older version of storage script.

Following tree shows directory structure of *service* egg residing in upstream git:

[dirtree] commands/service child node lmi child node __init__.py child node scripts child node __init__.py child node service child node cmd.py child node __init__.py child node Makefile child node README.md child node setup.cfg child node setup.py.skel ;

This library then can be imported with:

```
from lmi.scripts import service
```

`commands/service/lmi/scripts/service` must be a package (directory with `__init__.py`) because `lmi.scripts` is a namespace package. It can have arbitrary number of modules and subpackages. The care should be taken to make the API easy to use and learn though.

Use provided `commands/make_new.py` script to *generated it*.

Setup script Follows a minimal example of `setup.py.skel` script for service library.

```
from setuptools import setup, find_packages
setup(
    name="openlmi-scripts-service",
    version="@@VERSION@@",
    description='LMI command for system service administration.',
    url='https://github.com/openlmi/openlmi-scripts',
    platforms=['Any'],
    license="BSD",
    install_requires=['openlmi-scripts'],
    namespace_packages=['lmi', 'lmi.scripts'],
    packages=['lmi', 'lmi.scripts', 'lmi.scripts.service'],

    entry_points={
        'lmi.scripts.cmd': [
            'service = lmi.scripts.service.cmd:Service',
            ],
        },
    )
```

It's a template with just one variable `@@VERSION@@` being replaced with recent scripts version by running `make setup` command.

Entry points The most notable argument here is `entry_points` which is a dictionary containing python namespaces where plugins are registered. In this case, we register single *top-level command* called `service` in `lmi.scripts.cmd` namespace. This particular namespace is used by *LMI metacommand* when searching for registered user commands. `Service` is a command multiplexer, created with a call to `register_subcommands()` grouping end-point commands together.

Next example shows set up with more top-level commands [11]:

```
entry_points={
    'lmi.scripts.cmd': [
        'fs = lmi.scripts.storage.fs_cmd:Fs',
        'partition = lmi.scripts.storage.partition_cmd:Partition',
        'raid = lmi.scripts.storage.raid_cmd:Raid',
        'lv = lmi.scripts.storage.lv_cmd:Lv',
        'vg = lmi.scripts.storage.vg_cmd:Vg',
        'storage = lmi.scripts.storage.storage_cmd:Storage',
        'mount = lmi.scripts.storage.mount_cmd:Mount',
```

[11] These names must exactly match the names in usage strings.

```
    ],
},
```

Conventions There are several conventions you should try to follow in your shiny scripts.

Logging messages In each module where logging facilities are going to be used, define global varibale `LOG` like this:

```
from lmi.scripts.common import get_logger

LOG = get_logger(__name__)
```

It's a callable used throughout particular module in this way:

```
LOG().warn('All the data of "%s" will be lost!', partition)
```

Each message should be a whole sentence. It shall begin with an upper case letter and end with a dot or other sentence terminator.

Bad example:

```
LOG().info('processing %s', card)
```

Exceptions Again all the exceptions should be initialized with messages forming a whole sentence.

They will be catched and printed on `stderr` by *LMI metacommand*. If the *Trace* option in *Section [Main]* is on, traceback will be printed. There is just one exception. If the exception inherits from `LmiError`, traceback won't be printed unless verbosity level is the highest one as well:

```
# self refers to some command
self.app.config.verbosity == self.app.config.OUTPUT_DEBUG
```

This is a feature allowing for common error use-cases to be gracefully handled. In your scripts you should stick to using `LmiFailed` for such exceptions.

Following is an example of such a common error-case, where printing traceback does not add any interesting information:

```
iname = ns.LMI_Service.new_instance_name({
    "Name": service,
    "CreationClassName" : "LMI_Service",
    "SystemName" : cs.Name,
    "SystemCreationClassName" : cs.CreationClassName
})
inst = iname.to_instance()
if inst is None:
    raise errors.LmiFailed('No such service "%s".' % service)
# process the service instance
```

`service` is a name provided by user. If such a service is not found, `inst` will be assigned `None`. In this case we don't want to continue in script's execution thus we raise an exception. We provide very clear message that needs no other comment. We don't want any traceback to be printed, thus the use of `LmiFailed`.

Debugging To hunt down problems of your script during its development, metacommand comes with few options to assist you:

--trace This option turns on logging of tracebacks. Any exception but `LmiError` will be logged with traceback to `stderr` unless `--quite` option is on. `LmiError` will be logged with traceback if the verbosity (`-v`) is highest as well.

-v Raise a verbosity. Pass it twice to make the verbosity highest. That will cause a lot of messages being produced to `stderr`. It also turns on logging of tracebacks for `LmiError` if `--trace` option is on as well.

--log-file Allows to specify output file, where logging takes place. Logging level is not affected by `-v` option. It can be specified in configuration file.

While you debug it's convenient to put above in your configuration file `~/.lmirc`:

```
[Main]
# Print tracebacks.
Trace = True

[Log]
OutputFile = /tmp/lmi.log
# Logging level for OutputFile.
Level = DEBUG
```

See also:

Configuration

See also:

Docopt documentation, *Command classes* and *Command properties*.

Command classes

Before reading this, please make sure you're familiar with *Command wrappers overview*.

End-point commands Were already introduced before (see *End-point commands*). We'll dive into details here.

Every end-point command allows to verify and transform options parsed by docopt before they are passed to associated function. This can happen in methods:

verify_options(self, options) Taking pre-processed options dictionary as a first argument. Properties affecting this pre-processing can be found in *Options pre-processing*. This method shall check option values or their combinations and raise `LmiInvalidOptions` if any inconsistency is discovered.

Example usage:

```
class FileLister(command.LmiInstanceLister):
    DYNAMIC_PROPERTIES = True

    def verify_options(self, options):
        file_types = { 'all', 'file', 'directory', 'symlink'
                     , 'fifo', 'device'}
        if (   options['--type'] is not None
           and options['--type'] not in file_types):
            raise errors.LmiInvalidOptions(
                    'Invalid file type given, must be one of %s' %
                         file_types)
```

See also:

API documentation on `verify_options()`

`transform_options(self, options)` Takes verified options dictionary which it modifies in place.

Example usage:

```
class Lister(command.LmiLister):
    COLUMNS = ('Device', 'Name', "ElementName", "Type")

    def transform_options(self, options):
        """
        Rename 'device' option to 'devices' parameter name for better
        readability.
        """
        options['<devices>'] = options.pop('<device>')
```

See also:

API documentation on `transform_options()`

Above methods can be used to process options in a way that any script library function can be called. In case we need more control over what is called or when we want to decide at runtime which function shall be called, we may override `execute()` method instead. Example of this may be found at *Associating a function*.

`LmiCheckResult` This command invokes associated function on hosts in session, collects results from them and compares them to an expected value. It does not produce any output, when all returned values are expected.

This command class is very useful when wrapping up some CIM class's method such as `LMI_Service::StartService()`. Example can be seen in *Property descriptions*.

Its specific properties are listed in *LmiCheckResult properties*.

See also:

API documentation on `LmiCheckResult`

`LmiLister` Prints tablelike data. It expects associated function to return its result in form:

```
[row1, row2, ...]
```

Where `rowX` is a tuple containing row values. Each such row is `list` or `tuple` of the same length. There is a property `COLUMNS` defining column names [12] (see *LmiLister properties*). Generator is preferred over a `list` of rows.

```
class RaidList(command.LmiLister):
    COLUMNS = ('Name', "Level", "Nr. of members")

    def execute(self, ns):
        """
        Implementation of 'raid list' command.
        """
        for r in raid.get_raids(ns):
            members = raid.get_raid_members(ns, r)
            yield (r.ElementName, r.Level, len(members))

        # Could also be written as:
        #return [  (r.ElementName, r.Level, len(raid.get_raid_members(ns, r)))
        #       for r in raid.get_raids(ns)]
```

[12] Having the same length as each row in returned data.

produces:

```
$ lmi -h $HOST storage raid list
Name  Level Nr. of members
raid5 5     3
```

If `COLUMNS` property is omitted, returned value shall take the following form instead:

```
(columns, data)
```

Where `columns` has the same meaning as `COLUMNS` as a class property and `data` is the result of previous case [13].

```
def get_thin_pools(ns, verbose):
    for vg in lvm.get_tps(ns):
        extent_size = size2str(vg.ExtentSize, self.app.config.human_friendly)
        if verbose:
            total_space = size2str(vg.TotalManagedSpace,
                    self.app.config.human_friendly)
            yield (vg.ElementName, extent_size, total_space)
        else:
            yield (vg.ElementName, extent_size)

class ThinPoolList(command.LmiLister):

    def execute(self, ns):
        """
        Implementation of 'thinpool list' command.
        """
        columns = ['ElementName', "ExtentSize"]
        if self.app.config.verbose:
            columns.extend(["Total space"])
        return (columns, get_thin_pools(ns, self.app.config.verbose))
```

Produces:

```
$ lmi -H -h $HOST storage thinpool list
ElementName ExtentSize
tp1         4M
$ # The same with increased verbosity
$ lmi -v -H -h $HOST storage thinpool list
ElementName ExtentSize Total space
tp1         4M         1024M
```

See also:

API documentation on `LmiLister`

`LmiInstanceLister` Is a variant of `LmiLister`. Instead of rows being tuples, here they are instances of some CIM class. Instead of using `COLUMNS` property for specifying columns labels, `PROPERTIES` is used for the same purpose here. Its primary use is in specifying which properties of instances shall be rendered in which column. This is described in detail in *LmiShowInstance and LmiInstanceLister properties*.

The expected output of associated function is therefore:

```
[instance1, instance2, ...]
```

Again, usage of generators is preferred.

See also:

[13] Generator or a `list` of rows.

API documentation on `LmiInstanceLister`

`LmiShowInstance` Renders a single instance of some CIM class. It's rendered in a form of two-column table, where the first column contains property names and the second their corresponding values. Rendering is controlled in the same way as for `LmiInstanceLister` (see *LmiShowInstance and LmiInstanceLister properties*).

See also:

API documentation on `LmiShowInstance`

Command multiplexers Group a list of commands under one. They were introduced *earlier*. Their children can be end-point commands as well as multiplexers. Thus arbitrary tall command trees can be constructed - though not being very practical.

Multiplexer works like this

1. it consumes one argument from command line
2. selects one of its subcommands based on consumed argument
3. passes the rest of arguments to selected subcommand and executes it
4. returns the result to a caller

For example consider following list of arguments:

```
storage raid create --name raid5 5 /dev/vdb /dev/vdc /dev/vdd
```

LMI metacommand consumes `storage` command multiplexer and passes the rest to it:

```
Storage().run(["raid", "create", "--name", "raid5",  "5", "/dev/vdb",
        "/dev/vdc", "/dev/vdd"])
```

`Storage`, which can be defined like this:

```
Storage = command.register_subcommands(
        'storage', __doc__,
        { 'tree'      : Tree,
          'partition': lmi.scripts.storage.cmd.partition.Partition,
          'fs'        : lmi.scripts.storage.cmd.fs.FS,
          'raid'      : lmi.scripts.storage.cmd.raid.Raid,
        },
    )
```

, consumes the first argument and passes the rest to the `raid` command which is again a multiplexer defined like this:

```
class Raid(command.LmiCommandMultiplexer):
    OWN_USAGE = __doc__
    COMMANDS = {
            'list'    : RaidList,
            'create'  : RaidCreate,
            'delete'  : RaidDelete,
            'show'    : RaidShow,
    }
```

`create` end-point command will then be invoked with:

```
["--name", "raid5",  "5", "/dev/vdb", "/dev/vdc", "/dev/vdd"]
```

Note: Each above multiplexer is defined in its own module with usage string at its top. It is far more legible than having couple of multiplexers sharing single module.

Splitting usage string Multiplexers delegating work to children multiplexers, like in the example above, need to be given a special usage string.

Every multiplexer subcommand in the usage string must be followed with:

```
<cmd> [<args> ...]
```

Like in the usage of `Storage` above:

```
"""
Basic storage device information.

Usage:
    %(cmd)s tree [ <device> ]
    %(cmd)s partition <cmd> [<args> ...]
    %(cmd)s fs <cmd> [<args> ...]
    %(cmd)s raid <cmd> [<args> ...]
"""
```

`cmd` and `args` may be renamed to your liking. Only the form matters. It ensures that anything after the `cmd` won't be inspected by this multiplexer – the work is delegated to the children.

As you can see, end-point and multiplexer commands may be freely mixed. The `tree` end-point command does not have its own usage string because all its arguments are parsed by `Storage`.

See also:

General and class specific properties in *Command properties*.

Command properties

As noted before in *End-point commands*, command at first tries to process input arguments, calls an associated function and then renders its result. We'll now introduce properties affecting this process.

Command class properties are written in their bodies and handled by their metaclasses. After being processed, they are removed from class. So they are not accessible as class attributes or from their instances.

Options pre-processing Influencing properties:

- `OPT_NO_UNDERSCORES` (opt_no_underscores)
- `ARG_ARRAY_SUFFIX` (arg_array_suffix)
- `OWN_USAGE` (own_usage)

docopt will make a dictionary of options based on usage string such as the one above (*Usage string*). Options dictionary matching this example looks like this:

```
{ 'list'       : bool    #  Usage:
, '--all'      : bool    #      %(cmd)s list [--all | --disabled]
, '--disabled' : bool    #      %(cmd)s start <service>
, 'start'      : bool    #
, '<service>'  : str     #  Options:
}                        #      --all       List all services available.
                         #      --disabled  List only disabled services.
```

Values of this dictionary are passed to an associated function as arguments with names created out of matching keys. Since argument names can not contain characters such as `<`, `>`, `-`, etc., these need to be replaced. Process of renaming of these options can be described by the following pseudo algorithm:

1. arguments enclosed in brackets are un-surrounded – brackets get removed

   ```
   "<service>" -> "service"
   ```

2. arguments written in upper case are made lower cased

   ```
   "FILE" -> "file"
   ```

3. prefix of short and long options made of dashes shall be replaced with single underscore

   ```
   "-a"    -> "_a"
   "--all" -> "_all"
   ```

4. any non-empty sequence of characters not allowed in python's identitier shall be replaced with a single underscore

   ```
   "_long-option"     -> "_long_option"
   "special--cmd-#2"  -> "special_cmd_2"
   ```

Points 3 and 4 could be merged into one. But we separate them due to effects of `OPT_NO_UNDERSCORES` property described below.

See also:

Notes in *End-point commands* for method :py:meth‘lmi.scripts.common.command.endpoint.LmiEndPointCommand.transform_options‘ which is issued before the above algorithm is run.

Treating dashes Single dash and double dash are special cases of commands.

Double dash in usage string allows to pass option-like argument to a script e.g.:

```
lmi file show -- --file-prefix-with-double-dash
```

Without the `'--'` argument prefixing the file, docopt would throw an error because of `--file-prefix-with-double-dash` being treated as an unknown option. This way it's correctly treated as an argument `<file>` given the usage string:

```
Usage: %(cmd)s file show [--] <file>
```

Double dash isn't passed to an associated function.

Single dash on a command line is commonly used to specify `stdout` or `stdint`. For example in the following snippet:

```
Usage: %(cmd)s file copy (- | <file>) <dest>
```

`'-'` stands for standard input which will be read instead of a file if the user wishes to.

Property descriptions

`OPT_NO_UNDERSCORES` [`bool` (defaults to `False`)] Modifies point 3 of options pre-processing. It causes the prefix of dashes to be completely removed with no replacement:

```
"--long-options" -> "long-option"
```

This may not be save if there is a command with the same name as the option being removed. Setting this property to `True` will cause overwriting the command with a value of option. A warning shall be echoed if such a case occurs.

ARG_ARRAY_SUFFIX [`str` (defaults to `""`)] Adds additional point (5) to options_transform_algorithm. All repeatable arguments, resulting in a `list` of items, are renamed to `<original_name><suffix>` [14]. Repeatable argument in usage string looks like this:

```
"""
Usage: %(cmd)s start <service> ...
"""
```

Causing all of the `<service>` arguments being loaded into a `list` object.

OWN_USAGE [`bool` (defaults to `False`)] Says whether the documentation string of this class is a usage string. Each command in hierarchy can have its own usage string.

This can also be assigned a usage string directly:

```
class MySubcommand(LmiCheckResult):
    """
    Class doc string.
    """
    OWN_USAGE = "Usage: %(cmd)s --opt1 --opt1 <file> <args> ..."
    EXPECT = 0
```

But using a boolean value is more readable:

```
class MySubcommand(LmiCheckResult):
    """
    Usage: %(cmd)s --opt1 --opt1 <file> <args> ...
    """
    OWN_USAGE = True
    EXPECT = 0
```

Note: Using own usage strings in end-point commands is not recommended. It brings a lot of redundancy and may prove problematic to modify while keeping consistency among hierarchically nested usages.

It's more readable to put your usage strings in your command multiplexers and put each of them in its own module.

See also:

Command multiplexers

Associating a function Influencing properties:

- `CALLABLE` (callable)

When command is invoked, its method `execute()` will get verified and transformed options as positional and keyword arguments. This method shall pass them to an associated function residing in script library and return its result on completion.

One way to associate a function is to use `CALLABLE` property. The other is to define very own `execute()` method like this:

```
class Lister(command.LmiInstanceLister):
    PROPERTIES = ('Name', "Started", 'Status')
```

[14] Angle brackets here just mark the boundaries of name components. They have nothing to do with arguments.

```
def execute(self, ns, _all, _disabled, _oneshot):
    kind = 'enabled'
    if _all:
        kind = 'all'
    elif _disabled:
        kind = 'disabled'
    elif _oneshot:
        kind = 'oneshot'
    for service_inst in service.list_services(ns, kind):
        yield service_inst
```

This may come handy if the application object [15] needs to be accessed or if we need to decide which function to call based on command line options.

Property descriptions

CALLABLE [str (defaults to None)] This is a mandatory option if execute() method is not overriden. It may be a string composed of a full path of module and its callable delimited with ':':

```
CALLABLE = 'lmi.scripts.service:start'
```

Causes function start() of 'lmi.scripts.service' module to be associated with command.

Callable may also be assigned directly like this:

```
from lmi.scripts import service
class Start(command.LmiCheckResult):
    CALLABLE = service.start
    EXPECT = 0
```

The first variant (by assigning string) comes handy if the particular module of associated function is not yet imported. Thus delaying the import until the point of function's invocation - if the execution comes to this point at all. In short it speeds up execution of *LMI metacommand* by reducing number of module imports that are not needed.

Function invocation Influencing properties:

- NAMESPACE (namespace)

Property descriptions

NAMESPACE [str (defaults to None)] This property affects the first argument passed to an associated function. Various values have different impact:

Value	Value of first argument.	Its type
None	Same impact as value "root/cimv2"	lmi.shell.LMINamespace.LMINamespace
False	Raw connection object	lmi.shell.LMIConnection.LMIConnection
any path	Namespace object with given path	lmi.shell.LMINamespace.LMINamespace

This usually won't need any modification. Sometimes perhaps associated function might want to access more than one namespace, in that case an instance of lmi.shell.LMIConnection.LMIConnection might prove more useful.

Namespace can also be overriden globally in a configuration file or with an option on command line.

[15] Application object is accessible through app property of each command instance.

Output rendering All these options begin with `FMT_` which is a shortcut for *formatter* as they become options to formatter objects. These can be defined not only in end-point commands but also in multiplexers. In the latter case they set the defaults for all their direct and indirect child commands.

Note: These options override configuration settings and command line options. Therefor use them with care.

They are:

FMT_NO_HEADINGS [`bool` (defaults to `False`)] Allows to suppress headings (column or row names) in the output.

> **Note:** With *LmiLister* command it's preferable to set the *COLUMNS* property to empty list instead. Otherwise associated function is expected to return column headers as a first row in its result.

FMT_HUMAN_FRIENDLY [`bool` (defaults to `False`)] Forces the output to be more pleasant to read by human beings.

Command specific properties Each command class can have its own specific properties. Let's take a look on them.

LmiCommandMultiplexer

COMMANDS [`dict` (mandatory)] Dictionary assigning subcommands to their names listed in usage string. Example follows:

```
class MyCommand(LmiCommandMultiplexer):
    '''
    My command description.

    Usage: %(cmd)s mycommand (subcmd1 | subcmd2)
    '''
    COMMANDS = {'subcmd1' : Subcmd1, 'subcmd2' : Subcmd2}
    OWN_USAGE = True
```

Where `Subcmd1` and `Subcmd2` are some other `LmiBaseCommand` subclasses. Documentation string must be parseable with docopt.

`COMMANDS` property will be translated to `child_commands()` class method by `MultiplexerMetaClass`.

FALLBACK_COMMAND [`lmi.scripts.common.command.endpoint.LmiEndPointCommand`] Command class used when no command defined in `COMMANDS` dictionary is passed on command line.

Take for example this usage string:

```
"""
Display hardware information.

Usage:
    %(cmd)s [all]
    %(cmd)s system
    %(cmd)s chassis
"""
```

This suggests there are tree commands defined taking care of listing hardware informations. Entry point definition could look like this:

```
class Hardware(command.LmiCommandMultiplexer):
    OWN_USAGE = __doc__      # usage string from above
    COMMANDS  = { 'all'      : All
```

```
                , 'system'   : System
                , 'chassis' : Chassis
                }
    FALLBACK_COMMAND = All
```

Without the `FALLBACK_COMMAND` property, the multiplexer would not handle the case when `'all'` argument is omitted as is suggested in the usage string. Adding it to command properties causes this multiplexer to behave exactly as `All` subcommand in case that no command is given on command line.

`LmiSelectCommand` properties Following properties allow to define profile and class requirements for commands.

`SELECT` [`list` (mandatory)] Is a list of pairs `(condition, command)` where `condition` is an expression in *LMIReSpL* language. And `command` is either a string with absolute path to command that shall be loaded or the command class itself.

Small example:

```
SELECT = [
      ( 'OpenLMI-Hardware < 0.4.2'
      , 'lmi.scripts.hardware.pre042.PreCmd'
      )
    , ('OpenLMI-Hardware >= 0.4.2 & class LMI_Chassis == 0.3.0'
      , HwCmd
      )
]
```

It says: Let the `PreHwCmd` command do the job on brokers having `openlmi-hardware` package older than `0.4.2`. Use the `HwCmd` anywhere else where also the `LMI_Chassis` CIM class in version `0.3.0` is available.

First matching condition wins and assigned command will be passed all the arguments. If no condition can be satisfied and no default command is set, an exception will be raised.

See also:

Definition of *LMIReSpL* mini-language: `parser`

`DEFAULT` [`string` or reference to command class] Defines fallback command used in case no condition in `SELECT` can be satisfied.

`LmiLister` properties

`COLUMNS` [`tuple`] Column names. It's a tuple with name for each column. Each row of data shall then contain the same number of items as this tuple. If omitted, associated function is expected to provide them in the first row of returned list. It's translated to `get_columns()` class method.

If set to empty list, no column headers will be printed. Every item of returned list of associated function will be treated as data. Note that setting this to empty list makes the *FMT_NO_HEADINGS* property redundant.

`LmiShowInstance` and `LmiInstanceLister` properties These two classes expect, as a result of their associated function, an instance or a list of instances of some CIM class. They take care of rendering them to standard output. Thus their properties affect the way how their properties are rendered.

`PROPERTIES` [`tuple`] Property names in the same order as the properties shall be listed. Items of this tuple can take multiple forms:

Property Name [str] Will be used for the name of column/property in output table and the same name will be used when obtaining the value from instance. Thus this form may be used only if the property name of instance can appear as a name of column.

(Column Name, Property Name) [(str, str)] This pair allows to render value of property under different name (*Column Name*).

(Column Name, getter) [(str, callable)] This way allows the value to be arbitrarily computed. The second item is a callable taking one and only argument – the instance of class to be rendered.

Example below shows different ways of rendering attributes for instances of LMI_Service CIM class:

```
class Show(command.LmiShowInstance):
    CALLABLE = 'lmi.scripts.service:get_instance'
    PROPERTIES = (
            'Name',
            ('Enabled', lambda i: i.EnabledDefault == 2),
            ('Active', 'Started'))
```

First property will be shown with the same label as the name of property. Second one modifies the value of EnabledDefault from int to bool representing enabled state. The last one uses different label for property name Started.

DYNAMIC_PROPERTIES [bool (defaults to False)] Whether the associated function is expected to return the properties tuple itself. If True, the result of associated function must be in form:

```
(properties, data)
```

Where properties have the same inscription and meaning as a PROPERTIES property of class.

Otherwise, only the data is expected.

Note: Both LmiShowInstance and LmiInstanceLister expect different data to be returned. See *LmiShowInstance* and *LmiInstanceLister* for more information.

Note: Omitting both PROPERTIES and DYNAMIC_PROPERTIES makes the LmiShowInstance render all attributes of instance. For LmiInstanceLister this is not allowed, either DYNAMIC_PROPERTIES must be True or PROPERTIES must be filled.

LmiCheckResult properties This command typically does not produce any output if operation succeeds. The operation succeeds if the result of associated function is expected. There are more ways how to say what is an expected result. One way is to use EXPECT property. The other is to provide very own implementation of check_result method.

EXPECT: (mandatory) Any value can be assigned to this property. This value is then expected to be returned by associated function. Unexpected result is treated as an error.

A callable object assigned here has special meaning. This object must accept exactly two parameters:

1. options - Dictionary with parsed command line options returned by docopt after being processed by transform_options().
2. result - Return value of associated function.

If the associated function does not return an expected result, an error such as:

```
There was 1 error:
host kvm-fedora-20
    0 != 1
```

will be presented to user which is not much helpful. To improve user experience, the `check_result` method could be implemented instead. Note the example:

```
class Update(command.LmiCheckResult):
    ARG_ARRAY_SUFFIX = '_array'

    def check_result(self, options, result):
        """
        :param list result: List of packages successfuly installed
            that were passed as an ''<package_array>'' arguments.
        """
        if options['<package_array>'] != result:
            return (False, ('failed to update packages: %s' %
                    ", ".join( set(options['<package_array>'])
                             - set(result))))
        return True
```

The `execute()` method is not listed to make the listing shorter. This command could be used with usage string such as:

```
%(cmd)s update [--force] [--repoid <repository>] <package> ...
```

In case of a failure, this would produce output like this one:

```
$ lmi sw update wt wt-doc unknownpackage
There was 1 error:
host kvm-fedora-20
    failed to update packages: unknownpackage
```

See also:

Docopt home page and its git: http://github.org/docopt/docopt.

3.1.2 LMIShell

LMIShell provides a (non)interactive way how to access CIM objects provided by *OpenPegasus* or *sblim-sfcb* broker. The shell is based on a python interpreter and added logic, therefore what you can do in pure python, it is possible in LMIShell. There are classes added to manipulate with CIM classes, instance names, instances, etc. Additional classes are added to fulfill wrapper pattern and expose only those methods, which are necessary for the purpose of a shell.

Short example:

```
$ lmishell
> c = connect('localhost', 'root', 'password')
> for proc in c.root.cimv2.LMI_Processor.instances():
...   print "Name:\t%s, Clock Speed:\t%s" % (proc.Name, proc.MaxClockSpeed)
...
Name:   QEMU Virtual CPU version 1.6.2, Clock Speed:    2000
Name:   QEMU Virtual CPU version 1.6.2, Clock Speed:    2000
```

Startup

By running the following, you will gain an interactive interface of the shell. The *LMIShell* is waiting for the end of an input to quit – by hitting `<ctrl+d>` you can exit from it:

```
$ lmishell
> <ctrl+d>
$
```

or:

```
$ lmishell
> quit()
$
```

Establish a connection

Following examples demonstrate, how to connect to a *CIMOM* by issuing a `connect()` call.

Username/Password authentication Common means of performing the authentication is done by providing a *username* and *password* to `connect()` function. See the following example:

```
> c = connect("host", "username")
password: # <not echoed>
>
```

or:

```
> c = connect("host", "username", "password")
>
```

Unix socket LMIShell can connect directly to the CIMOM using Unix socket. For this type of connection, the shell needs to be run under root user and the destination machine has to be either **localhost**, **127.0.0.1** or **::1**. This type of connection is supported by TOG-Pegasus and there needs to be a Unix socket file present at `/var/run/tog-pegasus/cimxml.socket`. If the condition is not met, classic username/password method will be used.

See following example:

```
> c = connect("localhost")
>
```

Credentials validation Function `connect()` returns either `LMIConnection` object, if the connection can be established, otherwise `None` is returned. Suppose, the LMIShell is run in verbose mode (*`-v`*, *`--verbose`*, *`-m`* or *`--more-verbose`* is used). See following example of creating a connection:

```
> # correct username or password
> c = connect("host", "username", "password")
INFO: Connected to host
> isinstance(c, LMIConnection)
True
> # wrong login username or password
> c = connect("host", "wrong_username", "wrong_password")
ERROR: Error connecting to host, <error message>
> c is None
True
>
```

NOTE: By default, LMIShell prints out only error messages, when calling a `connect()`; no INFO messages will be print out. It is possible to suppress all the messages by passing *`-q`* or *`--quiet`*).

Server's certificate validation

When using https transport protocol, LMIShell tries to validate each server-side certificate against platform provided *CA trust store*. It is necessary to copy the server's certificate from each CIMOM to the platform specific trust store directory.

NOTE: It is possible to make LMIShell skip the certificate validation process by **lmishell** *-n* or *--noverify*.

See following example:

```
$ lmishell --noverify
>
```

Namespaces

Namespaces in CIM and LMIShell provide a natural way, how to organize all the available classes and their instances. In the shell, they provide a hierarchic access point to other namespaces and corresponding classes.

The *root* namespace plays a special role in the managed system; it is the first entry point from the connection object and provides the access to other clamped namespaces.

Available namespaces

To get a `LMINamespace` object for the root namespace of the managed system, run following:

```
> root_namespace = c.root
>
```

To list all available namespace from the root one, run following code:

```
> c.root.print_namespaces()
...
> ns_lst = c.root.namespaces
>
```

If you want to access any namespace deeper (e.g. *cimv2*), run this:

```
> cimv2_namespace = c.root.cimv2
> cimv2_namespace = c.get_namespace("root/cimv2")
>
```

Available classes

Each namespace object can print its available classes. To print/get the list of the classes, run this:

```
> c.root.cimv2.print_classes()
...
> classes_lst = c.root.cimv2.classes()
>
```

Queries

Using a `LMINamespace` object, it is possible to retrieve a list of `LMIInstance` objects. The LMIShell supports 2 query languages:

- *WQL*
- *CQL*

Following code illustrates, how to execute *WQL* and *CQL* queries:

```
> instances_lst = namespace.wql("query")
> instances_lst = namespace.cql("query")
>
```

Classes

Each `LMIClass` in LMIShell represents a class implemented by a certain provider. You can get a list of its properties, methods, instances, instance names and ValueMap properties. It is also possible to print a documentation string, create a new instance or new instance name.

Getting a class object

To get a class which is provided by a broker, you can do following:

```
> cls = c.root.cimv2.ClassName
>
```

Fetching a class

Objects of `LMIClass` use lazy fetching method, because some methods do not need the `wbem.CIMClass` object.

To manually fetch the `wbem.CIMClass` object, call following:

```
> cls.fetch()
>
```

The methods, which need the `wbem.CIMClass` object to be fetched from CIMOM, do this action automatically, without the need of calling `LMIClass.fetch()` method by hand.

Class Methods

Following example illustrates, how to work with `LMIClass` methods:

```
> cls.print_methods()
...
> cls_method_lst = cls.methods()
>
```

Class Properties

To get a list of properties of a specific class, run following code:

```
> cls.print_properties()
...
> cls_property_lst = cls.properties()
>
```

Instances

Following part described basic work flow with `LMIInstance` and `LMIInstanceName` objects.

Get Instances Using a class object, you can access its instances. You can easily get a list of (filtered) instances, or the first one from the list. The filtering is uses input dictionary, if present, where the dictionary keys represent the instance properties and the dictionary values represent your desired instance property values.

To get `LMIInstance` object, execute the following example:

```
> inst = cls.first_instance()
> inst_lst = cls.instances()
>
```

Get Instance Names The `wbem.CIMInstanceName` objects clearly identify `wbem.CIMInstance` objects. LMIShell can retrieve `LMIInstanceName` objects, by calling following:

```
> inst_name = cls.first_instance_name()
> inst_names_lst = cls.instance_names()
>
```

Filtering Both methods `LMIClass.instances()` or `LMIClass.instance_names()` can filter returned objects by their keys/values. The filtering is achieved by passing a dictionary of `{property : value}` to the corresponding method. See following example:

```
> inst_lst = cls.instances({"FilterProperty" : FilterValue})
> inst_names_lst = cls.instance_names({"FilterProperty" : FilterValue})
>
```

New Instance Name LMIShell is able to create a new wrapped `wbem.CIMInstanceName`, if you know all the primary keys of a remote object. This instance name object can be then used to retrieve the whole instance object.

See the next example:

```
> inst_name = cls({Property1 : Value1, Property2 : Value2, ...})
> inst = inst_name.to_instance()
>
```

Creating a new instance LMIShell is able to create an object of specific class, if the provider support this operation.

See the following example:

```
> cls.create_instance({"Property1" : Value1, "Property2" : Value2})
>
```

NOTE: `Value` can be a `LMIInstance` object, as well. LMIShell will auto-cast such object.

ValueMap Properties

A CIM class may contain *ValueMap* properties (aliases for constant values) in its MOF definition. These properties contain constant values, which can be useful, when calling a method, or checking a returned value.

ValueMap properties are formed from 2 MOF properties of a class definition:

- *Values* – list of string names of the "constant" values
- *ValueMap* – list of values

Get ValueMap properties To get a list of all available constants, their values, use the following code:

```
> cls.print_valuemap_properties()
...
> valuemap_properties = cls.valuemap_properties()
...
> cls.PropertyValues.print_values()
...
>
```

NOTE: The suffix "**Values**" provides a way, how to access ValueMap properties.

Get ValueMap property value Following example shows, how to retrieve a constant value:

```
> constant_value_names_lst = cls.PropertyValues.values()
> cls.PropertyValues.ConstantValueName
ConstantValue
> cls.PropertyValues.value("ConstantValueName")
ConstantValue
>
```

Get ValueMap property value name LMIShell can also return string representing constant value. See the following code:

```
> cls.PropertyValue.value_name(ConstantValue)
'ConstantValueName'
>
```

Useful Properties

Following part describes few useful `LMIClass` properties.

Class Name Every class object can return a name of the CIM class, see following:

```
> cls.classname
ClassName
>
```

Namespace Every class belongs to certain namespace, to get a string containing the corresponding namespace for each class, run following:

```
> cls.namespace
Namespace
>
```

Connection Object This property returns a connection object, which was used to retrieve the class (refer to *Establish a connection*). See next example:

```
> cls.connection
LMIConnection(URI='uri', user='user'...)
>
```

Wrapped Object This property returns a wrapped `wbem` object. See the example:

```
> instance.wrapped_object
CIMClass(u'ClassName', ...)
>
```

Documentation

To see a class documentation (based on *MOF* definitions), run:

```
> cls.doc()
# ... pretty verbose output displayed in a pages (can be modified by
#     setting environment variable PAGER) ...
>
```

Instances

Each `LMIInstance` in LMIShell represents a CIM instance provided by a certain provider.

Operations that can be performed within a `LMIInstance`:

- get and set properties
- list/print/execute its methods
- print a documentation string
- get a list of associated objects
- get a list of association objects
- push (update) a modified object to CIMOM
- delete a single instance from the CIMOM.

Instance Methods

To get a list of methods, run following:

```
> instance.print_methods()
...
> method_lst = instance.methods()
>
```

To execute a method within an object, run this:

```
> instance.Method(
...     {"Param1" : value1,
...      "Param2" : value2, ...})
LMIReturnValue(
    rval=ReturnValue,
    rparams=ReturnParametersDictionary,
    errorstr="Possible error string"
```

```
)
>
```

NOTE: Instances **do not** auto-refresh after a method calls. It is necessary to perform this operation by hand (See *Instance refreshing*).

To get the result from a method call, see following:

```
> rval, rparams, errorstr = instance.Method(
...     {"Param1" : value1,
...      "Param2" : value2, ...})
>
```

The tuple in the previous example will contain return value of the method call (`rval`), returned parameters (`rparams`) and possible error string (`errorstr`).

Synchronous methods LMIShell can perform synchronous method call, which means, that the LMIShell is able to synchronously wait for a Job object to change its state to *Finished* state and then return the job's return parameters.

Most of the implemented methods in OpenLMI providers are *asynchronous* methods, which means that user can execute such method and do other desired actions while waiting for the result(s).

LMIShell can perform the synchronous method call, if the given method returns a object of following classes:

- `LMI_SELinuxJob`
- `LMI_StorageJob`
- `LMI_SoftwareInstallationJob`
- `LMI_SoftwareVerificationJob`
- `LMI_NetworkJob`

LMIShell first tries to use indications as the waiting method. If it fails, then it uses polling method instead.

Following example illustrates, how to perform a synchronous method call:

```
> rval, rparams, errorstr = instance.SyncMethod(
...     {"Param1" : value1,
...      "Param2" : value2, ...})
>
```

NOTE: See the prefix *Sync* of a method name.

When a synchronous method call is done:

- `rval` will contain the job's return value
- `rparams` will contain the job's return parameters
- `errorstr` will contain job's possible error string

It is possible to force LMIShell to use only polling method, see the next example:

```
> rval, rparams, errorstr = instance.SyncMethod(
...     {"Param1" : value1,
...      "Param2" : value2, ...},
...     PreferPolling=True)
>
```

Signal handling LMIShell can properly handle *SIGINT* and *SIGTERM*, which instruct the shell to cancel the synchronous call. When such signal is received, the background job, for which the LMIShell is waiting, will be asked to terminate, as well.

Instance Properties

To get a list of properties, see following:

```
> instance.print_properties()
...
> instance_prop_lst = instance.properties()
>
```

It is possible to access an instance object properties. To get a property, see the following example:

```
> instance.Property
PropertyValue
>
```

To modify a property, execute following:

```
> instance.Property = NewPropertyValue
> instance.push()
LMIReturnValue(rval=0, rparams={}, errorstr="")
>
```

NOTE: If you change an instance object property, you have to execute a `LMIInstance.push()` method to propagate the change to the CIMOM.

ValueMap Parameters

A CIM Method may contain *ValueMap* parameters (aliases for constant values) in its *MOF* definition.

To access these parameters, which contain constant values, see following code:

```
> instance.Method.print_valuemap_parameters()
...
> valuemap_parameters = instance.Method.valuemap_parameters()
>
```

Get ValueMap parameter value By using a *ValueMap* parameters, you can retrieve a constant value defined in the *MOF* file for a specific method.

To get a list of all available constants, their values, use the following code:

```
> instance.Method.ParameterValues.print_values()
...
>
```

NOTE: The suffix *Values* provides a way, how to access *ValueMap* parameters.

To retrieve a constant value, see the next example:

```
> constant_value_names_lst = instance.Method.ParameterValues.values()
> instance.Method.ParameterValues.ConstantValueName
ConstantValue
> instance.Method.ParameterValues.value("ConstantValueName")
```

```
ConstantValue
>
```

Get ValueMap parameter Method can also contain a mapping between constant property name and corresponding value. Following code demonstrates, how to access such parameters:

```
> instance.Method.ConstantValueName
>
```

Get ValueMap parameter value name LMIShell can also return string representing constant value. See the following code:

```
> instance.Method.ParameterValue.value_name(ConstantValue)
ConstantValueName
>
```

Instance refreshing

Local objects used by LMIShell, which represent CIM objects at CIMOM side, can get outdated, if such object changes while working with LMIShell's one.

To update object's properties, methods, etc. follow the next example:

```
> instance.refresh()
LMIReturnValue(rval=True, rparams={}, errorstr="")
>
```

Instance deletion

A single instance can be removed from the CIMOM by executing:

```
> instance.delete()
True
>
```

NOTE: After executing the `LMIInstance.delete()` method, all the object properties, methods will become inaccessible.

Deletion of the instance can be verified by:

```
> instance.is_deleted
True
>
```

Documentation

For an instance object, you can also use a documentation method, which will display verbose information of its properties and values.

See next example:

```
> instance.doc()
# ... pretty verbose output displayed in a pages (can be modified by
#     setting environment variable PAGER) ...
>
```

MOF representation

An instance object can also print out its *MOF* representation. This can be achieved by running:

```
> instance.tomof()
... verbose output of the instance in MOF syntax ...
>
```

Useful Properties

Following part describes `LMIInstance` useful properties.

Class Name Each instance object provide a property, that returns its class name. To get a string of the class name, run following:

```
> instance.classname
ClassName
>
```

Namespace Each instance object also provides a property, that returns a namespace name. To get a string of the namespace name, run following:

```
> instance.namespace
Namespace
>
```

Path To retrieve a unique, wrapped, identification object for the instance, `LMIInstanceName`, execute following:

```
> instance.path
LMIInstanceName(classname="ClassName"...)
>
```

Connection Object This property returns a connection object, which was used to retrieve the instance (refer to *Establish a connection*). See next example:

```
> instance.connection
LMIConnection(URI='uri', user='user'...)
>
```

Wrapped Object This property returns a wrapped `wbem` object. See the example:

```
> instance.wrapped_object
CIMInstance(classname=u'ClassName', ...)
>
```

Instance Names

`LMIInstanceName` is a object, which holds a set of primary keys and their values. This type of object exactly identifies an instance.

Key properties

To get a list of key properties, see following example:

```
> instance_name.print_key_properties()
...
> instance_name.key_properties()
...
> instance_name.SomeKeyProperty
...
>
```

Instance Names deletion

A single instance can be removed from the CIMOM by executing:

```
> instance_name.delete()
True
>
```

NOTE: After executing the `LMIInstanceName.delete()` method, all the object key properties, methods will become inaccessible.

Deletion of the instance can be verified by:

```
> instance_name.is_deleted
True
>
```

Conversion to a LMIInstance

This type of object may be returned from a method call. Each instance name can be converted into the instance, see next example:

```
> instance = instance_name.to_instance()
>
```

Useful Properties

Following part describes `LMIInstanceName` useful properties.

Class Name The property returns a string representation of the class name. See next example:

```
> instance_name.classname
ClassName
>
```

Namespace The property returns a string representation of namesapce. See next example:

```
> instance_name.namespace
Namespace
>
```

Host Name This property returns a string representation of the host name, where the CIM instance is located.

```
> instance_name.hostname
Hostname
>
```

Connection Object This property returns a connection object, which was used to retrieve the instance name (refer to *Establish a connection*). See next example:

```
> instance.connection
LMIConnection(URI='uri', user='user'...)
>
```

Wrapped Object This property returns a wrapped `wbem` object. See the example:

```
> instance.wrapped_object
CIMInstanceName(classname='ClassName', keybindings=NocaseDict(...), host='hostname', namespace='names
>
```

Associated Objects

An association from CIM perspective is a type of class that contains two or more references. Associations represent relationships between two or more classes.

Associations are classes which establish a relationship between classes without affecting any of the related classes. In other words, the addition of an association has no effect on any of the related classes.

Following text describes the means of retrieving associated objects within a given one.

Associated Instances

To get a list of associated `LMIInstance` objects with a given object, run following:

```
> associated_objects = instance.associators(
...     AssocClass=cls,
...     ResultClass=cls,
...     ResultRole=role,
...     IncludeQualifiers=include_qualifiers,
...     IncludeClassOrigin=include_class_origin,
...     PropertyList=property_lst)
> first_associated_object = instance.first_associator(
...     AssocClass=cls,
...     ResultClass=cls,
...     ResultRole=role,
...     IncludeQualifiers=include_qualifiers,
...     IncludeClassOrigin=include_class_origin,
...     PropertyList=property_lst))
```

Associated Instance Names

To get a list of associated `LMIInstanceName` objects with a given object, run following:

```
> associated_object_names = instance.associator_names(
...     AssocClass=cls,
...     ResultClass=cls,
...     Role=role,
...     ResultRole=result_role)
> first_associated_object_name = instance.first_associator_name(
...     AssocClass=cls,
...     ResultClass=cls,
...     Role=role,
...     ResultRole=result_role)
>
```

Association Objects

CIM defines an association relationship between managed objects. Following text describes the means of retrieving association objects within a given one. An association object is the object, which defines the relationship between two other objects.

Association Instances

To get association `LMIInstance` objects that refer to a particular target object, run following:

```
> association_objects = instance.references(
...     ResultClass=cls,
...     Role=role,
...     IncludeQualifiers=include_qualifiers,
...     IncludeClassOrigin=include_class_origin,
...     PropertyList=property_lst)
> first_association_object = instance.first_reference(
...     ResultClass=cls,
...     Role=role,
...     IncludeQualifiers=include_qualifiers,
...     IncludeClassOrigin=include_class_origin,
...     PropertyList=property_lst)
>
```

Association Instance Names

To get a list of association `LMIInstanceName` objects, run following:

```
> association_object_names = instance.reference_names(
...     ResultClass=cls,
...     Role=role)
> first_association_object_name = instance.first_reference_name(
...     ResultClass=cls,
...     Role=role)
>
```

Indications

From CIM point of view, an *indication* is the representation of the occurrence of an event. Indications are classes so they can have properties and methods. Instances of an indication are transient and can not be obtained by using CIM Operations, such as `GetInstance()` or `EnumerateInstances()`. Indications can only be received by subscribing to them.

An indication subscription is performed by the creation of an *CIM_IndicationSubscription* association instance that references an *CIM_IndicationFilter* (a filter) instance, and an *CIM_IndicationHandler* (a handler) instance. The filter contains the query that selects an *Indication* class or classes. The size and complexity of the result delivered to the destination is defined by the query.

LMIShell can perform an indication subscription, by which we can receive such event responses. The shell also provides a mechanism for the indication reception.

Indication handler

When working with indications, the first step is to set up an indication handler. This is a routine that will be triggered when the CIMOM sends us an indication for which we have subscribed (see below). It is important to set up the handler first so that we can generate a unique registration name and avoid conflicts with other clients that may wish to register for the same indication. The indication handler may be part of the same process that will initiate the provider registration or it may be an independent script, but the unique registration name must be acquired first in either case.

The following example describes creating a handler and a listener for an indication:

```
> def handler(indication, arg1, arg2, **kwargs):
...     """
...     Indication handler.
...
...     :param wbem.CIMInstance indication: exported wbem.CIMInstance
...     :param arg1: ...
...     :param arg2: ...
...     ...
...     """
...     do_something_with(indication)
> listener = LMIIndicationListener(listening_address, listening_port, certfile, keyfile, trust_store)
> unique_name = listener.add_handler("indication-name-XXXXXXXX", handler, arg1, arg2, **kwargs)
> listener.start()
>
```

The first argument of the handler is a `wbem.CIMInstance` object; the exported indication. The other arguments are handler-specific; Any number of arguments may be specified as necessary; those arguments must then be provided to the `LMIIndicationListener.add_handler()` method of the listener. In the above example, the string used in the `LMIIndicationListener.add_handler()` call is specified with, at least, eight **"X"** characters. Those characters will be replaced by unique string, which is generated by the listeners to avoid a handler name clash. Use of this uniqueness capability is not mandatory but is highly recommended. The substituted name is returned as the result of the `LMIIndicationListener.add_handler()` method so it can be used later.

When all necessary handlers are registered, the listener can be started by calling `LMIIndicationListener.start()`.

When a secure connection is desired, `LMIIndicationListener` can be constructed with *keyfile*, *certfile* and *trust_store* (paths to X509 certificate, private key in PEM format and trust store).

Subscribing to an indication

Now, when the indication listener is up and running, the indication subscription can be done. The LMIShell is capable of creating an indication subscription with the filter and handler objects in one single step.

Example of indication subscription with 3 mandatory arguments:

```
> c = connect("host", "privileged_user", "password")
> c.subscribe_indication(
...     Name=unique_name,
...     Query='SELECT * FROM CIM_InstModification',
...     Destination="http://192.168.122.1:%d" % listening_port
...  )
LMIReturnValue(rval=True, rparams={}, errorstr="")
>
```

NOTE: Make sure, that you an account which has write privileges in the *root/interop* namespace.

The indication subscription can created with an extensive list of arguments, where optional arguments can be specified:

- *QueryLanguage*: *DMTF:CQL*
- *CreationNamespace*: *root/interop*
- *SubscriptionCreationClassName*: *CIM_IndicationSubscription*
- *FilterCreationClassName*: *CIM_IndicationFilter*
- *FilterSystemCreationClassName*: *CIM_ComputerSystem*
- *FilterSourceNamespace*: *root/cimv2*
- *HandlerCreationClassName*: *CIM_IndicationHandlerCIMXML*
- *HandlerSystemCreationClassName*: *CIM_ComputerSystem*

```
> c = connect("host", "privileged_user", "password")
> c.subscribe_indication(
...     QueryLanguage="WQL",
...     Query='SELECT * FROM CIM_InstModification',
...     Name=unique_name,
...     CreationNamespace="root/interop",
...     SubscriptionCreationClassName="CIM_IndicationSubscription",
...     FilterCreationClassName="CIM_IndicationFilter",
...     FilterSystemCreationClassName="CIM_ComputerSystem",
...     FilterSourceNamespace="root/cimv2",
...     HandlerCreationClassName="CIM_IndicationHandlerCIMXML",
...     HandlerSystemCreationClassName="CIM_ComputerSystem",
...     Destination="http://192.168.122.1:%d" % listening_port
...  )
LMIReturnValue(rval=True, rparams={}, errorstr="")
>
```

In this state, we have a indication subscription created.

Auto-delete subscriptions By default all subscriptions created by LMIShell will be **auto-deleted**, when the shell quits. To change this behavior, you can pass `Permanent=True` keyword parameter to `LMIConnection.subscribe_indication()` call, which will prevent LMIShell from deleting the subscription.

Listing subscribed indications

To list all the subscribed indications, run following code:

```
> c.print_subscribed_indications()
...
> subscribed_ind_lst = c.subscribed_indications()
>
```

Unsubscribing from an indication

By default, the subscriptions created by the shell are auto-deleted, when the shell quits.

If you want to delete the subscriptions sooner, you can use the following methods:

To unsubscribe from a specific indication:

```
> c.unsubscribe_indication(unique_name)
LMIReturnValue(rval=True, rparams={}, errorstr="")
```

Or to unsubscribe from all indications:

```
> c.unsubscribe_all_indications()
>
```

Return Values

Method calls return an object, that represents a return value of the given method. This type of object can be converted into python's typical 3-item tuple and consists of 3 items:

- `rval` – return value
- `rparams` – return value parameters
- `errorstr` – error string, if any

Following example shows, how to use and convert `LMIReturnValue` object to tuple:

```
> return_value = instance.MethodCall()
> return_value.rval
0
> return_value.rparams
[]
> return_value.errorstr

> (rval, rparams, errorstr) = return_value
> rval
0
> rparams
[]
> errorstr

>
```

Interactive Interface

This section covers some features, that are present in the interactive interface or are related to the LMIShell.

History

When using the interactive interface of the LMIShell, you can use up/down arrows to navigate in history of all the commands you previously used.

Clearing the history If you want to clear the history, simply run:

```
> clear_history()
>
```

Reversed search The LMIShell can also search in the history of commands by hitting <ctrl+r> and typing the command prefix (as your default shell does). See following:

```
(reverse-i-search)'connect': c = connect("host", "username")
```

Exception handling

Exception handling by the shell can be turned off – since then, all the exceptions need to be handled by your code. By default, LMIShell **handles** the exceptions and uses C-like return values (See section *Return Values*) To allow all the exceptions to propagate to your code, run this:

```
> use_exceptions()
>
```

To turn exception handling by the shell back on, run this:

```
> use_exceptions(False)
>
```

Cache

The LMIShell's connection objects use a temporary cache for storing CIM class names and CIM classes to save network communication.

The cache can be cleared, see following example:

```
> c.clear_cache()
>
```

The cache can be also turned off, see next example:

```
> c.use_cache(False)
>
```

Tab-completion

Interactive interface also supports tab-completion for basic programming structures and also for CIM objects (such as namespace, classes, methods and properties completion, etc).

Following code shows few examples:

```
> c = conn<tab>
> c = connect(

> lmi_service_class = c.root.c<tab>
> lmi_service_class = c.root.cimv2
> lmi_service_class = c.root.cimv2.lmi_ser<tab>
> lmi_service_class = c.root.cimv2.LMI_Service

> sshd_service = lmi_s<tab>
> sshd_service = lmi_service_class

> sshd_service.Stat<tab>
> sshd_service.Status

> sshd_service.Res<tab>
> sshd_service.RestartService(

> lmi_service_class.Req<tab>
> lmi_service_class.RequestedStateChangeValues
> lmi_service_class.RequestesStateChangeValues.Sh<tab>
> lmi_service_class.RequestedStateChangeValues.Shutdown
> # similar for method calls, as well
>
```

Builtin features

This section describes built-in features of the LMIShell.

Configuration file

The LMIShell has a tiny configuration file with location `~/.lmishellrc`. In configuration file, you can set these properties:

```
# location of the history used by interactive mode
history_file = "~/.lmishell_history"
# length of history file, -1 for unlimited
history_length = -1
# default value for cache usage
use_cache = True
# default value for exceptions
use_exceptions = False
# default value for indication_cert_file
indication_cert_file = ""
# default value for indication_key_file
indication_key_file = ""
```

NOTE: `indication_cert_file` and `indication_key_file` are used by *Synchronous methods*, if the given method waits for an indication using `LMIIndicationListener`. Both configuration options may contain path to X509 certificate and private key in PEM format, respectively. If the configuration options are not set, SSL connection will not be used.

Inspecting a script

If you want to inspect a script after it has been interpreted by the LMIShell, run this:

```
$ lmishell -i some_script.lmi
# some stuff done
>
```

NOTE: Prefered extension of LMIShell's scripts is `.lmi`.

LMI Is Instance

LMIShell is able to verify, if a `LMIInstance` or `LMIInstanceName` object passed to `lmi_isinstance()` is a instance of `LMIClass`.

The function is similar to python's `isinstance()`:

```
> lmi_isinstance(inst, cls)
True/False
>
```

LMI Associators

LMIShell can speed up associated objects' traversal by manual joining, instead of calling `LMIInstance.associators()`. The call needs to get a list of **association** classes, for which the referenced objects will be joined. The list must contain objects of `LMIClass`.

See following example:

```
> associators = lmi_associators(list_of_association_classes)
>
```

3.1.3 OpenLMI Tools API reference

This is a generated documentation from *OpenLMI Tools* sources.

Generated from version: `0.10.1`

LMIShell API reference

This is a generated documentation from *LMIShell*'s sources.

Generated from version: `0.10.1`

LMIBaseObject

class `lmi.shell.LMIBaseObject.`**`LMIWrapperBaseObject`**`(`*conn*`)`

Base class for all LMI wrapper classes, such as `LMINamespace`, `LMIClass`, `LMIInstanceName`, `LMIInstance`, `LMIMethod`.

Parameters **conn** (*LMIConnection*) – connection object

`connection`

Property returning `LMIConnection` object.

Returns connection object

Return type `LMIConnection`

LMICIMXMLClient

class lmi.shell.LMICIMXMLClient.**LMICIMXMLClient**(*uri*, *username=''*, *password=''*, *verify_server_cert=True*, *key_file=None*, *cert_file=None*)

CIM-XML client.

Parameters

- **uri** (*string*) – URI of the CIMOM
- **username** (*string*) – account, under which, the CIM calls will be performed
- **password** (*string*) – user's password
- **verify_server_cert** (*bool*) – indicates, whether a server side certificate needs to be verified, if SSL used; default value is True
- **key_file** (*string*) – path to x509 key file; default value is None
- **cert_file** (*string*) – path to x509 cert file; default value is None

call_method(**args*, ***kwargs*)

Executes a method within a given instance.

Parameters

- **instance** – object, on which the method will be executed. The object needs to be instance of following classes:
  - wbem.CIMInstance
  - wbem.CIMInstanceName
  - LMIInstance
  - LMIInstanceName
- **method** (*string*) – string containing a method name
- **params** (*dictionary*) – parameters passed to the method call

Returns LMIReturnValue object with rval set to return value of the method call, rparams set to returned parameters from the method call, if no error occurs; otherwise rval is set to -1 and errorstr to appropriate error string

Raises CIMError, ConnectionError, TypeError

connect(**args*, ***kwargs*)

Connects to CIMOM.

NOTE: Applicable only wbem lmiwbem is used.

create_instance(**args*, ***kwargs*)

Creates a new wbem.CIMInstance object.

Parameters

- **classname** (*string*) – class name of a new instance
- **namespace** (*string*) – namespace, of the new instance
- **properties** (*dictionary*) – property names and values
- **qualifiers** (*dictionary*) – qualifier names and values
- **property_list** (*list*) – list for property filtering; see wbem.CIMInstance

Returns new `wbem.CIMInstance`, if no error occurs; otherwise None is returned

Raises `CIMError`, `ConnectionError`

delete_instance(**args*, ***kwargs*)
Deletes a `wbem.CIMInstance` from the CIMOM side.

Parameters **instance** – object to be deleted. The object needs to be instance of following classes:

- `wbem.CIMInstance`
- `wbem.CIMInstanceName`
- `LMIInstance`
- `LMIInstanceName`

Returns `LMIReturnValue` object with `rval` set to True, if no error occurs; otherwise `rval` is set to False and `errorstr` is set to corresponding error string

Raises `CIMError`, `ConnectionError`, `TypeError`

disconnect()
Disconnects from CIMOM.

NOTE: Applicable only wbem `lmiwbem` is used.

dummy()
Sends a "dummy" request to verify credentials.

Returns `LMIReturnValue` with rval set to True, if provided credentials are OK; False otherwise. If LMIShell uses exceptions, `CIMError` or `ConnectionError` will be raised.

Raises `CIMError`, `ConnectionError`

exec_query(**args*, ***kwargs*)
Executes a query and returns a list of `wbem.CIMInstance` objects.

Parameters

- **query_lang** (*string*) – query language
- **query** (*string*) – query to execute
- **namespace** (*string*) – target namespace for the query

Returns `LMIReturnValue` object with `rval` set to list of `wbem.CIMInstance` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` is set to corresponding error string

Raises `CIMError`, `ConnectionError`

get_associator_names(**args*, ***kwargs*)
Returns a list of associated `wbem.CIMInstanceName` objects with an input instance.

Parameters

- **instance** – for this object the list of associated `wbem.CIMInstanceName` will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.
- **limit** (*int*) – unused

Returns list of associated `wbem.CIMInstanceName` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`, `ConnectionError`, `TypeError`

get_associators(**args*, ***kwargs*)

Returns a list of associated `wbem.CIMInstance` objects with an input instance.

Parameters

- **instance** – for this object the list of associated `wbem.CIMInstance` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

- **IncludeQualifiers** (*bool*) – indicates, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – indicates, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the array define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If it is None, no additional filtering is defined.
- **limit** (*int*) – unused

Returns list of associated `wbem.CIMInstance` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`, `ConnectionError`, `TypeError`

get_class(**args*, ***kwargs*)

Returns a `wbem.CIMClass` object.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – – namespace name, from which the `wbem.CIMClass` should be retrieved; if None, default namespace will be used (**NOTE:** see `wbem`)
- **LocalOnly** (*bool*) – indicates, if only local members should be present in the returned `wbem.CIMClass`; any CIM elements (properties, methods, and qualifiers), except those added or overridden in the class as specified in the classname input parameter, shall not be included in the returned class.
- **IncludeQualifiers** (*bool*) – indicates, if qualifiers for the class (including qualifiers on the class and on any returned properties, methods, or method parameters) shall be included in the response.
- **IncludeClassOrigin** (*bool*) – indicates, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in the returned class.
- **PropertyList** (*list*) – if present and not None, the members of the list define one or more property names. The returned class shall not include elements for properties missing from this list. Note that if LocalOnly is specified as True, it acts as an additional filter on the set of properties returned. For example, if property A is included in the PropertyList but LocalOnly is set to True and A is not local to the requested class, it is not included in the response. If the PropertyList input parameter is an empty list, no properties are included in the response. If the PropertyList input parameter is None, no additional filtering is defined.

Returns `LMIReturnValue` object with rval set to `wbem.CIMClass`, if no error occurs; otherwise rval is set to None and errorstr to appropriate error string

Raises `CIMError`, `ConnectionError`

get_class_names(**args*, ***kwargs*)

Returns a list of class names.

Parameters

- **namespace** (*string*) – namespace, from which the class names list should be retrieved; if None, default namespace will be used (**NOTE:** see `wbem`)

- **ClassName** (*string*) – defines the class that is the basis for the enumeration. If the ClassName input parameter is absent, this implies that the names of all classes.
- **DeepInheritance** (*bool*) – if not present, of False, only the names of immediate child subclasses are returned, otherwise the names of all subclasses of the specified class should be returned.

Returns `LMIReturnValue` object with `rval` set to a list of strings containing class names, if no error occurs; otherwise `rval` is set to None and `errorstr` contains an appropriate error string

Raises `CIMError`, `ConnectionError`

get_instance(**args*, ***kwargs*)

Returns a `wbem.CIMInstance` object.

Parameters

- **path** – path of the object, which is about to be retrieved. The object needs to be instance of following classes:
  - `wbem.CIMInstanceName`
  - `wbem.CIMInstance`
  - `LMIInstanceName`
  - `LMIInstance`
- **LocalOnly** (*bool*) – indicates if to include the only elements (properties, methods, references) overridden or defined in the class
- **IncludeQualifiers** (*bool*) – indicates, if all Qualifiers for the class and its elements shall be included in the response
- **IncludeClassOrigin** (*bool*) – indicates, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in the returned class
- **PropertyList** (*list*) – if present and not None, the members of the list define one or more property names. The returned class shall not include elements for properties missing from this list. Note that if LocalOnly is specified as True, it acts as an additional filter on the set of properties returned. For example, if property A is included in the PropertyList but LocalOnly is set to True and A is not local to the requested class, it is not included in the response. If the PropertyList input parameter is an empty list, no properties are included in the response. If the PropertyList input parameter is None, no additional filtering is defined.

Returns `LMIReturnValue` object, where `rval` is set to `wbem.CIMInstance` object, if no error occurs; otherwise `rval` is set to None and `errorstr` is set to corresponding error string.

Raises `CIMError`, `ConnectionError`, `TypeError`

get_instance_names(**args*, ***kwargs*)

Returns a list of `wbem.CIMInstanceName` objects.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace name, where the instance names live
- **inst_filter** (*dictionary*) – dictionary containing filter values. The key corresponds to the primary key of the `wbem.CIMInstanceName`; value contains the filtering value.
- **limit** (*int*) – unused

- **kwargs** (*dictionary*) – supported keyword arguments (these are **deprecated**)
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** (*string*) – filtering value, see above

Returns `LMIReturnValue` object with `rval` contains a list of `wbem.CIMInstanceName` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` contains appropriate error string

Raises `LMIFilterError`, `CIMError`, `ConnectionError`

get_instances(**args*, ***kwargs*)

Returns a list of `wbem.CIMInstance` objects.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace, where the instances live
- **inst_filter** (*dictionary*) – dictionary containing filter values. The key corresponds to the primary key of the `wbem.CIMInstanceName`; value contains the filtering value.
- **client_filtering** (*bool*) – if True, client-side filtering will be performed, otherwise the filtering will be done by a CIMOM. Default value is False.
- **limit** (*int*) – unused
- **kwargs** (*dictionary*) – supported keyword arguments (these are **deprecated**)
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** (*string*) – filtering value, see above

Returns `LMIReturnValue` object with `rval` set to a list of `wbem.CIMIntance` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` is set to corresponding error string.

Raises `CIMError`, `ConnectionError`

get_reference_names(**args*, ***kwargs*)

Returns a list of association `wbem.CIMInstanceName` objects with an input instance.

Parameters

- **instance** – for this object the association `wbem.CIMInstanceName` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.
- **limit** (*int*) – unused

Returns list of association `wbem.CIMInstanceName` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`, `ConnectionError`, `TypeError`

get_references(**args*, ***kwargs*)

Returns a list of association `wbem.CIMInstance` objects with an input instance.

Parameters

- **instance** – for this object the list of association `wbem.CIMInstance` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter.
- **IncludeQualifiers** (*bool*) – bool flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – bool flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If PropertyList is None, no additional filtering is defined.
- **limit** (*int*) – unused

Returns list of association `wbem.CIMInstance` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`, `ConnectionError`, `TypeError`

get_superclass(*classname*, *namespace=None*)

Returns a superclass to given class.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace name

Returns superclass to given class, if such superclass exists, None otherwise

Raises `CIMError`, `ConnectionError`

hostname

Returns hostname of CIMOM

Return type string

modify_instance(**args*, ***kwargs*)
Modifies a `wbem.CIMInstance` object at CIMOM side.

Parameters

- **instance** (*wbem.CIMInstance*) – object to be modified
- **IncludeQualifiers** (*bool*) – indicates, if the qualifiers are modified as specified in *ModifiedInstance*.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Only properties specified in the PropertyList are modified. Properties of the *ModifiedInstance* that are missing from the PropertyList are ignored. If the PropertyList is an empty list, no properties are modified. If the PropertyList is None, the set of properties to be modified consists of those of *ModifiedInstance* with values different from the current values in the instance to be modified.

Returns `LMIReturnValue` object with `rval` set to 0, if no error occurs; otherwise `rval` is set to -1 and `errorstr` is set to corresponding error string.

Raises `CIMError`, `ConnectionError`

uri

Returns URI of the CIMOM

Return type string

username

Returns user name as a part of provided credentials

Return type string

LMIClass

class `lmi.shell.LMIClass.`**LMIClass**(*conn*, *namespace*, *classname*)
LMI wrapper class representing `wbem.CIMClass`.

Parameters

- **conn** (*LMIConnection*) – connection object
- **namespace** (*LMINamespace*) – namespace object
- **classname** (*string*) – CIM class name

classname

Returns class name

Return type string

create_instance(*self_wr*, **args*, ***kwargs*)
Creates a new `wbem.CIMInstance` at the server side and returns `LMIReturnValue` object containing `LMIInstance` as a result.

Parameters

- **properties** (*dictionary*) – initial properties with corresponding values
- **qualifiers** (*dictionary*) – initial qualifiers
- **property_list** (*list*) – list of properties, which should be present in `LMIInstance` object

Usage: See *Creating a new instance*.

doc (*self_wr*, **args*, ***kwargs*)
Prints out pretty verbose message with documentation for the class. If the LMIShell is run in a interactive mode, the output will be redirected to a pager set by environment variable `PAGER`. If there is not `PAGER` set, less or more will be used as a fall-back.

fetch (**args*, ***kwargs*)
Manually fetches a wrapped `wbem.CIMClass` object.

Parameters **full_fetch** (*bool*) – True, if `wbem.CIMClass` should include qualifiers and class origin. Default value is False.

Raises `CIMError`, `ConnectionError`

Usage: See *Fetching a class*.

first_instance (*inst_filter=None*, *client_filtering=False*, ***kwargs*)
Returns the first `LMIInstance` of the corresponding class.

Parameters

- **inst_filter** (*dictionary*) – filter values, where the key corresponds to the key of `wbem.CIMInstance`; value contains the filtering value.
- **client_filtering** (*bool*) – if True, client-side filtering will be performed, otherwise the filtering will be done by a CIMOM. Default value is False.
- **kwargs** (*dictionary*) – deprecated keyword arguments
  - **Key** or **key** – filtering key, see above
  - **Value** or **value** – filtering value, see above

Returns first `LMIInstance` object

Usage: See *Get Instances* and *Filtering*.

first_instance_name (*inst_filter=None*, ***kwargs*)
Returns the first `LMIInstanceName` of the corresponding class.

Parameters

- **inst_filter** (*dictionary*) – filter values, where the key corresponds to the primary key of `wbem.CIMInstanceName`; value contains the filtering value
- **kwargs** (*dictionary*) – deprecated keyword arguments
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** – filtering value, see above

Returns first `LMIInstanceName` object

Usage: See *Get Instance Names* and *Filtering*.

instance_names (*self_wr*, **args*, ***kwargs*)
Returns a LMIReturnValue containing a list of LMIInstanceNames.

Parameters

- **inst_filter** (*dictionary*) – filter values. The key corresponds to the primary key of the `wbem.CIMInstanceName`; value contains the filtering value
- **kwargs** (*dictionary*) – deprecated keyword arguments
  - **Key** or **key** (*string*) – filtering key, see above

- **Value** or **value** – filtering value, see above

Returns `LMIReturnValue` object with `rval` set to a list of `LMIInstanceName` objects

Usage: See *Get Instance Names* and *Filtering*.

instances (*self_wr*, **args*, ***kwargs*)
Returns a list of objects of `LMIInstance`.

Parameters

- **inst_filter** (*dictionary*) – filter values, where the key corresponds to the key of `wbem.CIMInstance`; value contains the filtering value
- **client_filtering** (*bool*) – if True, client-side filtering will be performed, otherwise the filtering will be done by a CIMOM. Default value is False.
- **kwargs** (*dictionary*) – deprecated keyword arguments
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** – filtering value, see above

Returns list of `LMIInstance` objects

Usage: See *Get Instances* and *Filtering*.

is_fetched (*full_fetch=False*)
Returns True, if `wbem.CIMClass` has been fetched.

Parameters **full_fetch** (*bool*) – defines, if qualifiers are also included

methods (*self_wr*, **args*, ***kwargs*)

Returns list of strings of `wbem.CIMClass` methods.

Usage: See *Class Methods*. **Note:** When caching is turned off, this method may consume some time.

namespace

Returns namespace name

Return type string

new_instance_name (*keybindings*)

Create new `LMIInstanceName` object by passing all the keys/values of the object.

Parameters **keybindings** (*dictionary*) – primary keys of instance name with corresponding values

Returns new `LMIInstanceName` object

Usage: See *New Instance Name*.

print_methods (*self_wr*, **args*, ***kwargs*)
Prints out the list of `wbem.CIMClass` methods.

Usage: See *Class Methods*.

print_properties (*self_wr*, **args*, ***kwargs*)
Prints out the list of `wbem.CIMClass` properties.

Usage: See *Class Properties*.

print_valuemap_properties(*self_wr*, **args*, ***kwargs*)
Prints out the list of string of constant names.

Usage: *Get ValueMap properties*.

properties(*self_wr*, **args*, ***kwargs*)

Returns list of strings of the `wbem.CIMClass` properties

Usage: See *Class Properties*.

valuemap_properties(*self_wr*, **args*, ***kwargs*)

Returns list of strings of the constant names

Usage: *Get ValueMap properties*.

wrapped_object

Returns wrapped `wbem.CIMClass` object

LMICompleter

class `lmi.shell.LMICompleter.`**LMICompleter**(*namespace=None*)
This LMIShell completer, which is used in the interactive mode, provides tab-completion for user friendliness.

Parameters **namespace** (*dictionary*) – dictionary, where to perform a completion. If unspecified, the default namespace where completions are performed is `__main__` (technically, `__main__.__dict__`).

attr_matches(*text*)

Parameters **text** (*string*) – expression to complete

Returns list of attributes of a given expression; if the expression is instance of LMI wrapper class, its important properties/attributes/ methods/parameters will be added too

Return type list of strings

complete(*text*, *state*)

Parameters

- **text** (*string*) – string to be completed.
- **state** – order number of the completion, see rlcompleter

Returns completed string

global_matches(*text*)

Parameters **text** (*string*) – expression to complete

Returns list of all keywords, built-in functions and names

Return type list of strings

LMIConnection

class `lmi.shell.LMIConnection.`**LMIConnection**(*uri*, *username=''*, *password=''*, *interactive=False*, *use_cache=True*, *key_file=None*, *cert_file=None*, *verify_server_cert=True*)
Class representing a connection object. Each desired connection to separate CIMOM should have its own con-

nection object created. This class provides an entry point to the namespace/classes/instances/methods hierarchy present in the LMIShell.

Parameters

- **uri** (*string*) – URI of the CIMOM
- **username** (*string*) – account, under which, the CIM calls will be performed
- **password** (*string*) – user's password
- **interactive** (*bool*) – flag indicating, if the LMIShell client is running in the interactive mode; default value is False.
- **use_cache** (*bool*) – flag indicating, if the LMIShell client should use cache for CIMClass objects. This saves lot's of communication, if there are `EnumerateInstances()` and `EnumerateClasses()` intrinsic methods often issued. Default value is True.
- **key_file** (*string*) – path to x509 key file; default value is None
- **cert_file** (*string*) – path to x509 cert file; default value is None
- **verify_server_cert** (*bool*) – flag indicating, whether a server side certificate needs to be verified, if SSL used; default value is True

NOTE: If interactive is set to True, LMIShell will:

- prompt for username and password, if missing and connection via Unix socket can not be established.
- use pager for the output of: `LMIInstance.doc()`, `LMIClass.doc()`, `LMIInstance.tomof()` and `LMIMethod.tomof()`

clear_cache()
Clears the cache.

client

Returns CIMOM client

Return type `LMICIMXMLClient` or `LMIWSMANClient`

connect()
Connects to CIMOM and verifies credentials by performing a "dummy" request.

Returns `LMIReturnValue` object with rval set to True, if the user was properly authenticated; False otherwise. In case of any error, rval is set to False and errorstr contains appropriate error string.

Return type `LMIReturnValue`

disconnect()
Disconnects from CIMOM.

get_namespace(*namespace*)

Parameters **namespace** (*string*) – namespace path (eg. *root/cimv2*)

Returns `LMINamespace` object

Raises `LMINamespaceNotFound`

hostname

Returns hostname of CIMOM

Return type string

is_wsman()
Returns True, if the connection is made with WSMAN CIMOM; False otherwise.

namespaces

Returns list of all available namespaces

Usage: *Available namespaces*.

print_namespaces()
Prints out all available namespaces.

print_subscribed_indications()
Prints out all the subscribed indications.

root

Returns `LMINamespaceRoot` object for *root* namespace

subscribe_indication(***kwargs*)
Subscribes to an indication. Indication is formed by 3 objects, where 2 of them (filter and handler) can be provided, if the LMIShell should not create those 2 by itself.

NOTE: Currently the call registers `atexit` hook, which auto-deletes all subscribed indications by the LMIShell.

Parameters **kwargs** (*dictionary*) – parameters for the indication subscription

- **Filter** (*LMIInstance*) – if provided, the `LMIInstance` object will be used instead of creating a new one; **optional**
- **Handler** (*LMIInstance*) – if provided, the `LMIInstance` object will be used instead of creating a new one; **optional**
- **Query** (*string*) – string containing a query for the indications filtering
- **QueryLanguage** (*string*) – query language; eg. *WQL*, or *DMTF:CQL*. This parameter is optional, default value is *DMTF:CQL*.
- **Name** (*string*) – indication name
- **CreationNamespace** (*string*) – creation namespace. This parameter is optional, default value is *root/interop*.
- **SubscriptionCreationClassName** (*string*) – subscription object class name. This parameter is optional, default value is *CIM_IndicationSubscription*.
- **Permanent** (*bool*) – whether to preserve the created subscription on LMIShell's quit. Default value is False.
- **FilterCreationClassName** (*string*) – creation class name of the filter object. This parameter is options, default value is *CIM_IndicationFilter*.
- **FilterSystemCreationClassName** (*string*) – system creation class name of the filter object. This parameter is optional, default value is *CIM_ComputerSystem*.
- **FilterSourceNamespace** (*string*) – local namespace where the indications originate. This parameter is optional, default value is *root/cimv2*.
- **HandlerCreationClassName** (*string*) – creation class name of the handler object. This parameter is optional, default value is *CIM_IndicationHandlerCIMXML*.
- **HandlerSystemCreationClassName** (*string*) – system creation name of the handler object. This parameter is optional, default value is *CIM_ComputerSystem*.
- **Destination** (*string*) – destination URI, where the indications should be delivered

Returns `LMIReturnValue` object with `rval` set to True, if indication was subscribed; False otherwise. If a error occurs, `errorstr` is set to appropriate error string.

subscribed_indications()

Returns list of all the subscribed indications

timeout

Returns CIMOM connection timeout for a transaction (milliseconds)

Return type int

unsubscribe_all_indications()
Unsubscribes all the indications. This call ignores *Permanent* flag, which may be provided in `LMIConnection.subscribe_indication()`, and deletes all the subscribed indications.

unsubscribe_indication(*name*)
Unsubscribes an indication.

Parameters **name** (*string*) – indication name

Returns `LMIReturnValue` object with `rval` set to True, if unsubscribed; False otherwise

uri

Returns URI of the CIMOM

Return type string

use_cache(*active=True*)
Sets a bool flag, which defines, if the LMIShell should use a cache.

Parameters **active** (*bool*) – whether the LMIShell's cache should be used

`lmi.shell.LMIConnection.`**connect**(*uri*, *username=''*, *password=''*, *interactive=False*, *use_cache=True*, *key_file=None*, *cert_file=None*, *verify_server_cert=True*, *prompt_prefix=''*)
Creates a connection object with provided URI and credentials.

Parameters

- **uri** (*string*) – URI of the CIMOM
- **username** (*string*) – account, under which, the CIM calls will be performed
- **password** (*string*) – user's password
- **interactive** (*bool*) – flag indicating, if the LMIShell client is running in the interactive mode; default value is False.
- **use_cache** (*bool*) – flag indicating, if the LMIShell client should use cache for `wbem.CIMClass` objects. This saves lot's of communication, if there are `EnumerateInstances()` and `EnumerateClasses()` intrinsic methods often issued. Default value is True.
- **key_file** (*string*) – path to x509 key file; default value is None
- **cert_file** (*string*) – path to x509 cert file; default value is None
- **verify_server_cert** (*bool*) – flag indicating, whether a server side certificate needs to be verified, if SSL used; default value is True.
- **prompt_prefix** (*string*) – username and password prompt prefix in case the user is asked for credentials. Default value is empty string.

Returns `LMIConnection` object or None, if LMIShell does not use exceptions

Raises `ConnectionError`

NOTE: If interactive is set to True, LMIShell will:

- prompt for username and password, if missing and connection via Unix socket can not be established.
- use pager for the output of: `LMIInstance.doc()`, `LMIClass.doc()`, `LMIInstance.tomof()` and `LMIMethod.tomof()`

Usage: *Establish a connection*.

LMIConsole

class `lmi.shell.LMIConsole.`**LMIConsole**(*cwd_first_in_path=False*)

Class representing an interactive console.

clear_history()

Clears the current history.

interact(*locals=None*)

Starts the interactive mode.

Parameters **locals** (*dictionary*) – locals

interpret(*script_name*, *script_argv*, *locals=None*, *interactive=False*)

Interprets a specified script within additional provided locals. There are `LMIConsole.DEFAULT_LOCALS` present.

Parameters

- **script_name** (*string*) – script name
- **script_argv** (*list*) – script CLI arguments
- **locals** (*dictionary*) – dictionary with locals
- **interactive** (*bool*) – tells LMIShell, if the script should be treated as if it was run in interactive mode

Returns exit code of the script

Return type int

load_history()

Loads the shell's history from the history file.

save_history()

Saves current history of commands into the history file. If the length of history exceeds a maximum history file length, the history will be truncated.

set_verify_server_certificate(*verify_server_cert=True*)

Turns on or off server side certificate verification, if SSL used.

Parameters **verify_server_cert** (*bool*) – – flag which tells, whether a server side certificate needs to be verified, if SSL used

setup_completer()

Initializes tab-completer.

LMIConstantValues

class lmi.shell.LMIConstantValues.**LMIConstantValues**(*cim_obj*, *cast_type*)
Abstract class for constant value objects.

Parameters

- **cim_obj** – this object is either of type wbem.CIMParameter, wbem.CIMProperty or wbem.CIMMethod. Construction of this object requires to have a member _cast_type to properly cast CIM object. When constructing derived objects, make sure, that the mentioned member is present before calling this constructor.
- **cast_type** – parameter/property cast type

print_values()
Prints all available constant names.

Usage: *Get ValueMap properties*.

value(*value_name*)

Parameters **value_name** (*string*) – constant name

Returns constant value

Usage: *Get ValueMap property value*.

value_name(*value*)

Parameters **value** (*int*) – numeric constant value

Returns constant value

Return type string

Usage: *Get ValueMap property value name*.

values()

Returns list of all available constant values

values_dict()

Returns dictionary of constants' names and values

class lmi.shell.LMIConstantValues.**LMIConstantValuesMethodReturnType**(*cim_method*)
Derived class used for constant values of wbem.CIMMethod.

Parameters **cim_method** (*CIMMethod*) – wbem.CIMMethod object

class lmi.shell.LMIConstantValues.**LMIConstantValuesParamProp**(*cim_property*)
Derived class used for constant values of wbem.CIMProperty and wbem.CIMParameter.

Parameters **cim_property** – wbem.CIMProperty or wbem.CIMParameter object. Both objects have necessary member type which is needed for proper casting.

LMIDecorators

class lmi.shell.LMIDecorators.**lmi_class_fetch_lazy**(*full_fetch=False*)
Decorator for LMIClass, which first fetches a wrapped wbem.CIMClass object and then executes a wrapped method.

Parameters **full_fetch** (*bool*) – True, if wbem.CIMClass should include qualifiers and class origin. Default value is False.

class lmi.shell.LMIDecorators.**lmi_instance_name_fetch_lazy**(*full_fetch=False*)
Decorator for LMIInstanceName, which first fetches a wrapped wbem.CIMInstance object and then executes a wrapped method.

Parameters **full_fetch** (*bool*) – True, if wbem.CIMClass should include qualifiers and class origin. Default value is False.

class lmi.shell.LMIDecorators.**lmi_possibly_deleted**(*expr_ret*, *Self=False*, **expr_ret_args*, ***expr_ret_kwargs*)
Decorator, which returns None, if provided test expression is True.

Parameters

- **expr_ret** – callable or return value used, if expr_test fails
- **expr_ret_args** – expr_ret position arguments
- **expr_ret_kwargs** – expr_ret keyword arguments
- **Self** (*bool*) – flag, which specifies, if to pass self variable to the expr_ret, if expr_test failed

Example of usage:

```
class Foo:
    def __init__(self, deleted):
        self._deleted = deleted

    @lmi_possibly_deleted(lambda obj: obj._member, lambda: False)
    def some_method(self):
        print "some_method called"
        return True

f = Foo(None)
f.some_method() == False

f = Foo(True)
f.some_method() == True
```

class lmi.shell.LMIDecorators.**lmi_process_cim_exceptions**(*rval=None*, *error_callable=<function return_lmi_rval at 0x7f174a693050>*)
Decorator used for CIM-XML exception processing.

Parameters

- **rval** – rval passed to LMIReturnValue.__init__()
- **error_callable** – callable used for processing wbem.CIMError and ConnectionError

NOTE: callables need to take 2 arguments: return value and error string.

class lmi.shell.LMIDecorators.**lmi_process_cim_exceptions_rval**(*rval=None*)
Decorator used for CIM-XML exception processing.

Parameters **rval** – return value of a decorated method in case of exception

class lmi.shell.LMIDecorators.**lmi_process_wsman_exceptions**(*rval=None*, *error_callable=<function return_lmi_rval at 0x7f174a693050>*)
Decorator used for wsman exception processing.

Parameters

- **rval** – rval passed to `LMIReturnValue.__init__()`
- **error_callable** – callable used for processing `wbem.CIMError` and `ConnectionError`

NOTE: callables need to take 2 arguments: return value and error string.

class `lmi.shell.LMIDecorators.`**`lmi_process_wsman_exceptions_rval`**(*rval=None*)
Decorator used for wsman exception processing.

Parameters **rval** – return value of a decorated method in case of exception

class `lmi.shell.LMIDecorators.`**`lmi_return_expr_if_fail`**(*expr_test*, *expr_ret*, *Self=False*, **expr_ret_args*, ***expr_ret_kwargs*)
Decorator, which calls a specified expression and returns its return value instead of calling the decorated method, if provided test expression is False; otherwise a method is called.

Parameters

- **expr_test** – expression which determines, if to execute a return value expression
- **expr_ret** – expression, which is called, if the `expr_test` returns False
- **expr_ret_args** – `expr_ret` position arguments
- **expr_ret_kwargs** – `expr_ret` keyword arguments
- **Self** (*bool*) – flag, which specifies, if to pass `self` variable to the `expr_ret`, if `expr_test` failed

Example of usage:

```
class Foo:
    def __init__(self, member):
        self._member = member

    def failed(self):
        print "expression failed"
        return False

    # NOTE: the self parameter to the method call needs to be passed
    # via expr_ret_args, therefore, there is a dummy lambda obj: obj,
    # which is basically self variable.
    @lmi_return_expr_if_fail(lambda obj: obj._member, failed,
                             lambda obj: obj)
    def some_method(self):
        print "some_method called"
        return True

f = Foo(None)
f.some_method() == False

f = Foo(True)
f.some_method() == True
```

class `lmi.shell.LMIDecorators.`**`lmi_return_if_fail`**(*expr_test*)
Decorator, which returns None and no method call is performed, if provided test expression is False; otherwise a method is called.

Parameters **expr_test** – if the expression `expr_test` returns True, a method is called

Example of usage:

```
class Foo:
    def __init__(self, member):
        self._member = member

    @lmi_return_if_fail(lambda obj: obj._member)
    def some_method(self):
        print "some_method called"
        return True

f = Foo(None)
f.some_method() == None

f = Foo(True)
f.some_method() == True
```

class lmi.shell.LMIDecorators.**lmi_return_val_if_fail**(*expr_test*, *rval*)
Decorator, which returns a specified value and no method call is performed, if provided test expression is False; otherwise a method is called.

Parameters

- **expr_test** – if the expression returns False, a method call is called
- **rval** – return value of the method, if the object attribute as expression failed

Example of usage:

```
class Foo:
    def __init__(self, member):
        self._member = member

    @lmi_return_val_if_fail(lambda obj: obj._member, False)
    def some_method(self):
        print "some_method called"
        return True

f = Foo(None)
f.some_method() == False

f = Foo(True)
f.some_method() == True
```

LMIExceptions

exception lmi.shell.LMIExceptions.**CIMError**(**args*)
LMIShell's exception for CIM errors.

exception lmi.shell.LMIExceptions.**ConnectionError**(**args*)
LMIShell's exception for Connection errors.

exception lmi.shell.LMIExceptions.**LMIClassNotFound**(*namespace*, *class_name*)
Raised, when trying to access missing class in LMINamespace.

Parameters

- **namespace** (*string*) – namespace name
- **classname** (*string*) – class name, which was not found in **namespace**

exception lmi.shell.LMIExceptions.**LMIDeletedObjectError**
Raised, when there is an attempt to access properties on deleted LMIInstance object.

exception lmi.shell.LMIExceptions.**LMIFilterError**
Raised, when a filter error occurs, mostly when filter object is missing.

exception lmi.shell.LMIExceptions.**LMIHandlerNamePatternError**
Raised when the pattern string does not contain minimum replaceable characters.

exception lmi.shell.LMIExceptions.**LMIIndicationError**
Raised, if an error occurs while subscribing to/removing an indication.

exception lmi.shell.LMIExceptions.**LMIIndicationListenerError**
Raised, if there is an error while starting/stopping indication listener.

exception lmi.shell.LMIExceptions.**LMIMethodCallError**
Raised, when an error occurs within method call.

exception lmi.shell.LMIExceptions.**LMINamespaceNotFound**(*namespace*, **args*)
Raised, when trying to access not existing namespace from connection or namespace object.

Parameters

- **namespace** (*string*) – namespace which was not found
- **args** – other positional arguments

exception lmi.shell.LMIExceptions.**LMINoPagerError**
Raised, when there is no default pager like less or more.

exception lmi.shell.LMIExceptions.**LMINotSupported**
Raised, when non-WSMAN method is about to be called.

exception lmi.shell.LMIExceptions.**LMISynchroMethodCallError**
Raised, when an error occurs within synchronized method call.

exception lmi.shell.LMIExceptions.**LMISynchroMethodCallFilterError**
Raised, when the LMIShell can not find necessary static filter for synchronous method call.

exception lmi.shell.LMIExceptions.**LMIUnknownParameterError**
Raised, when there is a method call issued and unknown method parameter is provided.

exception lmi.shell.LMIExceptions.**LMIUnknownPropertyError**
Raised, when there is an attempt to create instance with unknown property provided.

LMIFormatter

class lmi.shell.LMIFormatter.**LMIClassFormatter**(*cim_class*)
Class formatter is used to print out wbem.CIMClass representation.

Parameters **cim_class** (*CIMClass*) – object to print out

format(*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*)
Formats out wbem.CIMClass object.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

format_property (*prop*, *indent*, *width*, *f*)
Prints out a property of `wbem.CIMClass`.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

class `lmi.shell.LMIFormatter.`**LMIFormatter**
Abstract class for derived subclasses.

fancy_format (*interactive*)
Formats a block of text. If the LMIShell is running in interactive mode, pager will be used, otherwise the output will be written to standard output.

Parameters **interactive** (*bool*) – defines, if to use pager

format (*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*)
Formats a block of text and prints it to the output stream.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

class `lmi.shell.LMIFormatter.`**LMIInstanceFormatter** (*cim_instance*)
Instance formatter is used to print out `wbem.CIMInstance` representation.

Parameters **cim_instance** (*CIMInstance*) – object to print out

format (*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*)
Prints out :py:class‘CIMInstance‘ object.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

format_property (*prop*, *indent*, *width*, *f*)
Prints out a property of `wbem.CIMInstance`.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

class `lmi.shell.LMIFormatter.`**LMIMethodFormatter** (*cim_method*)
Method formatter is used to print out `wbem.CIMMethod` representation.

format (*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*)
Prints out :py:class‘CIMMethod‘ object.

Parameters

- **indent** (*int*) – number of spaces to indent the text block

- **width** (*int*) – total text block width
- **f** – output stream

format_method(*method*, *indent*, *width*, *f*)
Prints out a method of `wbem.CIMClass`.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

format_parameter(*param*, *indent*, *width*, *f*)
Prints out a parameter of `wbem.CIMMethod`.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

format_qualifier(*qualif*, *indent*, *width*, *f*)
Prints out a parameter of `wbem.CIMMethod`.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

class `lmi.shell.LMIFormatter.`**LMIMofFormatter**(*obj*)
MOF formatter is used to print out MOF representation of a CIM object.

Parameters **obj** – object, which has `tomof()` method

format(*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*)
Formats a MOF object and prints it to the output stream.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

class `lmi.shell.LMIFormatter.`**LMITextFormatter**(*text*)
Text formatter class. Used when printing a block of text to output stream.

Parameters **text** (*string*) – text to be formatted

format(*indent=0*, *width=80*, *f=<open file '<stdout>', mode 'w' at 0x7f174f95a150>*, *separator=True*)
Formats a block of text and prints it to the output stream.

Parameters

- **indent** (*int*) – number of spaces to indent the text block
- **width** (*int*) – total text block width
- **f** – output stream

- **kwargs** (*dictionary*) – supported keyword arguments
- **separator** (*bool*) – if True, there will be a new line appended after the formatted text; default value is True

LMIHelper

class lmi.shell.LMIHelper.**LMIHelper**
LMI Helper class, which overrides python help.

LMIIndicationListener

class lmi.shell.LMIIndicationListener.**LMIIndicationListener**(*hostname*, *port*, *certfile=None*, *keyfile=None*, *trust_store=None*)
Class representing indication listener, which provides a unified API for the server startup and shutdown and for registering an indication handler.

Parameters

- **hostname** (*str*) – bind hostname
- **port** (*int*) – TCP port, where the server should listen for incoming messages
- **certfile** (*str*) – path to certificate file, if SSL used
- **keyfile** (*str*) – path to key file, if SSL used
- **trust_store** (*str*) – path to trust store

add_handler(*handler_name_pattern*, *handler*, **args*, ***kwargs*)
Registers a handler into the indication listener. Returns a string, which is used for the indication recognition, when a message arrives.

Parameters

- **handler_name_pattern** (*string*) – string, which may contain set of "**X**" characters at the end of the string. The "**X**" characters will be replaced by random characters and the whole string will form a unique string.
- **handler** – callable, which will be executed, when a indication is received
- **args** (*tuple*) – positional arguments for the handler
- **kwargs** (*dictionary*) – keyword arguments for the handler

Returns handler unique name

Return type string

LMIInstanceName

class lmi.shell.LMIInstanceName.**LMIInstanceName**(*conn*, *cim_instance_name*)
LMI wrapper class representing wbem.CIMInstanceName.

Parameters

- **conn** (*LMIConnection*) – connection object

- **cim_instance_name** (*CIMInstanceName*) – wrapped object

associator_names (*self_wr*, **args*, ***kwargs*)

Returns a list of associated `LMIInstanceName` with this object.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

Returns list of associated `LMIInstanceName` objects

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Associated Instance Names*.

associators (*self_wr*, **args*, ***kwargs*)

Returns a list of associated `LMIInstance` objects with this instance.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

- **IncludeQualifiers** (*bool*) – bool flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – bool flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the array define one or more property names. Each returned object shall not include elements for any properties missing from this list. If *PropertyList* is an empty list, no properties are included in each returned object. If it is None, no additional filtering is defined.

Returns list of associated `LMIInstance` objects

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Associated Instances*.

classname

Returns class name

Return type string

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty string. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

copy()

Returns copy of itself

delete(*self_wr*, **args*, ***kwargs*)

Deletes the instance defined by this object path from the CIMOM.

Returns True, if the instance is deleted; False otherwise

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return True. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Instance Names deletion*.

first_associator(*self_wr*, **args*, ***kwargs*)

Returns the first associated `LMIInstance` with this object.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an

association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.

- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.
- **IncludeQualifiers** (*bool*) – bool flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – bool flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the array define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If it is None, no additional filtering is defined.

Returns first associated `LMIInstance`

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Associated Instances*.

first_associator_name(*self_wr*, **args*, ***kwargs*)

Returns the first associated `LMIInstanceName` with this object.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

Returns first associated `LMIInstanceName` object

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Associated Instance Names*.

first_reference (*self_wr*, **args*, ***kwargs*)
Returns the first association LMIInstance with this object.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter.
- **IncludeQualifiers** (*bool*) – flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as <QUALIFIER> elements in the response.
- **IncludeClassOrigin** (*bool*) – flag indicating, if the CLASSORIGIN attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If PropertyList is None, no additional filtering is defined.

Returns first association LMIInstance object

Raises LMIDeletedObjectError

NOTE: If the method LMIInstanceName.delete() was called, this method will not execute its code and will return None. If the shell uses exceptions, LMIDeletedObjectError will be raised.

Usage: *Association Instances*.

first_reference_name (*self_wr*, **args*, ***kwargs*)
Returns the first association LMIInstanceName with this object.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.

Returns first association LMIInstanceName object

Raises LMIDeletedObjectError

NOTE: If the method LMIInstanceName.delete() was called, this method will not execute its code and will return None. If the shell uses exceptions, LMIDeletedObjectError will be raised.

Usage: *Association Instance Names*.

hostname

Returns host name

Return type string

Raises LMIDeletedObjectError

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty string. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

is_deleted

Returns True, if the instance was deleted from the CIMOM; False otherwise

key_properties(*self_wr*, **args*, ***kwargs*)

Returns list of strings of key properties

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Key properties*.

key_properties_dict(*self_wr*, **args*, ***kwargs*)

Returns dictionary with key properties and corresponding values

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty dictionary. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

key_property_value(*self_wr*, **args*, ***kwargs*)

Parameters **prop_name** (*string*) – key property name

Returns key property value

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

methods(*self_wr*, **args*, ***kwargs*)

Returns a list of `wbem.CIMInstance` methods' names.

Returns list of `wbem.CIMInstance` methods' names

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Instance Methods*.

namespace

Returns namespace name

Return type string

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty string. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

print_key_properties (*self_wr*, **args*, ***kwargs*)
Prints out the list of key properties.

Raises LMIDeletedObjectError

NOTE: If the method LMIInstanceName.delete() was called, this method will not execute its code and will return None. If the shell uses exceptions, LMIDeletedObjectError will be raised.

Usage: *Key properties*.

print_methods (*self_wr*, **args*, ***kwargs*)
Prints out the list of wbem.CIMInstance methods' names.

Raises LMIDeletedObjectError

NOTE: If the method LMIInstanceName.delete() was called, this method will not execute its code and will return None. If the shell uses exceptions, LMIDeletedObjectError will be raised.

Usage: *Instance Methods*.

reference_names (*self_wr*, **args*, ***kwargs*)
Returns a list of association LMIInstanceName objects with this object.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.

Returns list of association LMIInstanceName objects

Raises LMIDeletedObjectError

NOTE: If the method LMIInstanceName.delete() was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, LMIDeletedObjectError will be raised.

Usage: *Association Instance Names*.

references (*self_wr*, **args*, ***kwargs*)
Returns a list of association LMIInstance objects with this object.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter.
- **IncludeQualifiers** (*bool*) – flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as <QUALIFIER> elements in the response.
- **IncludeClassOrigin** (*bool*) – flag indicating, if the CLASSORIGIN attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Each returned object shall not include elements for any properties missing from

this list. If PropertyList is an empty list, no properties are included in each returned object. If PropertyList is None, no additional filtering is defined.

Returns list of association `LMIInstance` objects

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Association Instances*.

to_instance(*self_wr*, **args*, ***kwargs*)
Creates a new `LMIInstance` object from `LMIInstanceName`.

Returns `LMIInstance` object if the object was retrieved successfully; None otherwise.

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Usage: *Conversion to a LMIInstance*.

wrapped_object

Returns wrapped `wbem.CIMInstanceName` object

Raises `LMIDeletedObjectError`

NOTE: If the method `LMIInstanceName.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

LMIInstance

class `lmi.shell.LMIInstance.`**LMIInstance**(*conn*, *lmi_class*, *cim_instance*)
LMI wrapper class representing `wbem.CIMInstance`.

Parameters

- **conn** (*LMIConnection*) – connection object
- **lmi_class** (*LMIClass*) – wrapped creation class of the instance
- **cim_instance** (*CIMInstance*) – wrapped object

associator_names(*self_wr*, **args*, ***kwargs*)
Returns a list of associated `LMIInstanceName` with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).

- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

Returns list of associated `LMIInstanceName` objects

Raises `LMIDeletedObjectError`

Usage: *Associated Instance Names*.

associators(*self_wr*, **args*, ***kwargs*)

Returns a list of associated `LMIInstance` objects with this instance.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.
- **IncludeQualifiers** (*bool*) – bool flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – bool flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the array define one or more property names. Each returned object shall not include elements for any properties missing from this list. If *PropertyList* is an empty list, no properties are included in each returned object. If it is None, no additional filtering is defined.

Returns list of associated `LMIInstance` objects

Raises `LMIDeletedObjectError`

Usage: *Associated Instances*.

classname
Property returning a string of a class name.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty string. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns class name

Return type string

Raises `LMIDeletedObjectError`

copy()

Returns copy of itself

delete(*self_wr*, **args*, ***kwargs*)
Deletes this instance from the CIMOM.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns True, if the instance is deleted; False otherwise

Raises `LMIDeletedObjectError`

Usage: *Instance deletion*.

doc(*self_wr*, **args*, ***kwargs*)
Prints out pretty verbose message with documentation for the instance. If the LMIShell is run in a interactive mode, the output will be redirected to a pager set by environment variable `PAGER`. If there is not `PAGER` set, less or more will be used as a fall-back.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Raises `LMIDeletedObjectError`

first_associator(*self_wr*, **args*, ***kwargs*)
Returns the first associated `LMIInstance` with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the

property in the association class that refers to the returned object shall match the value of this parameter.

- **IncludeQualifiers** (*bool*) – bool flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – bool flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the array define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If it is None, no additional filtering is defined.

Returns first associated `LMIInstance`

Raises `LMIDeletedObjectError`

Usage: *Associated Instances*.

first_associator_name(*self_wr*, **args*, ***kwargs*)

Returns the first associated `LMIInstanceName` with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.

Returns first associated `LMIInstanceName` object

Raises `LMIDeletedObjectError`

Usage: *Associated Instance Names*.

first_reference(*self_wr*, **args*, ***kwargs*)

Returns the first association `LMIInstance` with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter.
- **IncludeQualifiers** (*bool*) – flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.
- **IncludeClassOrigin** (*bool*) – flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If PropertyList is None, no additional filtering is defined.

Returns first association `LMIInstance` object

Raises `LMIDeletedObjectError`

Usage: *Association Instances*.

first_reference_name (*self_wr*, **args*, ***kwargs*)

Returns the first association `LMIInstanceName` with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.

Returns first association `LMIInstanceName` object

Raises `LMIDeletedObjectError`

Usage: *Association Instance Names*.

is_deleted

Returns True, if the instance was deleted from the CIMOM; False otherwise

methods (*self_wr*, **args*, ***kwargs*)

Returns a list of `wbem.CIMInstance` methods' names.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns list of `wbem.CIMInstance` methods' names

Raises `LMIDeletedObjectError`

Usage: *Instance Methods*.

namespace
Property retuning a string of a namespace name.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty string. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns namespace name

Return type string

Raises `LMIDeletedObjectError`

path
Property returning a `LMIInstanceName` object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns `LMIInstanceName` object

Raises `LMIDeletedObjectError`

print_methods(*self_wr*, **args*, ***kwargs*)
Prints out the list of `wbem.CIMInstance` methods' names.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Raises `LMIDeletedObjectError`

Usage: *Instance Methods*.

print_properties(*self_wr*, **args*, ***kwargs*)
Prints out the list of `wbem.CIMInstance` properties.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Raises `LMIDeletedObjectError`

Usage: *Instance Properties*.

properties(*self_wr*, **args*, ***kwargs*)
Returns a list of `wbem.CIMInstance` properties.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns list of `wbem.CIMInstance` properties

Return type list

Raises `LMIDeletedObjectError`

Usage: *Instance Properties*.

properties_dict(*self_wr*, **args*, ***kwargs*)
Returns dictionary containing property name and value pairs. This method may consume significant memory amount when called.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty dictionary. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns dictionary of `wbem.CIMInstance` properties

Raises `LMIDeletedObjectError`

property_value(*self_wr*, **args*, ***kwargs*)
Returns a `wbem.CIMInstance` property value.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters **prop_name** (*string*) – `wbem.CIMInstance` property name

Raises `LMIDeletedObjectError`

push(*self_wr*, **args*, ***kwargs*)
Pushes the modified object to the CIMOM.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return `LMIReturnValue` object containing False as a return value with proper error string set. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns `LMIReturnValue` object with `rval` set to True, if modified; False otherwise

Raises `LMIDeletedObjectError`

Usage: *Instance Properties*.

reference_names(*self_wr*, **args*, ***kwargs*)
Returns a list of association `LMIInstanceName` objects with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.

Returns list of association `LMIInstanceName` objects

Raises `LMIDeletedObjectError`

Usage: *Association Instance Names*.

references(*self_wr*, **args*, ***kwargs*)
Returns a list of association `LMIInstance` objects with this object.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return an empty list. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Parameters

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter.
- **IncludeQualifiers** (*bool*) – flag indicating, if all qualifiers for each object (including qualifiers on the object and on any returned properties) shall be included as `<QUALIFIER>` elements in the response.

- **IncludeClassOrigin** (*bool*) – flag indicating, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in each returned object.
- **PropertyList** (*list*) – if not None, the members of the list define one or more property names. Each returned object shall not include elements for any properties missing from this list. If PropertyList is an empty list, no properties are included in each returned object. If PropertyList is None, no additional filtering is defined.

Returns list of association `LMIInstance` objects

Raises `LMIDeletedObjectError`

Usage: *Association Instances*.

refresh(*self_wr*, **args*, ***kwargs*)
Retrieves a new `wbem.CIMInstance` object. Basically refreshes the object properties. Returns `LMIReturnValue` with rval set to 0, if the wrapped `wbem.CIMInstance` object was refreshed; otherwise rval is set to -1.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return `LMIReturnValue` object containing -1 as a return value with proper error string set. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Returns `LMIReturnValue` object with `rval` set to 0, if refreshed; -1 otherwise

Raises `LMIDeletedObjectError`

Usage: *Instance refreshing*.

tomof(*self_wr*, **args*, ***kwargs*)
Prints out a message with MOF representation of `wbem.CIMMethod`. If the LMIShell is run in a interactive mode, the output will be redirected to a pager set by environment variable `PAGER`. If there is not `PAGER` set, less or more will be used as a fall-back.

NOTE: If the method `LMIInstance.delete()` was called, this method will not execute its code and will return None. If the shell uses exceptions, `LMIDeletedObjectError` will be raised.

Raises `LMIDeletedObjectError`

wrapped_object

Returns wrapped `wbem.CIMInstance` object

LMIJob

lmi.shell.LMIJob.**lmi_is_job_completed**(*job*)
Helper function, which returns bool flag, if the job is in completed state.

Parameters **job** – `LMIInstance` or `wbem.CIMInstance` representing a job

lmi.shell.LMIJob.**lmi_is_job_exception**(*job*)
Helper function, which returns bool flag, if the job is in the exception state.

Parameters **job** – `LMIInstance` or `wbem.CIMInstance` representing a job

lmi.shell.LMIJob.**lmi_is_job_finished**(*job*)
Helper function, which returns bool flag, if the job is in finished state.

Parameters **job** – `LMIInstance` or `wbem.CIMInstance` representing a job

lmi.shell.LMIJob.**lmi_is_job_killed**(*job*)
Helper function, which returns bool flag, if the job is killed.

Parameters **job** – `LMIInstance` or `wbem.CIMInstance` representing a job

`lmi.shell.LMIJob.`**`lmi_is_job_terminated`**(*job*)
Helper function, which returns bool flag, if the job is in terminated state.

Parameters **job** – `LMIInstance` or `wbem.CIMInstance` representing a job

LMIMethod

class `lmi.shell.LMIMethod.`**`LMIMethod`**(*conn*, *lmi_instance*, *method_name*)
LMI wrapper class representing `wbem.CIMMethod`.

Parameters

- **conn** (*LMIConnection*) – connection object
- **lmi_instance** (*LMIInstance(Name)*) – `LMIInstance` or `LMIInstanceName` object, on which the method call will be issued
- **method_name** (*string*) – method name

`doc`()
Prints out pretty verbose message with documentation for the class. If the LMIShell is run in a interactive mode, the output will be redirected to a pager set by environment variable `PAGER`. If there is not `PAGER` set, less or more will be used as a fall-back.

`parameters`()

Returns list of strings of `wbem.CIMMethod`'s parameters

`print_parameters`()
Prints out `wbem.CIMMethod`'s parameters.

`print_valuemap_parameters`()
Prints out the list of strings of constant names.

`return_type`

Returns string of the method call's return type

`tomof`()
Prints out a message with MOF representation of `wbem.CIMMethod`. If the LMIShell is run in a interactive mode, the output will be redirected to a pager set by environment variable `PAGER`. If there is not `PAGER` set, less or more will be used as a fall-back.

`valuemap_parameters`()

Returns list of strings of the constant names

`wrapped_object`

Returns wrapped `wbem.CIMmethod` object

class `lmi.shell.LMIMethod.`**`LMIMethodSignalHelper`**
Helper class which takes care of signal (de)registration and handling.

`callback_attach`(*cb_name*, *cb*)
Registers a callback, which will be called when a *SIGINT* or *SIGTERM* is caught.

Parameters

- **cb_name** (*string*) – callback name
- **cb** – callable object, which takes zero arguments

callback_detach(*cb_name*)
Removes a callback from the callback dictionary.

Parameters **cb_name** (*string*) – callback name

signal_attach()
Registers *SIGINT* and *SIGTERM* signals to local handler in which, the flags for each signal are modified, if such signal is caught.

signal_detach()
Unregisters *SIGINT* and *SIGTERM* handler and removes all the attached callbacks.

signal_handled()

Returns True, if any of *SIGINT* or *SIGTERM* has been caught; False otherwise

signal_handler(*signo*, *frame*)
Signal handler, which is called, when *SIGINT* and *SIGTERM* are sent to the LMIShell.

Parameters

- **signo** (*int*) – signal number
- **frame** – – stack frame

class lmi.shell.LMIMethod.**LMISignalHelperBase**
Base signal handling class.

static signal(*signo*, *handler*)
Calls signal() for signo, handler and returns the old signal handler. If signo is list of signals, the signal() call is applied for each signo. If handler is also list, each signal from signo will be handled by corresponding handler. In such case, tuple of previous handlers will be returned.

static signal_core(*signo*, *handler*)
Wrapper method for signal.signal(). In case of ValueError, it returns None, old signal handler otherwise. If handler is None, default signal handler is set for such signal.

LMINamespace

class lmi.shell.LMINamespace.**LMINamespace**(*conn*, *name*)
LMI class representing CIM namespace.

Parameters

- **conn** (*LMIConnection*) – connection object
- **name** (*string*) – namespace name

classes()
Returns a list of class names.

Parameters

- **filter_key** (*string*) – substring of a class name
- **exact_match** (*bool*) – tells, if to search for exact match or substring

Returns list of class names

Usage: *Available classes*.

cql(*query*)
Executes a CQL query and returns a list of LMIInstance objects.

Parameters **query** (*string*) – CQL query to execute

Returns `LMIReturnValue` object with `rval` set to a list of `LMIInstance` objects

Usage: *Queries*.

get_class(*classname*)
Returns `LMIClass`.

Parameters **classname** (*string*) – class name of new `LMIClass`

Raises `LMIClassNotFound`

name

Returns namespace name

Return type string

print_classes()
Prints out a list of classes.

Parameters

- **filter_key** (*string*) – substring of a class name
- **exact_match** (*bool*) – tells, if to search for exact match, or to search for a matching substring

Usage: *Available classes*.

wql(*query*)
Executes a WQL query and returns a list of `LMIInstance` objects.

Parameters **query** (*string*) – WQL query to execute

Returns `LMIReturnValue` object with `rval` set to a list of `LMIInstance` objects

Usage: *Queries*.

class `lmi.shell.LMINamespace.`**LMINamespaceRoot**(*conn*)
Derived class for *root* namespace. Object of this class is accessible from `LMIConnection` object as a hierarchy entry.

Parameters **conn** (*LMIConnection*) – connection object

namespaces

Returns list of strings with available namespaces

Usage: *Available namespaces*.

print_namespaces()
Prints out all available namespaces accessible via the namespace *root*.

Usage: *Available namespaces*.

LMIObjectFactory

class `lmi.shell.LMIObjectFactory.`**LMIObjectFactory**
Object factory class. Used to avoid circular import dependencies between several LMI classes. The class implements a singleton design pattern.

Example of usage:

```
LMIObjectFactory().register(SomeClass)
some_obj = LMIObjectFactory().SomeClass(*args, **kwargs)
```

register(*reg_class*)
Registers a class into the factory.

LMIReturnValue

class lmi.shell.LMIReturnValue.**LMIReturnValue**
Class representing a return value, which holds 3 main types of attributes:

Parameters

- **rval** – return value
- **rparams** (*dictionary*) – returned parameters of e.g. method call
- **errorstr** (*string*) – error string

LMIShellCache

class lmi.shell.LMIShellCache.**LMIClassCacheEntry**(*cim_class*, *full_fetch*)
Class used for storing wbem.CIMClass in LMIShellCache.

Parameters

- **cim_class** (*CIMClass*) – wbem.CIMClass to cache
- **full_fetch** (*bool*) – True, if class is cached with qualifiers

class lmi.shell.LMIShellCache.**LMIShellCache**(*active=True*, *classname_dict=None*, *class_dict=None*, *class_superclass_dict=None*)
Class representing a LMIShell cache.

Parameters

- **active** (*bool*) – specifies, if the cache is active
- **classname_list** (*list*) – list of strings of cached class names
- **class_dict** (*dictionary*) – cached wbem.CIMClass objects, where the key is the class name and value is CIMClass object
- **class_superclass_dict** (*dictionary*) – dictionary, where the key is namespace and value is dictionary of classname:superclass

active

Returns True, if the cache is active; False otherwise

add_class(*cim_class*, *namespace='root/cimv2'*, *full_fetch=False*)
Stores a new wbem.CIMClass object into the cache.

Parameters

- **cim_class** (*CIMClass*) – wbem.CIMClass object
- **namespace** (*string*) – namespace storing cached classes

add_superclass(*classname*, *superclass*, *namespace*)
Stores a new pair classname : superclassname into the cache.

Parameters

- **classname** (*string*) – class name to be stored
- **superclass** (*string*) – super class name to be stored
- **namespace** (*string*) – namespace name of the classname

clear()
Clears the cache.

get_class(*classname*, *namespace='root/cimv2'*)

Parameters

- **classname** (*string*) – cached class name
- **namespace** (*string*) – namespace storing cached classes

Returns cache object, if proper class name provided, None otherwise

Return type LMIClassCacheEntry

get_classes(*namespace='root/cimv2'*)

Parameters **namespace** (*string*) – namespace storing cached classes

Returns list of cached class names or None, if no cached classes is stored

get_superclass(*classname*, *namespace*)

Parameters

- **classname** (*string*) – cached class name
- **namespace** (*string*) – namespace name

Returns cached superclass to the given class name

Return type string

has_superclass(*classname*, *namespace*)

Parameters

- **classname** (*string*) – cached class name
- **namespace** (*string*) – namespace name

Returns True, if the cache contains superclass to the given class name; False otherwise

set_classes(*classname_list*, *namespace='root/cimv2'*)
Stores a new class names' list.

Parameters **namespace** (*string*) – namespace storing cached classes

LMIShellClient

class lmi.shell.LMIShellClient.**LMIShellClient**(*uri*, *username=''*, *password=''*, *interactive=False*, *use_cache=True*, *key_file=None*, *cert_file=None*, *verify_server_cert=True*)
LMIShellClient overrides few methods due to caching purposes.

Parameters

- **uri** (*string*) – URI of the CIMOM
- **username** (*string*) – account, under which, the CIM calls will be performed
- **password** (*string*) – user's password

- **interactive** (*bool*) – flag indicating, if the LMIShell client is running in the interactive mode; default value is False.
- **use_cache** (*bool*) – flag indicating, if the LMIShell client should use cache for `CIMClass` objects. This saves a lot's of communication, if there is often the `LMIShellClient.get_class_names()` or `LMIShellClient.attr.get_class()` call issued.
- **key_file** (*string*) – path to x509 key file; default value is None
- **cert_file** (*string*) – path to x509 cert file; default value is None
- **verify_server_cert** (*bool*) – indicates, whether a server side certificate needs to be verified, if SSL used; default value is True

NOTE: If interactive is set to True, LMIShell will:

•prompt for username and password, if missing and connection via Unix socket can not be established.

•use pager for the output of: `LMIInstance.doc()`, `LMIClass.doc()`, `LMIInstance.tomof()` and `LMIMethod.tomof()`

cache

Returns LMIShell's cache

Return type `LMIShellCache`

get_class(*classname*, *namespace=None*, *LocalOnly=True*, *IncludeQualifiers=True*, *IncludeClassOrigin=False*, *PropertyList=None*, *full_fetch=False*)

Returns a `wbem.CIMClass` object.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – – namespace name, from which the `wbem.CIMClass` should be retrieved; if None, default namespace will be used (**NOTE:** see `wbem`)
- **LocalOnly** (*bool*) – indicates, if only local members should be present in the returned `wbem.CIMClass`; any CIM elements (properties, methods, and qualifiers), except those added or overridden in the class as specified in the classname input parameter, shall not be included in the returned class.
- **IncludeQualifiers** (*bool*) – indicates, if qualifiers for the class (including qualifiers on the class and on any returned properties, methods, or method parameters) shall be included in the response.
- **IncludeClassOrigin** (*bool*) – indicates, if the `CLASSORIGIN` attribute shall be present on all appropriate elements in the returned class.
- **PropertyList** (*list*) – if present and not None, the members of the list define one or more property names. The returned class shall not include elements for properties missing from this list. Note that if LocalOnly is specified as True, it acts as an additional filter on the set of properties returned. For example, if property A is included in the PropertyList but LocalOnly is set to True and A is not local to the requested class, it is not included in the response. If the PropertyList input parameter is an empty list, no properties are included in the response. If the PropertyList input parameter is None, no additional filtering is defined.

Returns `LMIReturnValue` object with rval set to `wbem.CIMClass`, if no error occurs; otherwise rval is set to none and errorstr to appropriate error string

Raises `CIMError`, `ConnectionError`

get_class_names(*namespace=None*, *ClassName=None*, *DeepInheritance=False*)
Returns a list of class names.

Parameters

- **namespace** (*string*) – namespace, from which the class names list should be retrieved; if None, default namespace will be used (**NOTE:** see `wbem`)
- **ClassName** (*string*) – defines the class that is the basis for the enumeration. If the ClassName input parameter is absent, this implies that the names of all classes.
- **DeepInheritance** (*bool*) – if not present, of False, only the names of immediate child subclasses are returned, otherwise the names of all subclasses of the specified class should be returned.

Returns `LMIReturnValue` object with `rval` set to a list of strings containing class names, if no error occurs; otherwise `rval` is set to None and `errorstr` contains an appropriate error string

Raises `CIMError`, `ConnectionError`

get_superclass(*classname*, *namespace=None*)
Returns string of a superclass to given class, if such superclass exists, None otherwise.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace name

Returns superclass name to a given classname or None

Raises `CIMError`, `ConnectionError`

interactive

Returns flag, if the LMIShell is run in the interactive mode

Return type bool

use_cache

Returns flag, which tells, if the LMIShell should use a cache

Return type bool

LMIShellConfig

class `lmi.shell.LMIShellConfig.`**LMIShellConfig**
Class representing the shell's configuration. The class is responsible for loading the configuration from the file and provides a unified API to access these settings.

Constructs a `LMIShellConfig` object and loads the configuration from the file. If there is no such file, the shell's configuration properties are set to default values. Configuration file uses python syntax. If there is a syntax error in the configuration file, the properties are set to default values, as well.

cert_file
Property returning a file name containing x509 certificate. This is used for `LMIIndicationListener`.

Returns x509 certificate file name

Return type string

history_file
Property returning a string containing the shell's history file.

Returns history file

Return type string

history_length
Property returning a number with the shell's history file length.

Returns history file length

Return type int

key_file
Property returning a file name containing x509 certificate private key. This is used for `LMIIndicationListener`.

Returns x509 certificate private key

Return type string

use_cache
Property returning a bool flag, if the shell should use cache for class retrieval.

Returns flag, if the shell should use a cache

Return type bool

use_exceptions
Property returning a bool flag, if the shell should throw the exceptions away, or if they should be propagated further.

Returns flag, if the shell should use exceptions, or throw them away

Return type bool

LMIShellLogger

class `lmi.shell.LMIShellLogger.`**LMIShellLogger**(*name*, *level=0*)
LMIShell's logger with queueing capability.

critical(*msg*, **args*, ***kwargs*)
Log a message with severity 'CRITICAL'.

debug(*msg*, **args*, ***kwargs*)
Log a message with severity 'DEBUG'.

error(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR'.

exception(*msg*, **args*, ***kwargs*)
Log a message with severity 'ERROR' also with exception information.

info(*msg*, **args*, ***kwargs*)
Log a message with severity 'INFO'.

processQueue()
Logs all enabled log records stored in internal queue.

setLevel(*level*)
Sets a logging level of this handler. If there are any log records stored in internal queue, they are also handled.

Parameters **level** (*int*) – logging level

warning(*msg*, **args*, ***kwargs*)
Log a message with severity 'WARNING'.

lmi.shell.LMIShellLogger.**lmi_get_logger**()
Returns LMIShell's logger.

Returns logger

lmi.shell.LMIShellLogger.**lmi_init_logger**()
Initializes LMIShell's logging.

lmi.shell.LMIShellLogger.**lmi_setup_logger**(*log_options*)
Sets logging level.

Parameters **log_options** (*int*) – level defined in `LMIShellOptions`

LMIShellOptions

class lmi.shell.LMIShellOptions.**LMIShellOptionParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)
Helper class for CLI option parsing.

error(*msg*)
Prints help message, error message and exits with erro code 2.

class lmi.shell.LMIShellOptions.**LMIShellOptions**(*argv*)
Class representing a LMIShell command line options. In the constructor, all command line options before a script name are passed to the LMIShell. First position argument belongs to the script and the rest of command line options is passed to the script run under the LMIShell.

Parameters **argv** (*list*) – CLI arguments

cwd_first_in_path

Returns True, if CWD should be prepended in sys.path; False if appended

interact

Returns flag, which indicates, if the LMIShell should enter an interactive mode, after executing a provided script. The behavior is similar to python interpreter

Return type bool

interactive

Returns flag, which tells if the LMIShell should be initially run in the interactive mode

Return type bool

log

Returns log level

Return type int

Log level can be one of the following:

- `_LOG_DEFAULT`
- `_LOG_VERBOSE`
- `_LOG_MORE_VERBOSE`
- `_LOG_QUIET`

script_argv

Returns list of command line arguments of the interpreted script

script_name

Returns script name, which is about to be run under the LMIShell

Return type string

verify_server_cert

Returns flag, which indicates, if LMIShell should verify server side certificate, if SSL used

Return type bool

class `lmi.shell.LMIShellOptions.`**LMIShellOptionsHelpWithVersionFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Helper class used for help message formatting.

LMIShellVersion

LMISubscription

class `lmi.shell.LMISubscription.`**LMISubscription**(*client*, *cim_filter*, *cim_handler*, *cim_subscription*, *permanent*)

Class holding information about a indication subscription.

Parameters

- **client** (*LMIShellClient*) – client object used for CIMOM communication
- **cim_filter** (*tuple*) – contains filter object and bool indicator, if the filter object was created temporarily
- **cim_handler** (*tuple*) – contains handler object and bool indicator, if the handler object was created temporarily
- **cim_subscription** – subscription object
- **permanent** (*bool*) – indicates, if the indication should be deleted on the LMIShell's quit

delete()

Cleans up the indication subscription.

First it deletes subscription object. If `LMISubscription._cim_filter_tpl` contains a flag, that the filter object was created temporarily, it will be deleted by this call. If `LMISubscription._cim_handler_tlp` contains an flag, that the handler object was created temporarily, it will be deleted as well.

This is called from `LMIConnection` object, which holds an internal list of all subscribed indications by the LMIShell (if not created by hand).

LMIUtil

class `lmi.shell.LMIUtil.`**`LMIPassByRef`**(*val*)
Helper class used for passing a value by reference. It uses the advantage of python, where all the dictionaries are passed by reference.

Parameters **val** – value, which will be passed by reference

Example of usage:

```
by_ref = LMIPassByRef(some_value)
by_ref.value == some_value
```

`value`

Returns value passed by reference.

class `lmi.shell.LMIUtil.`**`LMIUseExceptionsHelper`**
Singleton helper class used for storing a bool flag, which defines, if the LMIShell should propagate exceptions or dump them.

`use_exceptions`

Returns whether the LMIShell should propagate the exceptions, or throw them away

Return type bool

`lmi.shell.LMIUtil.`**`lmi_associators`**(*assoc_classes*)
Helper function to speed up associator traversal. Returns a list of tuples, where each tuple contains `LMIInstance` objects, which are in association.

Parameters **assoc_classes** (*list*) – list of `LMIClass` objects, for which the associations will be returned

Returns list of tuples of `LMIInstance` objects in association

`lmi.shell.LMIUtil.`**`lmi_cast_to_cim`**(*t*, *value*)
Casts the value to CIM type.

Parameters

- **t** (*string*) – string of CIM type
- **value** – variable to cast

Returns cast value in `wbem` type

`lmi.shell.LMIUtil.`**`lmi_cast_to_lmi`**(*t*, *value*)
Casts the value to LMI (python) type.

Parameters

- **t** (*string*) – string of CIM type
- **value** – variable to cast

Returns cast value in `python` native type

`lmi.shell.LMIUtil.`**`lmi_get_use_exceptions`**()

Returns whether the LMIShell should use the exceptions, or throw them away

Return type bool

lmi.shell.LMIUtil.**lmi_instance_to_path**(*instance*)
Helper function, which returns wbem.CIMInstanceName extracted out of input instance.

Parameters **instance** – object, which can be instance of following classes:

- wbem.CIMInstance
- wbem.CIMInstanceName
- LMIInstance
- LMIInstanceName

Returns extracteed wbem.CIMInstanceName object

Raises TypeError

lmi.shell.LMIUtil.**lmi_is_localhost**(*uri*)
Helper function, which returns True, if URI points to localhost.

Parameters **uri** (*str*) – URI to check

lmi.shell.LMIUtil.**lmi_isinstance**(*lmi_obj*, *lmi_class*)
Function returns True if lmi_obj is an instance of a lmi_class, False otherwise. When passed LMIInstance, LMIInstanceName as lmi_obj and lmi_class is of LMIClass type, function can tell, if such lmi_obj is direct instance of LMIClass, or it's super class.

If lmi_obj and lmi_class is not instance of mentioned classes, an exception will be raised.

Parameters

- **lmi_obj** – instance of LMIInstance or LMIInstanceName which is checked, if such instance is instance of the lmi_class
- **lmi_class** (*LMIClass*) – instance of LMIClass object

Returns whether **lmi_obj** is instance of **lmi_class**

Return type bool

Raises TypeError

lmi.shell.LMIUtil.**lmi_parse_uri**(*uri*)
Parses URI into scheme, hostname, port, username and password.

lmi.shell.LMIUtil.**lmi_raise_or_dump_exception**(*e=None*)
Function which either raises an exception, or throws it away.

Parameters **e** (*Exception*) – exception, which will be either raised or thrown away

lmi.shell.LMIUtil.**lmi_set_use_exceptions**(*use=True*)
Sets a global flag indicating, if the LMIShell should use the exceptions, or throw them away.

Parameters **use** (*bool*) – specifies, whether the LMIShell should use the exceptions

lmi.shell.LMIUtil.**lmi_transform_to_cim_param**(*t*, *value*)
Helper function for method calls, which transforms input object into wbem.CIMInstanceName object. Members if lists, dictionaries and tuples are transformed as well. The function does not cast numeric types.

Parameters

- **t** (*string*) – string of CIM type
- **value** – object to be transformed to wbem type.

Returns transformed LMIShell's object into wbem one

lmi.shell.LMIUtil.**lmi_transform_to_lmi**(*conn*, *value*)
Transforms returned values from a method call into LMI wrapped objects. Returns transformed input, where wbem.CIMInstance and wbem.CIMInstanceName are wrapped into LMI wrapper classes and primitive types are cast to python native types.

Parameters

- **conn** (*LMIConnection*) – connection object
- **value** – object to be transformed into python type from wbem one

Returns transformed py:wbem object into LMIShell one

lmi.shell.LMIUtil.**lmi_wrap_cim_class**(*conn*, *cim_class_name*, *cim_namespace_name*)
Helper function, which returns wrapped wbem.CIMClass into LMIClass.

Parameters

- **conn** (*LMIConnection*) – connection object
- **cim_class_name** (*string*) – string containing wbem.CIMClass name
- **cim_namespace_name** (*string*) – string containing wbem.CIMNamespace name, or None, if the namespace is not known

Returns wrapped wbem.CIMClass into LMIClass

lmi.shell.LMIUtil.**lmi_wrap_cim_instance**(*conn*, *cim_instance*, *cim_class_name*, *cim_namespace_name*)
Helper function, which returns wrapped wbem.CIMInstance into LMIInstance.

Parameters

- **conn** (*LMIConnection*) – connection object
- **cim_instance** (*CIMInstance*) – wbem.CIMInstance object to be wrapped
- **cim_class_name** (*string*) – wbem.CIMClass name
- **cim_namespace_name** (*string*) – wbem.CIMNamespace name, or None, if the namespace is not known

Returns wrapped wbem.CIMInstance into LMIInstance

lmi.shell.LMIUtil.**lmi_wrap_cim_instance_name**(*conn*, *cim_instance_name*)
Helper function, which returns wrapped wbem.CIMInstanceName into LMIInstanceName.

Parameters

- **conn** (*LMIConnection*) – connection object
- **cim_instance_name** (*CIMInstanceName*) – wbem.CIMInstanceName object to be wrapped

Returns wrapped wbem.CIMInstanceName into LMIInstanceName

lmi.shell.LMIUtil.**lmi_wrap_cim_method**(*conn*, *cim_method_name*, *lmi_instance*)
Helper function, which returns wrapped wbem.CIMMethod into LMIMethod.

Parameters

- **conn** (*LMIConnection*) – connection object
- **cim_method_name** (*string*) – method name
- **lmi_instance** (*LMIInstance*) – object, on which the method call will be issued

Returns wrapped wbem.CIMMethod into LMIMethod

lmi.shell.LMIUtil.**lmi_wrap_cim_namespace**(*conn*, *cim_namespace_name*)
Helper function, which returns wrapped CIM namespace in LMINamespace.

Parameters

- **conn** (*LMIConnection*) – connection object
- **cim_namespace_name** (*string*) – CIM namespace name

Returns wrapped CIM namespace into LMINamespace

LMIWSMANClient

class lmi.shell.LMIWSMANClient.**LMIWSMANClient**(*uri*, *username=''*, *password=''*, *interactive=False*, *verify_server_cert=True*, *key_file=None*, *cert_file=None*)
WS-MAN client.

Parameters

- **uri** (*string*) – URI of the CIMOM
- **username** (*string*) – account, under which, the CIM calls will be performed
- **password** (*string*) – user's password
- **verify_server_cert** (*bool*) – indicates, whether a server side certificate needs to be verified, if SSL used; default value is True
- **key_file** (*string*) – path to x509 key file; default value is None
- **cert_file** (*string*) – path to x509 cert file; default value is None

association(*instance*, *relationship*, *result_cls*, *AssocClass=None*, *ResultClass=None*, *Role=None*, *ResultRole=None*, *limit=-1*)
Enumerates association instance (names).

Parameters

- **instance** – object, for which the association objects will be enumerated. The object needs to be instance of following classes:
  - wbem.CIMInstance
  - wbem.CIMInstanceName
  - LMIInstance
  - LMIInstanceName
- **relationship** – LMIWSMANClient.ASSOC_ASSOCIATORS or LMIWSMANClient.ASSOC_REFERENCES
- **result_cls** – LMIWSMANClient.RES_INSTANCE or LMIWSMANClient.RES_INSTANCE_NAME
- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).

- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.
- **limit** (*int*) – enumeration limit

Returns list of association objects

call_method(**args*, ***kwargs*)
Executes a method within a given instance.

Parameters

- **instance** – object, on which the method will be executed. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **method** (*string*) – string containing a method name
- **params** (*dictionary*) – parameters passed to the method call

Returns `LMIReturnValue` object with rval set to return value of the method call, rparams set to returned parameters from the method call, if no error occurs; otherwise rval is set to -1 and errorstr to appropriate error string

Raises `CIMError`

connect()
Compatibility method present due to `LMICIMXMLClient`.

create_instance(**args*, ***kwargs*)
Not supported.

delete_instance(**args*, ***kwargs*)
Not supported.

disconnect()
Compatibility method present due to `LMICIMXMLClient`.

dummy()
Sends a "dummy" request to verify credentials.

Returns `LMIReturnValue` with rval set to True, if provided credentials are OK; False otherwise. If LMIShell uses exceptions, `CIMError` will be raised.

Raises `CIMError`

enumerate(*result_cls*, *classname*, *namespace=None*, *inst_filter=None*, *limit=-1*, ***kwargs*)
Enumerates instance (names).

Parameters

- **result_cls** (*int*) – either `LMIWSMANClient.RES_INSTANCE` or `LMIWSMANClient.RES_INSTANCE_NAME`
- **classname** (*str*) – class name to enumerate
- **namespace** (*str*) – namespace where the class is located
- **inst_filter** (*dict*) – dictionary containing keys and values for the filter
- **limit** (*int*) – enumeration limit
- **kwargs** – keyword arguments used for inst_filter

Return type list containing `wbem.CIMInstance` of `wbem.CIMInstanceName`

Raises `CIMError`

enumerate_iter(*classname*, *namespace*, *filt*, *options=None*, *limit=-1*)
Enumerates instance (names).

Parameters

- **filt** (*pywsman.Filter*) – filter for enumeration
- **options** (*pywsman.ClientOptions*) – options for enumeration
- **limit** (*int*) – enumeration limit

Return type list containing `wbem.CIMInstance` of `wbem.CIMInstanceName`

Raises `CIMError`

enumerate_iter_with_uri(*uri*, *filt*, *options=None*, *limit=-1*)
Enumerates instance (names).

Parameters

- **uri** (*str*) – URI of the resource
- **filt** (*pywsman.Filter*) – filter for enumeration
- **options** (*pywsman.ClientOptions*) – options for enumeration
- **limit** (*int*) – enumeration limit

Return type list containing `wbem.CIMInstance` of `wbem.CIMInstanceName`

Raises `CIMError`

exec_query(**args*, ***kwargs*)
Executes a query and returns a list of `wbem.CIMInstance` objects.

Parameters

- **query_lang** (*string*) – query language
- **query** (*string*) – query to execute
- **namespace** (*string*) – target namespace for the query

Returns `LMIReturnValue` object with `rval` set to list of `wbem.CIMInstance` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` is set to corresponding error string

Raises `CIMError`

get_associator_names(**args*, ***kwargs*)
Returns a list of associated `wbem.CIMInstanceName` objects with an input instance.

Parameters

- **instance** – for this object the list of associated `wbem.CIMInstanceName` will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an instance of this class or one of its subclasses.
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of names by mandating that each returned name identify an object that shall be associated to the source object through an association in which the named returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter.
- **limit** (*int*) – enumeration limit

Returns list of associated `wbem.CIMInstanceName` objects with an input instance, if no error occurs; otherwise en empty list is returned

Raises `CIMError`

get_associators(**args*, ***kwargs*)
Returns a list of associated `wbem.CIMInstance` objects with an input instance.

Parameters

- **instance** – for this object the list of associated `wbem.CIMInstance` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **AssocClass** (*string*) – valid CIM association class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an instance of this class or one of its subclasses. Default value is None.

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be either an instance of this class (or one of its subclasses) or be this class (or one of its subclasses). Default value is None.
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated with the source object through an association in which the source object plays the specified role. That is, the name of the property in the association class that refers to the source object shall match the value of this parameter. Default value is None.
- **ResultRole** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall be associated to the source object through an association in which the returned object plays the specified role. That is, the name of the property in the association class that refers to the returned object shall match the value of this parameter. Default value is None.
- **IncludeQualifiers** – unused
- **IncludeClassOrigin** – unused
- **PropertyList** – unused
- **limit** (*int*) – enumeration limit

Returns list of associated `wbem.CIMInstance` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`

get_class(**args*, ***kwargs*)
Not supported.

get_class_names(**args*, ***kwargs*)
Not supported.

get_instance(**args*, ***kwargs*)
Returns a `wbem.CIMInstance` object.

Parameters

- **instance** – path of the object, which is about to be retrieved. The object needs to be instance of following classes:
  - `wbem.CIMInstanceName`
  - `wbem.CIMInstance`
  - `LMIInstanceName`
  - `LMIInstance`
- **LocalOnly** – unused
- **IncludeQualifiers** – unused
- **IncludeClassOrigin** – unused
- **PropertyList** – unused

Returns `LMIReturnValue` object, where `rval` is set to `wbem.CIMInstance` object, if no error occurs; otherwise `errorstr` is set to corresponding error string

Raises `CIMError`

get_instance_names(**args*, ***kwargs*)
Returns a list of `wbem.CIMInstanceName` objects.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace name, where the instance names live
- **inst_filter** (*dict*) – dictionary containing filter values. The key corresponds to the primary key of the `wbem.CIMInstanceName`; value contains the filtering value.
- **limit** (*int*) – enumeration limit
- **kwargs** (*dictionary*) – supported keyword arguments (these are **deprecated**)
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** (*string*) – filtering value, see above

Returns `LMIReturnValue` object with `rval` contains a list of `wbem.CIMInstanceName` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` contains appropriate error string

Raises `CIMError`

get_instances(**args*, ***kwargs*)

Returns a list of `wbem.CIMInstance` objects.

Parameters

- **classname** (*string*) – class name
- **namespace** (*string*) – namespace, where the instances live
- **inst_filter** (*dictionary*) – dictionary containing filter values. The key corresponds to the primary key of the `wbem.CIMInstanceName`; value contains the filtering value.
- **client_filtering** (*bool*) – if True, client-side filtering will be performed, otherwise the filtering will be done by a CIMOM. Default value is False.
- **limit** (*int*) – enumeration limit
- **kwargs** (*dictionary*) – supported keyword arguments (these are **deprecated**)
  - **Key** or **key** (*string*) – filtering key, see above
  - **Value** or **value** (*string*) – filtering value, see above

Returns `LMIReturnValue` object with `rval` set to a list of `wbem.CIMIntance` objects, if no error occurs; otherwise `rval` is set to None and `errorstr` is set to corresponding error string.

Raises `CIMError`

get_reference_names(**args*, ***kwargs*)

Returns a list of association `wbem.CIMInstanceName` objects with an input instance.

Parameters

- **instance** – for this object the association `wbem.CIMInstanceName` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`

- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of object names by mandating that each returned Object Name identify an instance of this class (or one of its subclasses) or this class (or one of its subclasses).
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of object names by mandating that each returned object name shall identify an object that refers to the target instance through a property with a name that matches the value of this parameter.
- **limit** (*int*) – enumeration limit

Returns list of association `wbem.CIMInstanceName` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`

get_references(**args*, ***kwargs*)
Returns a list of association `wbem.CIMInstance` objects with an input instance.

Parameters

- **instance** – for this object the association `wbem.CIMInstances` objects will be returned. The object needs to be instance of following classes:
  - `wbem.CIMInstance`
  - `wbem.CIMInstanceName`
  - `LMIInstance`
  - `LMIInstanceName`
- **ResultClass** (*string*) – valid CIM class name. It acts as a filter on the returned set of objects by mandating that each returned object shall be an instance of this class (or one of its subclasses) or this class (or one of its subclasses). Default value is None.
- **Role** (*string*) – valid property name. It acts as a filter on the returned set of objects by mandating that each returned object shall refer to the target object through a property with a name that matches the value of this parameter. Default value is None.
- **IncludeQualifiers** – unused
- **IncludeClassOrigin** – unused
- **PropertyList** – unused
- **limit** (*int*) – enumeration limit

Returns list of association `wbem.CIMInstance` objects with an input instance, if no error occurs; otherwise an empty list is returned

Raises `CIMError`

get_superclass(**args*, ***kwargs*)
Not supported.

hostname

Returns hostname of CIMOM

Return type string

modify_instance(**args*, ***kwargs*)
Not supported.

uri

Returns URI of the CIMOM

Return type string

username

Returns user name as a part of provided credentials

Return type string

LMI Scripts API reference

This is a generated documentation from *OpenLMI Scripts* sources. This covers everything under `lmi.scripts` namespace.

`lmi.scripts.common` package provides useful functionality for script development. Various scripts share this directory in order to provide command-line interface through *LMI metacommand*.

Generated from version: `0.10.1`

Scripts version: 0.3.0

LMI Scripts common library reference

This library builds on top of *LMIShell*'s functionality. It provides various utilities and wrappers for building command-line interfaces to *OpenLMI Providers*.

Generated from version: `0.10.1`

Exported members: Package with client-side python modules and command line utilities.

`lmi.scripts.common.`**`get_computer_system`**`(`*`ns`*`)`
Obtain an instance of `CIM_ComputerSystem` or its subclass. Preferred class name can be configured in configuration file. If such class does not exist, a base class (`CIM_ComputerSystem`) is enumerated instead. First feasible instance is cached and returned.

Parameters **ns** (`lmi.shell.LMINamespace`) – Namespace object where to look for computer system class.

Returns Instance of `CIM_ComputerSystem`.

Return type `lmi.shell.LMIInstance`.

Submodules:

command This subpackage defines base classes and utility functions for declaring commands. These serve as wrappers for functions in libraries specific to particular provider.

Tree of these commands build a command line interface for this library.

command.base Module defining base command class for all possible commands of `lmi` meta-command.

`lmi.scripts.common.command.base.`**`DEFAULT_FORMATTER_OPTIONS = {'padding': 0, 'human_friendly': False, 'no_`**
Default formatting options overriden by options passed onc ommand-line and set in configuration file.

class lmi.scripts.common.command.base.**LmiBaseCommand**(*app*, *cmd_name*, *parent=None*)

Abstract base class for all commands handling command line arguments. Instances of this class are organized in a tree with root element being the `lmi` meta-command (if not running in interactive mode). Each such instance can have more child commands if its `LmiBaseCommand.is_multiplexer()` method return `True`. Each has one parent command except for the top level one, whose `parent` property returns `None`.

Set of commands is organized in a tree, where each command (except for the root) has its own parent. `is_end_point()` method distinguishes leaves from nodes. The path from root command to the leaf is a sequence of commands passed to command line.

There is also a special command called selector. Its `is_selector()` method returns `True`. It selects proper command that shall be passed all the arguments based on expression with profile requirements. It shares its name and parent with selected child.

If the `LmiBaseCommand.has_own_usage()` returns `True`, the parent command won't process the whole command line and the remainder will be passed as a second argument to the `LmiBaseCommand.run()` method.

Parameters

- **app** – Main application object.
- **cmd_name** (*string*) – Name of command.
- **parent** (`LmiBaseCommand`) – Parent command.

app

Return application object.

classmethod child_commands()

Abstract class method returning dictionary of child commands with structure:

```
{ "command-name" : cmd_factory, ... }
```

Dictionary contains just a direct children (commands, which may immediately follow this particular command on command line).

cmd_name

Name of this subcommand as a single word.

cmd_name_parts

Convenience property calling `get_cmd_name_parts()` to obtain command path as a list of all preceding command names.

Return type list

format_options

Compose formatting options. Parent commands are queried for defaults. If command has no parent, default options will be taken from `DEFAULT_FORMATTER_OPTIONS` which are overriden by config settings.

Returns Arguments passed to formatter factory when formatter is for current command is constructed.

Return type dictionary

get_cmd_name_parts(*all_parts=False*, *demand_own_usage=True*, *for_docopt=False*)

Get name of this command as a list composed of names of all preceding commands since the top level one. When in interactive mode, only commands following the active one will be present.

Parameters

- **full** (*boolean*) – Take no heed to the active command or interactive mode. Return all command names since top level node inclusive. This is overriden with *for_docopt* flag.

- **demand_own_usage** (*boolean*) – Wether to continue the upward traversal through command hiearchy past the active command until the command with its own usage is found. This is the default behaviour.
- **for_docopt** (*boolean*) – Docopt parser needs to be given arguments list without the first item compared to command names in usage string it receives. Thus this option causes skipping the first item that would be otherwise included.

Returns Command path. Returned list will always contain at least the name of this command.

Return type list

classmethod get_description()
Return description for this command. This is usually a first line of documentation string of a class.

Return type string

get_usage(*proper=False*)
Get command usage. Return value of this function is used by docopt parser as usage string. Command tree is traversed upwards until command with defined usage string is found. End point commands (leaves) require manually written usage, so the first command in the sequence of parents with own usage string is obtained and its usage returned. For nodes missing own usage string this can be generated based on its subcommands.

Parameters **proper** (*boolean*) – Says, whether the usage string written manually is required or not. It applies only to node (not a leaf) commands without its own usage string.

classmethod has_own_usage()

Returns `True`, if this command has its own usage string, which is returned by `LmiBaseCommand.get_description()`. Otherwise the parent command must be queried.

Return type boolean

classmethod is_end_point()

Returns `True`, if this command parses the rest of command line and can not have any child subcommands.

Return type boolean

classmethod is_multiplexer()
Is this command a multiplexer? Note that only one of `is_end_point()`, `is_selector()` and this method can evaluate to‘‘True‘‘.

Returns `True` if this command is not an end-point command and it's a multiplexer. It contains one or more subcommands. It consumes the first argument from command-line arguments and passes the rest to one of its subcommands.

Return type boolean

classmethod is_selector()
Is this command a selector?

Returns `True` if this command is a subclass of `lmi.scripts.common.command.select.LmiSelectComma`

Return type boolean

parent
Return parent command.

run(*args*)
Handle the command line arguments. If this is not an end point command, it will pass the unhandled

arguments to one of it's child commands. So the arguments are processed recursively by the instances of this class.

Parameters **args** (*list*) – Arguments passed to the command line that were not yet parsed. It's the contents of `sys.argv` (if in non-interactive mode) from the current command on.

Returns Exit code of application. This maybe also be a boolean value or `None`. `None` and `True` are treated as a success causing exit code to be 0.

Return type integer

session

Returns Session object. Session for command and all of its children may be overriden with a call to `set_session_proxy()`.

Return type `lmi.scripts.common.session.Session`

set_session_proxy(*session*)

Allows to override session object. This is useful for especially for conditional commands (subclasses of `LmiSelectCommand`) that devide connections to groups satisfying particular expression. These groups are turned into session proxies containing just a subset of connections in global session object.

Parameters **session** – Session object.

command.checkresult This module defines LmiCheckResult command class and related utilities.

class `lmi.scripts.common.command.checkresult.`**LmiCheckResult**(**args*, ***kwargs*)

Run an associated action and check its result. It implicitly makes no output if the invocation is successful and expected result matches.

List of additional recognized properties:

EXPECT: Value, that is expected to be returned by invoked associated function. This can also be a callable taking two arguments:

1. options - Dictionary with parsed command line options returned by `docopt`.
2. result - Return value of associated function.

Using metaclass: `CheckResultMetaClass`.

check_result(*options*, *result*)

Check the returned value of associated function.

Parameters

- **options** (*dictionary*) – Dictionary as returned by `docopt` parser after running `transform_options()`.
- **result** – Any return value that will be compared against what is expected.

Returns Whether the result is expected value or not. If `tuple` is returned, it contains `(passed_flag, error_description)`.

Return type boolean or tuple.

take_action(*connection*, *args*, *kwargs*)

Invoke associated method and check its return value for single host.

Parameters

- **args** (*list*) – List of arguments to pass to the associated function.
- **kwargs** (*dictionary*) – Keyword arguments to pass to the associated function.

Returns Exit code (0 on success).

Return type integer

exception lmi.scripts.common.command.checkresult.**LmiResultFailed**
Exception raised when associated function returns unexpected result. This is evaluated by LmiCheckResult.check_result() method.

command.endpoint Defines base command class for all endpoint commands. Those having no children.

class lmi.scripts.common.command.endpoint.**LmiEndPointCommand**(**args*, ***kwargs*)
Base class for any leaf command.

List of additional recognized properties:

CALLABLE [tuple] Associated function. Will be wrapped in LmiEndPointCommand.execute() method and will be accessible directly as a cmd.execute.dest property. It may be specified either as a string in form "<module_name>:<callable>" or as a reference to callable itself.

ARG_ARRAY_SUFFIX [str] String appended to every option parsed by docopt having list as an associated value. It defaults to empty string. This modification is applied before calling LmiEndPointCommand.verify_options() and LmiEndPointCommand.transform_options().

FORMATTER [callable] Default formatter factory for instances of given command. This factory accepts an output stream as the only parameter and returns an instance of Formatter.

Using metaclass: meta.EndPointCommandMetaClass.

classmethod dest_pos_args_count()
Number of positional arguments the associated function takes from command. These arguments are created by the command alone – they do not belong to options in usage string. Function can take additional positional arguments that need to be covered by usage string.

Return type integer

execute(**args*, ***kwargs*)
Subclasses must override this method to pass given arguments to command library function. This function shall be specified in CALLABLE property.

formatter
Return instance of default formatter.

Return type Formatter

formatter_factory()
Subclasses shall override this method to provide default formatter factory for printing output.

Returns Subclass of basic formatter.

produce_output(*data*)
This method can be use to render and print results with default formatter.

Parameters **data** – Is an object expected by the produce_output() method of formatter.

run(*args*)
Create options dictionary from input arguments, verify them, transform them, make positional and keyword arguments out of them and pass them to process_session().

Parameters **args** (*list*) – List of command arguments.

Returns Exit code of application.

Return type integer

run_with_args(*args*, *kwargs*)
Process end-point arguments and exit.

Parameters

- **args** (*list*) – Positional arguments to pass to associated function in command library.
- **kwargs** (*dictionary*) – Keyword arguments as a dictionary.

Returns Exit code of application.

Return type integer

transform_options(*options*)
This method can be overriden in subclasses if options shall be somehow modified before passing them associated function.

Note: Run after `verify_options()` method.

Parameters **options** (*dictionary*) – Dictionary as returned by `docopt` parser.

verify_options(*options*)
This method can be overriden in subclasses to check, whether the options given on command line are valid. If any flaw is discovered, an `LmiInvalidOptions` exception shall be raised. Any returned value is ignored.

Note: This is run before `transform_options()` method.

Parameters **options** (*dictionary*) – Dictionary as returned by `docopt` parser.

lmi.scripts.common.command.endpoint.**opt_name_sanitize**(*opt_name*)
Make a function parameter name out of option name. This replaces any character not suitable for python identificator with `'_'` and make the whole string lowercase.

Parameters **opt_name** (*string*) – Option name.

Returns Modified option name.

Return type string

lmi.scripts.common.command.endpoint.**options_dict2kwargs**(*options*)
Convert option name from resulting `docopt` dictionary to a valid python identificator token used as function argument name.

Parameters **options** (*dictionary*) – Dictionary returned by docopt call.

Returns New dictionary with keys passable to function as argument names.

Return type dictionary

command.helper Module with convenient function for defining user commands.

lmi.scripts.common.command.helper.**make_list_command**(*func*, *name=None*, *columns=None*, *verify_func=None*, *transform_func=None*)

Create a command subclassed from LmiLister. Please refer to this class for detailed usage.

Parameters

- **func** (*string or callable*) – Contents of CALLABLE property.
- **name** (*string*) – Optional name of resulting class. If not given, it will be made from the name of associated function.
- **columns** (*tuple*) – Contents of COLUMNS property.
- **verify_func** (*callable*) – Callable overriding py:meth:~.*endpoint.LmiEndPointCommand.verify_options* method.
- **transform_func** (*callable*) – Callable overriding transform_options() method.

Returns Subclass of LmiLister.

Return type type

lmi.scripts.common.command.helper.**register_subcommands**(*command_name*, *usage*, *command_map*, *fallback_command=None*)

Create a multiplexer command (a node in a tree of commands).

Parameters

- **command_name** (*string*) – Name of created command. The same as will be given on a command line.
- **usage** (*string*) – Usage string parseable by docopt.
- **command_map** (*dictionary*) – Dictionary of subcommands. Associates command names to their factories. It's assigned to COMMANDS property.
- **fallback_command** (LmiEndPointCommand) – Command factory used when no command is given on command line.

Returns Subclass of LmiCommandMultiplexer.

Return type type

lmi.scripts.common.command.helper.**select_command**(*command_name*, **args*, ***kwargs*)

Create command selector that loads command whose requirements are met.

Example of invocation:

```
Hardware = select_command('Hardware',
    ("Openlmi-Hardware >= 0.4.2", "lmi.scripts.hardware.current.Cmd"),
    ("Openlmi-Hardware < 0.4.2" , "lmi.scripts.hardware.pre042.Cmd"),
    default=HwMissing
)
```

Above example checks remote broker for OpenLMI-Hardware provider. If it is installed and its version is equal or higher than 0.4.2, command from current module will be used. For older registered versions command contained in pre042 module will be loaded. If hardware provider is not available, HwMissing command will be loaded instead.

See also:

Check out the grammer describing language used in these conditions at `lmi.scripts.common.versioncheck.parser`.

Parameters

- **args** – List of pairs `(condition, command)` that are inspected in given order until single condition is satisfied. Associated command is then loaded. Command is either a reference to command class or path to it given as string. In latter case last dot divides module's import path and command name.
- **default** – This command will be loaded when no condition from *args* is satisfied.

command.lister Defines command classes producing tablelike output.

class `lmi.scripts.common.command.lister.`**LmiBaseListerCommand**(**args*, ***kwargs*)
Base class for all lister commands.

classmethod **get_columns**()

Returns Column names for resulting table. `COLUMNS` property will be converted to this class method. If `None`, the associated function shall return column names as the first tuple of returned list. If empty tuple or list, no header shall be printed and associated function returns just data rows.

Return type list or tuple or None

class `lmi.scripts.common.command.lister.`**LmiInstanceLister**(**args*, ***kwargs*)
End point command outputting a table of instances for each host. Associated function shall return a list of instances. They may be prepended with column names depending on value of `DYNAMIC_PROPERTIES`. Each instance will occupy single row of table with property values being a content of cells.

List of additional recognized properties is the same as for `LmiShowInstance`. There is just one difference. Either `DYNAMIC_PROPERTIES` must be `True` or `PROPERTIES` must be filled.

Using metaclass: `InstanceListerMetaClass`.

classmethod **render**(*_self*, *inst*)
Return tuple of `(column_names, values)` ready for output by formatter.

take_action(*connection*, *args*, *kwargs*)
Collects results of single host.

Parameters

- **connection** (`lmi.shell.LMIConnection`) – Connection to a single host.
- **args** (*list*) – Positional arguments for associated function.
- **kwargs** (*dictionary*) – Keyword arguments for associated function.

Returns Column names and item list as a pair.

Return type tuple

class `lmi.scripts.common.command.lister.`**LmiLister**(**args*, ***kwargs*)
End point command outputting a table for each host. Associated function shall return a list of rows. Each row is represented as a tuple holding column values.

List of additional recognized properties:

COLUMNS [`tuple`] Column names. It's a tuple with name for each column. Each row shall then contain the same number of items as this tuple. If omitted, associated function is expected to provide them in the first row of returned list. It's translated to `get_columns()` class method.

Using metaclass: `ListerMetaClass`.

take_action(*connection*, *args*, *kwargs*)
Collects results of single host.

Parameters

- **connection** (`lmi.shell.LMIConnection`) – Connection to a single host.
- **args** (*list*) – Positional arguments for associated function.
- **kwargs** (*dictionary*) – Keyword arguments for associated function.

Returns Column names and item list as a pair.

Return type tuple

command.meta Meta classes simplifying declaration of user commands.

Each command is defined as a class with a set of properties. Some are mandatory, the others have some default values. Each of them is transformed by metaclasse to some function, class method or other property depending on command type and semantic of property. Property itself is removed from resulting class after being processed by meta class.

class `lmi.scripts.common.command.meta.`**CheckResultMetaClass**
Meta class for end-point command "check result". Additional handled properties:

EXPECT: Value to compare against the return value. Mandatory property.

`EXPECT` property is transformed into a `checkresult.LmiCheckResult.check_result()` method taking two arguments `(options, result)` and returning a boolean.

class `lmi.scripts.common.command.meta.`**EndPointCommandMetaClass**
End point command does not have any subcommands. It's a leaf of command tree. It wraps some function in command library being referred to as an *associated function*. It handles following class properties:

CALLABLE [`str` or callable] An associated function. Mandatory property.

OWN_USAGE [`bool` or `str`] Usage string. Optional property.

ARG_ARRAY_SUFFIX [`str`] Suffix added to argument names containing array of values. Optional property.

FMT_NO_HEADINGS [`bool`] Allows to force printing of table headers on and off for this command. Default is to print them.

FMT_HUMAN_FRIENDLY [`bool`] Tells formatter to make the output more human friendly. The result is dependent on the type of formatter used.

class `lmi.scripts.common.command.meta.`**InstanceListerMetaClass**
Meta class for instance lister command handling the same properties as `ShowInstanceMetaClass`.

class `lmi.scripts.common.command.meta.`**ListerMetaClass**
Meta class for end-point lister commands. Handles following class properties:

COLUMNS [`tuple`] List of column names. Optional property. There are special values such as:

None or omitted Associated function provides column names in a first row of returned list or generator.

empty list, empty tuple or False They mean that no headers shall be printed. It is simalar to using `FMT_NO_HEADINGS = True`. But in this case all the rows returned from associated functions are treated as data.

class `lmi.scripts.common.command.meta.`**MultiplexerMetaClass**
Meta class for node command (not an end-point command). It handles following class properties:

COMMANDS [dict] Command names with assigned command classes. Each of them is a direct subcommands of command with this property. Mandatory property.

FALLBACK_COMMAND [LmiEndPointCommand] Command factory to use in case that no command is passed on command line.

Formatting options (starting with FMT_ are also accepted, and may used to set defaults for all subcommands.

class lmi.scripts.common.command.meta.**SelectMetaClass**
Meta class for select commands with guarded commands. Additional handled properties:

SELECT [list] List of commands guarded with expressions representing requirements on server's side that need to be met.

DEFAULT [str or LmiBaseCommand] Defines fallback command used in case no condition can is satisfied.

class lmi.scripts.common.command.meta.**SessionCommandMetaClass**
Meta class for commands operating upon a session object. All associated functions take as first argument an namespace abstraction of type lmi.shell.

Handles following class properties:

NAMESPACE [str] CIM namespace abstraction that will be passed to associated function. Defaults to "root/cimv2". If False, raw lmi.shell.LMIConnection object will be passed to associated function.

class lmi.scripts.common.command.meta.**ShowInstanceMetaClass**
Meta class for end-point show instance commands. Additional handled properties:

DYNAMIC_PROPERTIES [bool] Whether the associated function itself provides list of properties. Optional property.

PROPERTIES [tuple] List of instance properties to print. Optional property.

These are translated in a render(), which should be marked as abstract in base lister class.

command.multiplexer Defines command class used to nest multiple commands under one.

class lmi.scripts.common.command.multiplexer.**LmiCommandMultiplexer**(*app*, *cmd_name*, *parent=None*)
Base class for node commands. It consumes just part of command line arguments and passes the remainder to one of its subcommands.

Example usage:

```
class MyCommand(LmiCommandMultiplexer):
    '''
    My command description.

    Usage: %(cmd)s mycommand (subcmd1 | subcmd2)
    '''
    COMMANDS = {'subcmd1' : Subcmd1, 'subcmd2' : Subcmd2}
```

Where Subcmd1 and Subcmd2 are some other LmiBaseCommand subclasses. Documentation string must be parseable with docopt.

Recognized properties:

COMMANDS [dictionary] property will be translated to LmiCommandMultiplexer.child_commands() class method by MultiplexerMetaClass.

Using metaclass: `meta.MultiplexerMetaClass`.

classmethod child_commands()
Abstract class method, that needs to be implemented in subclass. This is done by associated meta-class, when defining a command with assigned `COMMANDS` property.

Returns Dictionary of subcommand names with assigned command factories.

Return type dictionary

classmethod fallback_command()
This is overriden by `MultiplexerMetaClass` when the `FALLBACK_COMMAND` gets processed.

Returns Command factory invoked for missing command on command line.

Return type `LmiEndPointCommand`

run(*args*)
Handle optional parameters, retrieve desired subcommand name and pass the remainder of arguments to it.

Parameters **args** (*list*) – List of arguments with at least subcommand name.

run_subcommand(*cmd_name*, *args*)
Pass control to a subcommand identified by given name.

Parameters

- **cmd_name** (*string*) – Name of direct subcommand, whose `run()` method shall be invoked.
- **args** (*list*) – List of arguments for particular subcommand.

Returns Application exit code.

Return type integer

command.select Defines command class used to choose other commands depending on profile and class requirements.

class `lmi.scripts.common.command.select.`**LmiSelectCommand**(*app*, *cmd_name*, *parent=None*)
Base class for command selectors. It does not process command line arguments. Thery are passed unchanged to selected command whose requirements are met. Its doc string is not interpreted in any way.

If there are more hosts, conditions are evaluated per each. They are then split into groups, each fulfilling particular condition. Associated commands are then invoked on these groups separately.

Example usage:

```
class MySelect(LmiSelectCommand):
    SELECT = [
        ( 'OpenLMI-Hardware >= 0.4.2'
        , 'lmi.scripts.hardware.current.Cmd'),
        ('OpenLMI-Hardware', 'lmi.scripts.hardware.pre042.Cmd')
    ]
    DEFAULT = MissingHwProviderCmd
```

Using metaclass: `meta.SelectMetaClass`.

eval_expr(*expr*, *hosts*, *cache=None*)
Evaluate expression on group of hosts.

Parameters

- **expr** (*string*) – Expression to evaluate.
- **hosts** (*list*) – Group of hosts that shall be checked.
- **cache** (*dictionary*) – Optional cache object speeding up evaluation by reducing number of queries to broker.

Returns Subset of hosts satisfying *expr*.

Return type list

classmethod **get_conditionals**()
Get the expressions with associated commands. This shall be overriden by a subclass.

Returns Pair of `(expressions, default)`. Where `expressions` is a list of pairs `(condition, command)`. And `default` is the same as `command` used in case no `condition` is satisfied.

Return type list

get_usage(*proper=False*)
Try to get usage of any command satisfying some expression.

Raises Same exceptions as `select_cmds()`.

run(*args*)
Iterate over command factories with associated sessions and execute them with unchanged *args*.

select_cmds(*cache=None*)
Generator of command factories with associated groups of hosts. It evaluates given expressions on session. In this process all expressions from `get_conditionals()` are checked in a row. Host satisfying some expression is added to group associated with it and is excluded from processing following expressions.

Parameters **cache** (*dictionary*) – Optional cache object speeding up the evaluation by reducing number of queries to broker.

Returns Pairs in form `(command_factory, session_proxy)`.

Return type generator

Raises

- **LmiUnsatisfiedDependencies** if no condition is satisfied for at least one host. Note that this exception is raised at the end of evaluation. This lets you choose whether you want to process satisfied hosts - by processing the generator at once. Or whether you want to be sure it is satisfied by all of them - you turn the generator into a list.
- **LmiNoConnections** if no successful connection was done.

command.session Defines a base class for all command classes operating upon a `Session` object.

class `lmi.scripts.common.command.session.`**LmiSessionCommand**(**args*, ***kwargs*)
Base class for end-point commands operating upon a session object.

Using metaclass: `SessionCommandMetaClass`.

classmethod **cim_namespace**()
This returns default cim namespace, the connection object will be nested into before being passed to associated function. It can be overriden in few ways:

1.by setting `[CIM] Namespace` option in configuration

2.by giving `--namespace` argument on command line to the `lmi` meta-command

3.by setting `NAMESPACE` property in declaration of command

Higher number means higher priority.

classmethod dest_pos_args_count()
There is a namespace/connection object passed as the first positional argument.

execute_on_connection(*connection*, **args*, ***kwargs*)
Wraps the `execute()` method with connection adjustments. Connection object is not usually passed directly to associated function. Mostly it's the namespace object that is expected. This method checks, whether the namespace object is desired and modifies connection accordingly.

Parameters

- **connection** (`lmi.shell.LMIConnection`) – Connection to single host.
- **args** (*list*) – Arguments handed over to associated function.
- **kwargs** (*dictionary*) – Keyword arguments handed over to associated function.

process_host_result(*hostname*, *success*, *result*)
Called from `process_session()` after single host gets processed. By default this prints obtained *result* with default formatter if the execution was successful. Children of this class may want to override this.

Parameters

- **hostname** (*string*) – Name of host involved.
- **success** (*boolean*) – Whether the action on host succeeded.
- **result** – Either the value returned by associated function upon a successful invocation or an exception.

process_session(*session*, *args*, *kwargs*)
Process each host of given session, call the associated command function, collect results and print it to standard output.

Parameters

- **session** (`Session`) – Session object with set of hosts.
- **args** (*list*) – Positional arguments to pass to associated function in command library.
- **kwargs** (*dictionary*) – Keyword arguments as a dictionary.

Returns Exit code of application.

Return type integer

process_session_results(*session*, *results*)
Called at the end of `process_session()` 's execution. It's supposed to do any summary work upon results from all hosts. By default it just prints errors in a form of list.

Parameters

- **session** (`lmi.scripts.common.session.Session`) – Session object.
- **results** (*dictionary*) – Dictionary of form:

  ```
  { 'hostname' : (success_flag, result), ... }
  ```

  where *result* is either an exception or returned value of associated function, depending on *success_flag*. See the `process_host_result()`.

take_action(*connection*, *args*, *kwargs*)
Executes an action on single host and collects results.

Parameters

- **connection** (`lmi.shell.LMIConnection`) – Connection to a single host.
- **args** (*list*) – Positional arguments for associated function.
- **kwargs** (*dictionary*) – Keyword arguments for associated function.

Returns Column names and item list as a pair.

Return type tuple

command.show Contains command classes producing key-value pairs to output.

class `lmi.scripts.common.command.show.`**`LmiShowInstance`**(**args*, ***kwargs*)

End point command producing a list of properties of particular CIM instance. Either reduced list of properties to print can be specified, or the associated function alone can decide, which properties shall be printed. Associated function is expected to return CIM instance as a result.

List of additional recognized properties:

`DYNAMIC_PROPERTIES` [`bool`] A boolean saying, whether the associated function alone shall specify the list of properties of rendered instance. If `True`, the result of function must be a pair:

```
(props, inst)
```

Where props is the same value as can be passed to `PROPERTIES` property. Defaults to `False`.

`PROPERTIES` [`tuple`] May contain list of instance properties, that will be produced in the same order as output. Each item of list can be either:

name [`str`] Name of property to render.

pair [`tuple`] A tuple `(Name, render_func)`, where former item an arbitraty name for rendered value and the latter is a function taking as the only argument particular instance and returning value to render.

`DYNAMIC_PROPERTIES` and `PROPERTIES` are mutually exclusive. If none is given, all instance properties will be printed.

Using metaclass: `ShowInstanceMetaClass`.

classmethod `render`(*_self*, *inst*)

Return tuple of `(column_names, values)` ready for output by formatter.

`take_action`(*connection*, *args*, *kwargs*)

Process single connection to a host, render result and return a value to render.

Returns List of pairs, where the first item is a label and second a value to render.

Return type list

command.util Utility functions used in command sub-package.

`lmi.scripts.common.command.util.`**`RE_COMMAND_NAME = <_sre.SRE_Pattern object at 0x7f174b293cd8>`**

Command name can also be a single or double dash.

`lmi.scripts.common.command.util.`**`RE_OPT_BRACKET_ARGUMENT = <_sre.SRE_Pattern object at 0x7f1748bd4030>`**

Regular expression matching bracket argument such as `<arg_name>`.

`lmi.scripts.common.command.util.`**`RE_OPT_LONG_OPTION = <_sre.SRE_Pattern object at 0x7f1749aa7770>`**

Regular expression matching long options (prefixed with double dash).

lmi.scripts.common.command.util.**RE_OPT_SHORT_OPTION = <_sre.SRE_Pattern object at 0x7f1749083500>**
Regular expression matching showt options. They are one character long, prefixed with single dash.

lmi.scripts.common.command.util.**RE_OPT_UPPER_ARGUMENT = <_sre.SRE_Pattern object at 0x7f1748bcd030>**
Regular expression matching argument written in upper case such as ARG_NAME.

lmi.scripts.common.command.util.**get_module_name**(*frame_level=2*)
Get a module name of caller from particular outer frame.

Parameters **frame_level** (*integer*) – Number of nested frames to skip when searching for called function scope by inspecting stack upwards. When the result of this function is applied directly on the definition of function, it's value should be 1. When used from inside of some other factory, it must be increased by 1.

Level 0 returns name of this module. Level 1 returns module name of caller. Level 2 returns module name of caller's caller.

Returns Module name.

Return type string

lmi.scripts.common.command.util.**is_abstract_method**(*clss*, *method*, *missing_is_abstract=False*)
Check, whether the given method is abstract in given class or list of classes. May be used to check, whether we should override particular abstract method in a meta-class in case that no non-abstract implementation is defined.

Parameters

- **clss** (*type or tuple*) – Class or list of classes that is searched for non-abstract implementation of particular method. If the first class having particular method in this list contain non-abstract implementation, False is returned.
- **method** (*string*) – Name of method to look for.
- **missing_is_abstract** (*boolean*) – This is a value returned, when not such method is defined in a set of given classes.

Returns Are all occurences of given method abstract?

Return type boolean

configuration Module for Configuration class.

class lmi.scripts.common.configuration.**Configuration**(*user_config_file_path='~/.lmirc'*, ***kwargs*)
Configuration class specific to software providers. *OpenLMI* configuration file should reside in:

```
/etc/openlmi/scripts/lmi.conf
```

Parameters **user_config_file_path** (*string*) – Path to the user configuration options.

classmethod default_options()

Returns Dictionary of default values.

Return type dictionary

history_file
Path to a file with history of interactive mode.

history_max_length
Maximum number of lines kept in history file.

human_friendly
Whether to print human-friendly values.

lister_format

Output format used for lister commands. Returns one of

- LISTER_FORMAT_CSV
- LISTER_FORMAT_TABLE

Return type integer

load()
Read additional user configuration file if it exists.

log_file
Path to a file, where logging messages shall be written.

no_headings
Whether to print headings of tables.

silent
Whether to suppress all output messages except for errors.

trace
Whether the tracebacks shall be printed upon errors.

verbose
Whether to output more details.

verbosity
Return integer indicating verbosity level of output to console.

verify_server_cert
Return boolean saying, whether the server-side certificate should be checked.

lmi.scripts.common.configuration.**DEFAULT_FORMAT_STRING = '%(cseq)s%(levelname_)-8s:%(creset)s %(messa**
Default format string to use in stderr handlers.

errors Module with predefined exceptions for use in scripts.

exception lmi.scripts.common.errors.**LmiBadSelectExpression**(*module_name*, *class_name*, *expr*)
Raised, when expression of LmiSelectCommand could not be evaluated.

exception lmi.scripts.common.errors.**LmiCommandError**(*module_name*, *class_name*, *msg*)
Generic exception related to command declaration.

exception lmi.scripts.common.errors.**LmiCommandImportError**(*cmd_name*, *cmd_path*, *reason*)
Exception raised when command can not be imported.

exception lmi.scripts.common.errors.**LmiCommandInvalidCallable**(*module_name*, *class_name*, *msg*)
Raised, when given callback is not callable.

exception lmi.scripts.common.errors.**LmiCommandInvalidName**(*module_name*, *class_name*, *cmd_name*)
Raised, when command gets invalid name.

exception lmi.scripts.common.errors.**LmiCommandInvalidProperty**(*module_name*, *class_name*, *msg*)
Raised, when any command property contains unexpected value.

exception lmi.scripts.common.errors.**LmiCommandMissingCallable**(*module_name*, *class_name*)
Raised, when command declaration is missing callable object.

exception lmi.scripts.common.errors.**LmiCommandNotFound**(*cmd_name*)
Raised, when user requests not registered command.

exception lmi.scripts.common.errors.**LmiError**
The base Lmi scripts error. All the other exceptions inherit from it.

exception lmi.scripts.common.errors.**LmiFailed**
Raised, when operation on remote host failes. It's supposed to be used especially in command libraries.

exception lmi.scripts.common.errors.**LmiImportCallableFailed**(*module_name*, *class_name*, *callable_prop*)
Raised, when callable object of command could not be imported.

exception lmi.scripts.common.errors.**LmiInvalidOptions**
Raised in `verify_options()` method if the options given are not valid.

exception lmi.scripts.common.errors.**LmiNoConnections**
Raised, when no connection to remote hosts could be made.

exception lmi.scripts.common.errors.**LmiTerminate**(*exit_code=0*)
Raised to cleanly terminate interavtive shell.

exception lmi.scripts.common.errors.**LmiUnexpectedResult**(*command_class*, *expected*, *result*)
Raised, when command's associated function returns something unexpected.

exception lmi.scripts.common.errors.**LmiUnsatisfiedDependencies**(*uris*)
Raised when no guarded command in `LmiSelectCommand` can be loaded due to unsatisfied requirements.

formatter Subpackage with formatter classes used to render and output results.

Each formatter has a `Formatter.produce_output()` method taking one argument which gets rendered and printed to output stream. Each formatter expects different argument, please refer to doc string of particular class.

class lmi.scripts.common.formatter.**CsvFormatter**(*stream*, *padding=0*, *no_headings=False*)
Renders data in a csv (Comma-separated values) format.

This formatter supports following commands:

- `NewHostCommand`
- `NewTableCommand`
- `NewTableHeaderCommand`

class lmi.scripts.common.formatter.**ErrorFormatter**(*stream*, *padding=4*)

Render error strings for particular host. Supported commands:

- `NewHostCommand`

class lmi.scripts.common.formatter.**Formatter**(*stream*, *padding=0*, *no_headings=False*)
Base formatter class.

It produces string representation of given argument and prints it.

This formatter supports following commands:

- `NewHostCommand`.

Parameters

- **stream** (*file*) – Output stream.
- **padding** (*integer*) – Number of leading spaces to print at each line.
- **no_headings** (*boolean*) – If table headings should be omitted.

encoding
Try to determine encoding for output terminal.

Returns Encoding used to encode unicode strings.

Return type string

host_counter = None
counter of hosts printed

line_counter = None
counter of lines producted for current table

print_host(*hostname*)
Prints header for new host.

Parameters **hostname** (*string*) – Hostname to print.

print_line(*line*, **args*, ***kwargs*)
Prints single line. Output message is prefixed with `padding` spaces, formated and printed to output stream.

Parameters

- **line** (*string*) – Message to print, it can contain markers for other arguments to include according to `format_spec` language. Please refer to `Format Specification Mini-Language` in python documentation.
- **args** (*list*) – Positional arguments to `format()` function of `line` argument.
- **kwargs** (*dictionary*) – Keyword arguments to `format()` function.

produce_output(*data*)
Render and print given data.

Data can be also instance of `FormatterCommand`, see documentation of this class for list of allowed commands.

This shall be overridden by subclasses.

Parameters **data** – Any value to print. Subclasses may specify their requirements for this argument. It can be also am instance of `FormatterCommand`.

render_value(*val*)
Rendering function for single value.

Parameters **val** – Any value to render.

Returns Value converted to its string representation.

Return type str

table_counter = None
counter of tables produced for current host

class lmi.scripts.common.formatter.**ListFormatter**(*stream*, *padding=0*, *no_headings=False*)

Base formatter taking care of list of items. It renders input data in a form of table with mandatory column names at the beginning followed by items, one occupying single line (row).

This formatter supports following commands:

- NewHostCommand
- NewTableCommand
- NewTableHeaderCommand

The command must be provided as content of one row. This row is then not printed and the command is executed.

This class should be subclassed to provide nice output.

print_header()

Print table header.

print_row(*data*)

Print data row. Optionaly print header, if requested.

Parameters **data** (*tuple*) – Data to print.

print_table_title(*title*)

Prints title of next tale.

Parameters **title** (*string*) – Title to print.

print_text_row(*row*)

Print data row without any header.

Parameters **row** (*tuple*) – Data to print.

produce_output(*rows*)

Prints list of rows.

There can be a FormatterCommand instance instead of a row. See documentation of this class for list of allowed commands.

Parameters **rows** (list, generator or command.FormatterCommand) – List of rows to print.

class lmi.scripts.common.formatter.**ShellFormatter**(*stream*, *padding=0*, *no_headings=False*)

Specialization of SingleFormatter having its output executable as a shell script.

This formatter supports following commands:

- NewHostCommand

class lmi.scripts.common.formatter.**SingleFormatter**(*stream*, *padding=0*, *no_headings=False*)

Meant to render and print attributes of single object. Attributes are rendered as a list of assignments of values to variables (attribute names).

This formatter supports following commands:

- NewHostCommand

produce_output(*data*)

Render and print attributes of single item.

There can be a FormatterCommand instance instead of a data. See documentation of this class for list of allowed commands.

Parameters **data** (*tuple or dict*) – Is either a pair of property names with list of values or a dictionary with property names as keys. Using the pair allows to order the data the way it should be printing. In the latter case the properties will be sorted by the property names.

class lmi.scripts.common.formatter.**TableFormatter**(*stream*, *padding=0*, *no_headings=False*, *min_column_sizes=False*)

Print nice human-readable table to terminal.

To print the table nicely aligned, the whole table must be populated first. Therefore this formatter stores all rows locally and does not print them until the table is complete. Column sizes are computed afterwards and the table is printed at once.

This formatter supports following commands:

- NewHostCommand
- NewTableCommand
- NewTableHeaderCommand

The command must be provided as content of one row. This row is then not printed and the command is executed.

Parameters **min_column_sizes** (*dictionary*) – Dictionary of minimal column sizes, where keys are column numbers starting from 0, and values are minimal column sizes.

print_host(*hostname*)
Prints header for new host.

Parameters **hostname** (*string*) – Hostname to print.

print_row(*data*)
Print data row.

Parameters **data** (*tuple*) – Data to print.

print_table_title(*title*)
Prints title of next tale.

Parameters **title** (*string*) – Title to print.

produce_output(*rows*)
Prints list of rows.

There can be FormatterCommand instance instead of a row. See documentation of this class for list of allowed commands.

Parameters **rows** (*list or generator*) – List of rows to print.

lmi.scripts.common.formatter.**get_terminal_width**()
Get the number of columns of current terminal if attached to it. It defaults to 79 characters.

Returns Number of columns of attached terminal.

Return type integer

formatter.command Contains command classes used to control formatters from inside of command execution functions.

class lmi.scripts.common.formatter.command.**FormatterCommand**
Base class for formatter commands.

class lmi.scripts.common.formatter.command.**NewHostCommand**(*hostname*)
Command for formatter to finish current table (if any), print header for new host and (if there are any data) print table header.

Parameters **hostname** (*string*) – Name of host appearing at the front of new table.

class lmi.scripts.common.formatter.command.**NewTableCommand**(*title=None*)
Command for formatter to finish current table (if any), print the **title** and (if there are any data) print table header.

Parameters **title** (*string*) – Optional title for new table.

class lmi.scripts.common.formatter.command.**NewTableHeaderCommand**(*columns=None*)
Command for formatter to finish current table (if any), store new table header and (if there are any data) print the table header. The table header will be printed in all subsequent tables, until new instance of this class arrives.

Parameters **columns** (*tuple*) – Array of column names.

lmi_logging Utilities for logging framework.

lmi.scripts.common.lmi_logging.**LOG_LEVEL_2_COLOR = {40: 9, 50: 13, 30: 11}**
Dictionary assigning color code to logging level.

class lmi.scripts.common.lmi_logging.**LevelDispatchingFormatter**(*formatters*, *default='%(cseq)s%(levelname_)-8s:%(creset)s %(message)s'*, *datefmt=None*)
Formatter class for logging module. It allows to predefine different format string used for some level ranges.

Parameters

- **formatters** (*dict*) – Mapping of module names to *format*. It is a dictionary of following format:

```
{ max_level1 : format1
, max_level2 : format2
, ... }
```

format in parameters description can be either `string` or another formatter object.

For example if you want to have format3 used for *ERROR* and *CRITICAL* levels, *format2* for *INFO* and *format1* for anything else, your dictionary will look like this:

```
{ logging.INFO - 1 : format1
, logging.INFO     : format2 }
```

And the *default* value should have *format3* assigned.

- **default** – Default *format* to use. This format is used for all levels higher than the maximum of *formatters*' keys.

format(*record*)
Interface for logging module.

class lmi.scripts.common.lmi_logging.**LogRecord**(*name*, *level*, **args*, ***kwargs*)
Overrides `logging.LogRecord`. It adds new attributes:

- *levelname_* - Name of level in lowercase.
- ***cseq* - Escape sequence for terminal used to set color** assigned to particular log level.
- ***creset* - Escape sequence for terminal used to reset foreground** color.

These can be used in format strings initializing logging formatters.

Accepts the same arguments as base class.

lmi.scripts.common.lmi_logging.**get_color_sequence**(*color_code*)
Computer color sequence for particular color code.

Returns Escape sequence for terminal used to set foreground color.

Return type str

lmi.scripts.common.lmi_logging.**get_logger**(*module_name*)
Convenience function for getting callable returning logger for particular module name. It's supposed to be used at module's level to assign its result to global variable like this:

```
from lmi.scripts import common

LOG = common.get_logger(__name__)
```

This can be used in module's functions and classes like this:

```
def module_function(param):
    LOG().debug("This is debug statement logging param: %s", param)
```

Thanks to `LOG` being a callable, it always returns valid logger object with current configuration, which may change overtime.

Parameters **module_name** (*string*) – Absolute dotted module path.

Return type `logging.Logger`

lmi.scripts.common.lmi_logging.**setup_logger**(*use_colors=True*)
This needs to be called before any logging takes place.

session Module for session object representing all connection to remote hosts.

class lmi.scripts.common.session.**Session**(*app*, *hosts*, *credentials=None*, *same_credentials=False*)
Session object keeps connection objects to remote hosts. Their are associated with particular hostnames. It also caches credentials for them. Connections are made as they are needed. When credentials are missing for connection to be made, the user is asked to supply them from standard input.

Parameters

- **app** – Instance of main application.
- **hosts** (*list*) – List of hostname strings.
- **credentials** (*dictionary*) – Mapping assigning a pair `(user, password)` to each hostname.
- **same_credentials** (*boolean*) – Use the same credentials for all hosts in session. The first credentials given will be used.

get_credentials(*hostname*)

Parameters **hostname** (*string*) – Name of host to get credentials for.

Returns Pair of `(username, password)` for given hostname. If no credentials were given for this host, `('', '')` is returned.

Return type tuple

get_unconnected()

Returns List of hostnames, which do not have associated connection yet.

Return type list

hostnames
List of hostnames in session.

Return type list

class lmi.scripts.common.session.**SessionProxy**(*session*, *uris*)
Behaves like a session. But it just encapsulates other session object and provides access to a subset of its items.

Parameters

- **session** – Session object or even another session proxy.
- **uris** (*list*) – Subset of uris in encapsulated session object.

util Various utilities for LMI Scripts.

class lmi.scripts.common.util.**FilteredDict**(*key_filter*, *original=None*)
Dictionary-like collection that wraps some other dictionary and provides limited access to its keys and values. It permits to get, delete and set items specified in advance.

Note: Please use only the methods overriden. This class does not guarantee 100% API compliance. Not overriden methods won't work properly.

Parameters

- **key_filter** (*list*) – Set of keys that can be get, set or deleted. For other keys, `KeyError` will be raised.
- **original** (*dictionary*) – Original dictionary containing not only keys in *key_filter* but others as well. All modifying operations operate also on this dictionary. But only those keys in *key_filter* can be affected by them.

versioncheck Package with utilities for checking availability of profiles or CIM classes. Version requirements can also be specified.

lmi.scripts.common.versioncheck.**cmp_profiles**(*fst*, *snd*)
Compare two profiles by their version.

Returns

- -1 if the *fst* profile has lower version than *snd*
- 0 if their versions are equal
- 1 otherwise

Return type int

lmi.scripts.common.versioncheck.**eval_respl**(*expr*, *conn*, *namespace=None*, *cache=None*)
Evaluate LMIReSpL expression on particular broker.

Parameters

- **expr** (*string*) – Expression to evaluate.
- **conn** – Connection object.
- **namespace** (*string*) – Optional CIM namespace where CIM classes will be searched.

- **cache** (*dictionary*) – Optional cache speeding up evaluation.

Returns `True` if requirements in expression are satisfied.

Return type boolean

`lmi.scripts.common.versioncheck.`**`get_class_version`**(*conn*, *name*, *namespace=None*, *cache=None*)

Query broker for version of particular CIM class. Version is stored in `Version` qualifier of particular CIM class.

Parameters

- **conn** – Connection object.
- **name** (*string*) – Name of class to query.
- **namespace** (*string*) – Optional CIM namespace. Defaults to configured namespace.
- **cache** (*dictionary*) – Optional cache used to speed up expression prrocessing.

Returns Version of CIM matching class. Empty string if class is registered but is missing `Version` qualifier and `None` if it is not registered.

Return type string

`lmi.scripts.common.versioncheck.`**`get_profile_version`**(*conn*, *name*, *cache=None*)

Get version of registered profile on particular broker. Queries `CIM_RegisteredProfile` and `CIM_RegisteredSubProfile`. The latter comes in question only when `CIM_RegisteredProfile` does not yield any matching result.

Parameters

- **conn** – Connection object.
- **name** (*string*) – Name of the profile which must match value of *RegisteredName* property.
- **cache** (*dictionary*) – Optional cache where the result will be stored for later use. This greatly speeds up evaluation of several expressions refering to same profiles or classes.

Returns Version of matching profile found. If there were more of them, the highest version will be returned. `None` will be returned when no matching profile or subprofile is found.

Return type string

versioncheck.parser Parser for mini-language specifying profile and class requirements. We call the language LMIReSpL (openLMI Requirement Specification Language).

The only thing designed for use outside this module is `bnf_parser()`.

Language is generated by BNF grammer which served as a model for parser.

Formal representation of BNF grammer is following:

```
expr          ::= term [ op expr ]*
term          ::= '!'? req
req           ::= profile_cond | clsreq_cond | '(' expr ')'
profile_cond  ::= 'profile'? [ profile | profile_quot ] cond?
clsreq_cond   ::= 'class' [ clsname | clsname_quot] cond?
profile_quot  ::= '"' /\w+[ +.a-zA-Z0-9_-]*/ '"'
profile       ::= /\w+[+.a-zA-Z_-]*/
clsname_quot  ::= '"' clsname '"'
clsname       ::= /[a-zA-Z]+_[a-zA-Z][a-zA-Z0-9_]*/
cond          ::= cmpop version
cmpop         ::= /(<|=|>|!)=|<|>/
```

```
version       ::= /[0-9]+(\.[0-9]+)*/
op            ::= '&' | '|'
```

String surrounded by quotes is a literal. String enclosed with slashes is a regular expression. Square brackets encloses a group of words and limit the scope of some operation (like iteration).

class lmi.scripts.common.versioncheck.parser.**And**(*fst*, *snd*)
Represents logical *AND* of two expressions. Short-circuit evaluation is being exploited here.

Parameters

- **fst** – An object of Term non-terminal.
- **snd** – An object of Term non-terminal.

class lmi.scripts.common.versioncheck.parser.**Expr**(*term*)
Initial non-terminal. Object of this class (or one of its subclasses) is a result of parsing.

Parameters **term** – An object of Term non-terminal.

lmi.scripts.common.versioncheck.parser.**OP_MAP = {'>=': (<built-in function ge>, <built-in function all>), '==': (**
Dictionary mapping supported comparison operators to a pair. First item is a function making the comparison and the second can be of two values (all or any). Former sayes that each part of first version string must be in relation to corresponding part of second version string in order to satisfy the condition. The latter causes the comparison to end on first satisfied part.

class lmi.scripts.common.versioncheck.parser.**Or**(*fst*, *snd*)
Represents logical *OR* of two expressions. Short-circuit evaluation is being exploited here.

Parameters

- **fst** – An object of Term non-terminal.
- **snd** – An object of Term non-terminal.

class lmi.scripts.common.versioncheck.parser.**Req**
Represents one of following subexpressions:

•single requirement on particular profile

•single requirement on particular class

•a subexpression

class lmi.scripts.common.versioncheck.parser.**ReqCond**(*kind*, *version_getter*, *name*, *cond=None*)
Represents single requirement on particular class or profile.

Parameters

- **kind** (*str*) – Name identifying kind of thing this belongs. For example 'class' or 'profile'.
- **version_getter** (*callable*) – Is a function called to get version of either profile or CIM class. It must return corresponding version string if the profile or class is registered and None otherwise. Version string is read from RegisteredVersion property of CIM_RegisteredProfile. If a class is being queried, version shall be taken from Version qualifier of given class.
- **name** (*str*) – Name of profile or CIM class to check for. In case of a profile, it is compared to RegisteredName property of CIM_RegisteredProfile. If any instance of this class has matching name, it's version will be checked. If no matching instance is found, instances of CIM_RegisteredSubProfile are queried the same way. Failing to find it results in False.

- **cond** (*str*) – Is a version requirement. Check the grammer above for `cond` non-terminal.

class `lmi.scripts.common.versioncheck.parser.`**SemanticGroup**
Base class for non-terminals. Just a minimal set of non-terminals is represented by objects the rest is represented by strings.

All subclasses need to define their own `evaluate()` method. The parser builds a tree of these non-terminals with single non-terminal being a root node. This node's *evaluate* method returns a boolean saying whether the condition is satisfied. Root node is always an object of `Expr`.

evaluate()

Returns `True` if the sub-condition represented by this non-terminal is satisfied.

Return type boolean

class `lmi.scripts.common.versioncheck.parser.`**Subexpr**(*expr*)
Represents a subexpression originally enclosed in brackets.

class `lmi.scripts.common.versioncheck.parser.`**Term**(*req*, *negate*)
Represents possible negation of expression.

Parameters

- **req** – An object of `Req`.
- **negate** (*boolean*) – Whether the result of children shall be negated.

class `lmi.scripts.common.versioncheck.parser.`**TreeBuilder**(*stack*, *profile_version_getter*, *class_version_getter*)
A stack interface for parser. It defines methods modifying the stack with additional checks.

expr(*strg*, *loc*, *toks*)
Operates upon a stack. It takes either one or two *terms* there and makes an expression object out of them. Terms need to be delimited with logical operator.

push_class(*strg*, *loc*, *toks*)
Handles `clsreq_cond` non-terminal in one go. It extracts corresponding tokens and pushes an object of `ReqCond` to a stack.

push_literal(*strg*, *loc*, *toks*)
Pushes operators to a stack.

push_profile(*strg*, *loc*, *toks*)
Handles `profile_cond` non-terminal in one go. It behaves in the same way as `push_profile()`.

subexpr(*strg*, *loc*, *toks*)
Operates upon a stack. It creates an instance of `Subexpr` out of `Expr` which is enclosed in brackets.

term(*strg*, *loc*, *toks*)
Creates a `term` out of requirement (`req` non-terminal).

`lmi.scripts.common.versioncheck.parser.`**bnf_parser**(*stack*, *profile_version_getter*, *class_version_getter*)
Builds a parser operating on provided stack.

Parameters

- **stack** (*list*) – Stack to operate on. It will contain the resulting `Expr` object when the parsing is successfully over - it will be the only item in the list. It needs to be initially empty.
- **profile_version_getter** (*callable*) – Function returning version of registered profile or `None` if not present.

- **class_version_getter** (*callable*) – Fucntion returning version of registered class or `None` if not present.

Returns Parser object.

Return type `pyparsing,ParserElement`

`lmi.scripts.common.versioncheck.parser.`**`cmp_version`**(*fst*, *snd*, *opsign='<'*)
Compare two version specifications. Each version string shall contain digits delimited with dots. Empty string is also valid version. It will be replaced with -1.

Parameters

- **fst** (*str*) – First version string.
- **snd** (*str*) – Second version string.
- **opsign** (*str*) – Sign denoting operation to be used. Supported signs are present in `OP_MAP`.

Returns `True` if the relation denoted by particular operation exists between two operands.

Return type boolean

Account Script python reference

LMI account provider client library.

This set of functions can create, modify and delete users and groups on a remote managed system.

`lmi.scripts.account.`**`add_to_group`**(*ns*, *group*, *users*)
Add users to a group.

Parameters

- **group** (*LMIInstance or LMIInstanceName of LMI_Group.*) – The group.
- **users** (*List (or generator) of LMIInstances or LMIInstanceNames of LMI_Account.*) – Users to add.

`lmi.scripts.account.`**`create_group`**(*ns*, *group*, *reserved=False*, *gid=None*)
Create a new group on the system.

Parameters

- **group** (*string*) – Name of the group.
- **reserved** (*boolean*) – Whether the group is a system one (its GID will be allocated lower than system-defined threshold).
- **gid** (*int*) – Required GID. It will be allocated automatically if it's None.

Return type LMIInstanceName of the created group.

Returns Created group.

`lmi.scripts.account.`**`create_user`**(*ns*, *name*, *gecos=None*, *home=None*, *create_home=True*, *shell=None*, *uid=None*, *gid=None*, *create_group=True*, *reserved=False*, *password=None*, *plain_password=False*)
Create a new user.

Parameters

- **name** (*string*) – Name of the user.
- **gecos** (*string*) – GECOS information of the new user.

- **home** (*string*) – Home directory.
- **create_home** (*boolean*) – True, if home directory should be automatically created.
- **shell** (*string*) – User's shell.
- **uid** (*int*) – Desired UID. If None, system will allocate a free one.
- **gid** (*int*) – Desired GID. If None, system will allocate a free one.
- **create_group** (*boolean*) – True, if user's private group should be created.
- **reserved** (*boolean*) – True, if the account is system one, i.e. it's UID will be allocated in system account space (below system defined threshold). (default=False, the account is an user).
- **password** (*string*) – User password.
- **plain_password** (*boolean*) – True, if the provided password is plain text string, False if it is already hashed by crypt().

Return type LMIInstanceName

Returns Created used.

lmi.scripts.account.**delete_group**(*ns*, *group*)
Delete a group.

Parameters **group** (*LMIInstance or LMIInstanceName of LMI_Group.*) – The group to delete.

lmi.scripts.account.**delete_user**(*ns*, *user*, *no_delete_group=False*, *no_delete_home=False*, *force=False*)
Delete a user.

Parameters

- **user** (*LMIInstance or LMIInstanceName of LMI_Account.*) – User to delete.
- **no_delete_group** (*boolean*) – True, if the user's private group should be preserved. (default = False, the group is deleted).
- **no_delete_home** (*boolean*) – True, if user's home directory should be preserved. (default = False, home is deleted).
- **force** (*boolean*) – True, if the home directory should be remove even though the user is not owner of the directory. (default = False, do not remove user's home if it is owned by someone else).

lmi.scripts.account.**get_group**(*ns*, *groupname*)
Return LMIInstance of the group. This function raises LmiFailed if the user is not found.

Parameters **groupname** (*string*) – Name of the group.

Return type LMIInstance of LMI_Group

Returns The group.

lmi.scripts.account.**get_user**(*ns*, *username*)
Return LMIInstance of the user. This function raises LmiFailed if the user is not found.

Parameters **username** (*string*) – Name of the user.

Return type LMIInstance of LMI_Account

Returns The user.

lmi.scripts.account.**get_users_in_group**(*ns*, *group*)
Yields users in given group.

Parameters **group** (*LMIInstance or LMIInstanceName of LMI_Group.*) – The group to inspect.

Returns Generator of LMIInstances of LMI_Account.

`lmi.scripts.account.`**`is_in_group`**(*group*, *user*)
Return True if user is in group

Parameters

- **group** (*LMIInstance or LMIInstanceName of LMI_Group.*) – The group.
- **user** (*LMIInstance or LMIInstanceName of LMI_Account.*) – User to check.

`lmi.scripts.account.`**`list_groups`**(*ns*)
Yield all groups on the system.

Return type generator of LMIInstances.

`lmi.scripts.account.`**`list_users`**(*ns*)
Yield all users on the system.

Return type generator of LMIInstances.

`lmi.scripts.account.`**`remove_from_group`**(*ns*, *group*, *users*)
Remove users from a group.

Parameters

- **group** (*LMIInstance or LMIInstanceName of LMI_Group.*) – The group.
- **users** (*List (or generator) of LMIInstances or LMIInstanceNames of LMI_Account.*) – Users to remove.

Hardware Script python reference

Main interface functions wrapped with `lmi` commands are:

- `get_all_info()`
- `get_system_info()`
- `get_motherboard_info()`
- `get_cpu_info()`
- `get_memory_info()`
- `get_disks_info()`

All of these accept `NS` (namespace) object as the first argument, an instance of `lmi.shell.LMINamespace`.

Hardware Module API LMI hardware provider client library.

`lmi.scripts.hardware.`**`format_memory_size`**(*size*)
Returns formatted memory size.

Parameters **size** (*Number*) – Size in bytes

Returns Formatted size string.

Return type String

`lmi.scripts.hardware.`**`get_all_info`**(*ns*)

Returns Tabular data of all available info.

Return type List of tuples

lmi.scripts.hardware.**get_all_instances**(*ns*, *class_name*)
Returns all instances of instance_name.

Parameters **instance_name** (*String*) – Instance name

Returns List of instances of instance_name

Return type List of `lmi.shell.LMIInstance`

lmi.scripts.hardware.**get_colored_string**(*msg*, *color*)
Returns colored message with ANSI escape sequences for terminal.

Parameters

- **msg** (*String*) – Message to be colored.
- **color** (*Integer*) – Color of the message [GREEN_COLOR, YELLOW_COLOR, RED_COLOR].

Returns Colored message.

Return type String

lmi.scripts.hardware.**get_cpu_info**(*ns*)

Returns Tabular data of processor info.

Return type List of tuples

lmi.scripts.hardware.**get_disks_info**(*ns*)

Returns Tabular data of disk info.

Return type List of tuples

lmi.scripts.hardware.**get_hostname**(*ns*)

Returns Tabular data of system hostname.

Return type List of tuples

lmi.scripts.hardware.**get_memory_info**(*ns*)

Returns Tabular data of memory info.

Return type List of tuples

lmi.scripts.hardware.**get_motherboard_info**(*ns*)

Returns Tabular data of motherboard info.

Return type List of tuples

lmi.scripts.hardware.**get_single_instance**(*ns*, *class_name*)
Returns single instance of instance_name.

Parameters **instance_name** (*String*) – Instance name

Returns Instance of instance_name

Return type `lmi.shell.LMIInstance`

lmi.scripts.hardware.**get_system_info**(*ns*)

Returns Tabular data of system info, from the `LMI_Chassis` instance.

Return type List of tuples

Journald Script python reference

Journald Module API

lmi.scripts.journald.**list_messages**(*ns*, *reverse=False*, *tail=False*)
List messages from the journal.

Parameters

- **reverse** (*boolean*) – List messages from newest to oldest.
- **tail** (*boolean*) – List only the last 50 messages

lmi.scripts.journald.**log_message**(*ns*, *message*)
Logs a new message in the journal.

Parameters **message** (*string*) – A message to log.

lmi.scripts.journald.**watch**(*ns*)
Sets up a new indication listener and waits for events.

Locale Script python reference

LMI Locale Provider client library.

lmi.scripts.locale.**get_locale**(*ns*)
Get locale.

Return type LMIInstance/LMI_Locale

lmi.scripts.locale.**set_locale**(*ns*, *locales*, *values*)
Set given locale variables with new values.

Parameters

- **locales** (*list*) – List of locale variable names to be set.
- **values** (*list*) – List of new values for locale variables.

lmi.scripts.locale.**set_vc_keyboard**(*ns*, *keymap*, *keymap_toggle*, *convert*)
Set the key mapping on the virtual console.

Parameters

- **keymap** (*string*) – Requested keyboard mapping for the virtual console.
- **keymap_toggle** (*string*) – Requested toggle keyboard mapping for the virtual console.
- **convert** (*bool*) – Whether also X11 keyboard should be set to the nearest X11 keyboard setting for the chosen console keyboard setting.

lmi.scripts.locale.**set_x11_keymap**(*ns*, *layouts*, *model*, *variant*, *options*, *convert*)
Set the default key mapping of the X11 server.

Parameters

- **layouts** (*string*) – Requested X11 keyboard mappings.
- **model** (*string*) – Requested X11 keyboard model.
- **variant** (*string*) – Requested X11 keyboard variant.
- **options** (*string*) – Requested X11 keyboard options.
- **convert** (*bool*) – Whether also console keyboard should be set to the nearest console keyboard setting for the chosen X11 keyboard setting.

Logical File Script python reference

Logicalfile management functions.

lmi.scripts.logicalfile.logicalfile.**get_directory_instance**(*ns*, *directory*)
Retrieve LMIInstance of a directory.

Parameters **directory** (*string*) – Full path to the directory.

Return type LMIInstance

lmi.scripts.logicalfile.logicalfile.**get_directory_name_properties**(*ns*, *directory*)
Retrieve object path of a directory.

Parameters **directory** (*string*) – Full path to the directory.

Return type LMIInstanceName

lmi.scripts.logicalfile.logicalfile.**get_file_identification**(*file_instance*)
Retrieve file identification.

Parameters **file_instance** (*LMIInstance*) – The file's instance object.

Return type string

lmi.scripts.logicalfile.logicalfile.**lf_createdir**(*ns*, *directory*)
Create a directory.

The parent directory must exist.

Parameters **directory** (*string*) – Full path to the directory.

lmi.scripts.logicalfile.logicalfile.**lf_deletedir**(*ns*, *directory*)
Delete a directory.

The directory must be empty.

Parameters **directory** (*string*) – Full path to the directory.

lmi.scripts.logicalfile.logicalfile.**lf_list**(*ns*, *directory*, *depth=None*)
List all files in a directory.

If depth is positive, directory is walked recursively up to the given depth.

Parameters

- **directory** (*string*) – Full path to the directory.
- **depth** (*integer*) – Maximum depth to be recursed to.

lmi.scripts.logicalfile.logicalfile.**lf_show**(*ns*, *target*)
Show detailed information about the target.

Target can be either a file or a directory.

Parameters **target** (*string*) – Full path to the target.

lmi.scripts.logicalfile.logicalfile.**walk_cim_directory**(*directory*, *depth=0*)
Retrieve all files in a directory.

If depth is positive, directory is walked recursively up to the given depth. Files and directories are yielded as they are encountered. This function does not return, it is a generator.

Parameters

- **directory** (*string*) – Full path to the directory.

- **depth** (*integer*) – Maximum depth to be recursed to.

Networking Script python reference

Main interface functions wrapped with `lmi` commands are:

- `get_device_by_name()`
- `get_setting_by_caption()`
- `list_devices()`
- `list_settings()`
- `get_mac()`
- `get_ip_addresses()`
- `get_default_gateways()`
- `get_dns_servers()`
- `get_available_settings()`
- `get_active_settings()`
- `get_setting_type()`
- `get_setting_ip4_method()`
- `get_setting_ip6_method()`
- `get_sub_setting()`
- `get_applicable_devices()`
- `activate()`
- `deactivate()`
- `create_setting()`
- `delete_setting()`
- `add_ip_address()`
- `remove_ip_address()`
- `replace_ip_address()`

All of these accept `NS` object as the first argument. It is an instance of `lmi.shell.LMINamespace`.

Networking Module API LMI networking provider client library.

`lmi.scripts.networking.`**`SETTING_IP_METHOD_DHCP = 4`**
IP configuration obtained from DHCP server

`lmi.scripts.networking.`**`SETTING_IP_METHOD_DHCPv6 = 7`**
IP configuration obtained from DHCPv6 server

`lmi.scripts.networking.`**`SETTING_IP_METHOD_DISABLED = 0`**
Disabled IP configuration

`lmi.scripts.networking.`**`SETTING_IP_METHOD_STATELESS = 9`**
Stateless IPv6 configuration

lmi.scripts.networking.**SETTING_IP_METHOD_STATIC = 3**
Static IP address configuration

lmi.scripts.networking.**SETTING_TYPE_BOND_MASTER = 4**
Configuration for bond master

lmi.scripts.networking.**SETTING_TYPE_BOND_SLAVE = 40**
Configuration for bond slave

lmi.scripts.networking.**SETTING_TYPE_BRIDGE_MASTER = 5**
Configuration for bridge master

lmi.scripts.networking.**SETTING_TYPE_BRIDGE_SLAVE = 50**
Configuration for bridge slave

lmi.scripts.networking.**SETTING_TYPE_ETHERNET = 1**
Configuration for ethernet

lmi.scripts.networking.**SETTING_TYPE_UNKNOWN = 0**
Unknown type of setting

lmi.scripts.networking.**activate**(*ns*, *setting*, *device=None*)
Activate network setting on given device

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – Setting to be activated.
- **device** (*LMI_IPNetworkConnection or None*) – Device to activate the setting on or None for autodetection

lmi.scripts.networking.**add_dns_server**(*ns*, *setting*, *address*)
Add a dns server to the given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.

lmi.scripts.networking.**add_ip_address**(*ns*, *setting*, *address*, *prefix*, *gateway=None*)
Add an IP address to the given static setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.
- **prefix** (*int*) – network prefix.
- **gateway** (*str or None*) – default gateway or None

lmi.scripts.networking.**add_static_route**(*ns*, *setting*, *address*, *prefix*, *metric=None*, *next_hop=None*)
Add a static route to the given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.
- **prefix** (*int*) – network prefix.
- **metric** – metric for the route or None
- **next_hop** (*str or None*) – IPv4 or IPv6 address for the next hop of the route or None

lmi.scripts.networking.**create_setting**(*ns*, *caption*, *device*, *type*, *ipv4method*, *ipv6method*)
Create new network setting.

Parameters

- **caption** (*str*) – Caption for the new setting.
- **device** (*LMI_IPNetworkConnection*) – Device for which the setting will be.
- **type** (*SETTING_TYPE_* constant*) – Type of the setting.
- **ipv4method** (*SETTING_IP_METHOD_* constant*) – Method for obtaining IPv4 address.
- **ipv4method** – Method for obtaining IPv6 address.

lmi.scripts.networking.**deactivate**(*ns*, *setting*, *device=None*)
Deactivate network setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – Setting to deactivate.
- **device** (*LMI_IPNetworkConnection or None*) – Device to deactivate the setting on

lmi.scripts.networking.**delete_setting**(*ns*, *setting*)
Delete existing network setting.

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting.

lmi.scripts.networking.**enslave**(*ns*, *master_setting*, *device*)
Create slave setting of the master_setting with given device.

Parameters

- **master_setting** (*LMI_IPAssignmentSettingData*) – Master setting to use
- **device** (*LMI_IPNetworkConnection*) – Device to enslave

lmi.scripts.networking.**get_active_settings**(*ns*, *device*)
Get a list of settings that are currently active on the device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns Settings that are active on the device

Return type list of LMI_IPAssignmentSettingData

lmi.scripts.networking.**get_applicable_devices**(*ns*, *setting*)
Get list of network devices that this setting can be applied to.

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns devices that the setting can be applied to

Return type list of LMI_IPNetworkConnection

lmi.scripts.networking.**get_available_settings**(*ns*, *device*)
Get a list of settings that can be applied to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns Settings applicable to the device

Return type list of LMI_IPAssignmentSettingData

lmi.scripts.networking.**get_default_gateways**(*ns*, *device*)
Get a list of default gateways assigned to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns Default gateways assigned to the device

Return type list of str

lmi.scripts.networking.**get_device_by_name**(*ns*, *device_name*)
Get instance of LMI_IPNetworkConnection class by the device name.

Parameters **device_name** (*str*) – Name of the device.

Returns LMI_IPNetworkConnection representing the device.

Return type `LMI_IPNetworkConnection` or `None` if not found

lmi.scripts.networking.**get_dns_servers**(*ns*, *device*)
Get a list of DNS servers assigned to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns DNS servers assigned to the device

Return type list of str

lmi.scripts.networking.**get_ip_addresses**(*ns*, *device*)
Get a list of IP addresses assigned to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns IP addresses with subnet masks (IPv4) or prefixes (IPv6) that is assigned to the device.

Return type list of tuple (IPAddress, SubnetMask/Prefix)

lmi.scripts.networking.**get_ipv4_addresses**(*ns*, *device*)
Get a list of IPv4 addresses assigned to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns IPv4 addresses with subnet masks that is assigned to the device.

Return type list of tuple (IPAddress, SubnetMask)

lmi.scripts.networking.**get_ipv6_addresses**(*ns*, *device*)
Get a list of IPv6 addresses assigned to given device

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns IPv6 addresses with prefixes that is assigned to the device.

Return type list of tuple (IPAddress, Prefix)

lmi.scripts.networking.**get_mac**(*ns*, *device*)
Get a MAC address for given device.

Parameters **device** (*LMI_IPNetworkConnection*) – network device

Returns MAC address of given device or 00:00:00:00:00:00 when no MAC is found.

Return type str

lmi.scripts.networking.**get_setting_by_caption**(*ns*, *caption*)
Get instance of LMI_IPAssignmentSettingData class by the caption.

Parameters **caption** (*str*) – Caption of the setting.

Returns LMI_IPAssignmentSettingData representing the setting.

Return type `LMI_IPAssignmentSettingData` or `None` if not found

lmi.scripts.networking.**get_setting_ip4_method**(*ns*, *setting*)
Get method of obtaining IPv4 configuration on given setting

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns IPv4 method

Return type SETTING_IP_METHOD_* constant

lmi.scripts.networking.**get_setting_ip6_method**(*ns*, *setting*)
Get method of obtaining IPv6 configuration on given setting

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns IPv6 method

Return type SETTING_IP_METHOD_* constant

lmi.scripts.networking.**get_setting_type**(*ns*, *setting*)
Get type of the setting

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns type of setting

Return type SETTING_TYPE_* constant

lmi.scripts.networking.**get_static_routes**(*ns*, *setting*)
Return list of static routes for given setting

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns list of static routes

Return type list of LMI_IPRouteSettingData

lmi.scripts.networking.**get_sub_setting**(*ns*, *setting*)
Get list of detailed configuration setting for each part of the setting.

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Returns detailed setting

Return type list of LMI_IPAssignmentSettingData subclasses

lmi.scripts.networking.**is_setting_active**(*ns*, *setting*)
Return true if the setting is currently active

Parameters **setting** (*LMI_IPAssignmentSettingData*) – network setting

Retval True setting is currently active

Retval False setting is not currently active

Return type bool

lmi.scripts.networking.**list_devices**(*ns*, *device_names=None*)
Get a list of network devices.

Parameters **device_name** (*list of str*) – List of device names that will be used as filter for devices.

Returns List of instances of LMI_IPNetworkConnection

Return type list of LMI_IPNetworkConnection

lmi.scripts.networking.**list_settings**(*ns*, *captions=None*)
Get a list of network settings.

Parameters **captions** (*list of str*) – List of setting captions that will be used as filter for settings.

Returns Settings that matches given captions

Return type list of LMI_IPAssignmentSettingData

lmi.scripts.networking.**remove_dns_server**(*ns*, *setting*, *address*)
Remove dns server from given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.

lmi.scripts.networking.**remove_ip_address**(*ns*, *setting*, *address*)
Remove the IP address from given static setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.

lmi.scripts.networking.**remove_static_route**(*ns*, *setting*, *address*)
Remove static route to the given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.

lmi.scripts.networking.**replace_dns_server**(*ns*, *setting*, *address*)
Remove all dns servers and add given dns server to the given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.

lmi.scripts.networking.**replace_ip_address**(*ns*, *setting*, *address*, *prefix*, *gateway=None*)
Remove all IP addresses from given static setting and add new IP address.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.
- **prefix** (*int*) – network prefix.
- **gateway** (*str or None*) – default gateway or None

lmi.scripts.networking.**replace_static_route**(*ns*, *setting*, *address*, *prefix*, *metric=None*, *next_hop=None*)
Remove all static routes and add given static route to the given setting.

Parameters

- **setting** (*LMI_IPAssignmentSettingData*) – network setting.
- **address** (*str*) – IPv4 or IPv6 address.
- **prefix** (*int*) – network prefix.
- **metric** – metric for the route or None
- **next_hop** (*str or None*) – IPv4 or IPv6 address for the next hop of the route or None

Power Management Script python reference

Main interface functions wrapped with `lmi` commands are:

- `list_power_states()`
- `switch_power_state()`

All of these accept `NS` object as the first argument. It is an instance of `lmi.shell.LMINamespace`.

Power Management Module API LMI power management provider client library.

`lmi.scripts.powermanagement.`**`POWER_STATE_HIBERNATE = 7`**
Hibernate the system.

`lmi.scripts.powermanagement.`**`POWER_STATE_POWEROFF = 12`**
Turn off the system.

`lmi.scripts.powermanagement.`**`POWER_STATE_POWEROFF_FORCE = 8`**
Turn off the system without shutting down services first.

`lmi.scripts.powermanagement.`**`POWER_STATE_REBOOT = 15`**
Reboot the system.

`lmi.scripts.powermanagement.`**`POWER_STATE_REBOOT_FORCE = 5`**
Reboot the system without shutting down services first.

`lmi.scripts.powermanagement.`**`POWER_STATE_SUSPEND = 4`**
Suspend the system.

`lmi.scripts.powermanagement.`**`list_power_states`**(*ns*)
Get list of available power states.

Returns list of power states

Return type list of POWER_STATE_* constants

`lmi.scripts.powermanagement.`**`switch_power_state`**(*ns*, *state*)
Switch system power state.

Parameters **state** (*POWER_STATE_* constant*) – Requested power state.

Realmd Script python reference

Realmd Module API LMI realmd provider client library.

`lmi.scripts.realmd.`**`join`**(*ns*, *domain*, *user*, *_password=None*)
Join the domain.

Parameters

- **domain** (*string*) – The domain to be joined.
- **user** (*string*) – User name to authenticate with
- **password** (*string*) – The authentication password

`lmi.scripts.realmd.`**`leave`**(*ns*, *domain*, *user*, *_password=None*)
Leave the domain.

Parameters

- **domain** (*string*) – The domain to be left.

- **user** (*string*) – User name to authenticate with
- **password** (*string*) – The authentication password

`lmi.scripts.realmd.`**`show`**(*ns*)
Show the joined domain.

Service Script python reference

LMI service provider client library.

`lmi.scripts.service.`**`enable_service`**(*ns*, *service*, *enable=True*)
Enable or disable service.

Parameters

- **service** (string or `lmi.shell.LMIInstanceName`) – Service name or instance.
- **enable** (*boolean*) – Whether the service should be enabled or disabled. Enabled service is started on system boot.

`lmi.scripts.service.`**`get_enabled_string`**(*ns*, *service*)
Return human friendly string for enabled state.

Parameters **service** – Either a service instance of its name.

Returns Status description. One of: `{ Yes, No, Static }`. Where *Static* represents a service that can not be enabled or disabled, and are run only if something depends on them. It lacks `[Install]` section.

Return type string

`lmi.scripts.service.`**`get_service`**(*ns*, *service*)
Return `lmi.shell.LMIInstance` object matching the given service name.

Parameters **service** (*string*) – Service name.

`lmi.scripts.service.`**`get_status_string`**(*ns*, *service*)
Return human friendly status description.

Parameters **service** – Either a service instance or its name.

Returns Status description. One of `{ OK, Running, Stopped - OK, Stopped - Error }`.

Return type string

`lmi.scripts.service.`**`invoke_on_service`**(*ns*, *method*, *service*, *description*)
Invoke parameter-less method on given service.

Parameters

- **method** (*string*) – Name of method of `LMI_Service` to invoke.
- **service** (string or `lmi.shell.LMIInstanceName`) – Name of service or an instance to operate upon.
- **description** (*string*) – Description of what has been done with service. This is used just for logging.

Returns Success flag.

Return type boolean

lmi.scripts.service.**list_services**(*ns*, *kind='enabled'*)
List services. Yields service instances.

Parameters **kind** (*string*) – What kind of services to list. Possible options are:

- 'enabled' - list only enabled services
- 'disabled' - list only disabled services
- 'all' - list all services

Returns Instances of LMI_Service.

Return type generator over lmi.shell.LMIInstance.

lmi.scripts.service.**reload_service**(*ns*, *service*, *force=False*, *just_try=False*)
Reload service.

Parameters

- **service** (string or lmi.shell.LMIInstanceName) – Service name or instance.
- **force** (*boolean*) – Whether the service should be restarted if the reload can no be done.
- **just_try** (*boolean*) – This applies only when force is True. If True, only the the running service will be restarted. Nothing is done for stopped service.

lmi.scripts.service.**restart_service**(*ns*, *service*, *just_try=False*)
Restart service.

Parameters

- **service** (string or lmi.shell.LMIInstanceName) – Service name or instance.
- **just_try** (*boolean*) – When False, the service will be started even if it is not running. Otherwise only running service will be restarted.

lmi.scripts.service.**start_service**(*ns*, *service*)
Start service.

Parameters **service** (string or lmi.shell.LMIInstanceName) – Service name.

lmi.scripts.service.**stop_service**(*ns*, *service*)
Stop service.

Parameters **service** (string or lmi.shell.LMIInstanceName) – Service name or instance.

Software Script python reference

LMI software provider client library.

Package specification Referred to as PKG_SPEC. Is a string identifying set of packages. It constitutes at least of package name. Each additional detail narrows the the possible set of matchin packages. The most complete specifications are nevra and envra.

Follows the list of all possible specifications:

- <name>
- <name>.<arch>
- <name>-<version>-<release>.<arch> (nvra)
- <name>-<epoch>:<version>-<release>.<arch> (nevra)

- `<epoch>:<name>-<version>-<release>.<arch>` (envra)

Regular expressions These may be used check, whether the given PKG_SPEC (package specification) is valid and allows to get all the interesting parts out of it.

`lmi.scripts.software.`**`RE_NA`**
Regular expression matching package specified as `<name>.<arch>`.

`lmi.scripts.software.`**`RE_NEVRA`**
Regular expression matching package specified as:

```
<name>-<epoch>:<version>-<release>.<arch>
```

The epoch part is optional. So it can be used also to match `nvra` string.

`lmi.scripts.software.`**`RE_ENVRA`**
Regular expression matching package specified as:

```
<epoch>:<name>-<version>-<release>.<arch>
```

Functions

`lmi.scripts.software.`**`FILE_TYPES`** **= ('Unknown', 'File', 'Directory', 'Symlink', 'FIFO', 'Character Device', 'Block De**
Array of file type names.

`lmi.scripts.software.`**`find_package`**(*ns*, *allow_duplicates=False*, *exact_match=True*, ***kwargs*)
Yields just a limited set of packages matching particular filter. Keyword arguments are used to specify this filter, which can contain following keys:

`name` **:** Package name.

`epoch` **:** package's epoch

`version` **:** version of package

`release` **:** release of package

`arch` **:** requested architecture of package

`nevra` **:** string containing all previous keys in following notation:

```
<name>-<epoch>:<version>-<release>.<arch>
```

`envra` **:** similar to nevra, the notation is different:

```
<epoch>:<name>-<version>-<release>.<arch>
```

`repoid` **:** repository identification string, where package must be available

`pkg_spec` **:** Package specification string. See *Package specification*.

Parameters

- **allow_duplicates** (*boolean*) – Whether the output shall contain multiple versions of the same packages identified with `<name>.<architecture>`.
- **exact_match** (*boolean*) – Whether the `name` key shall be tested for exact match. If `False` it will be tested for inclusion.

Returns Instance names of `LMI_SoftwareIdentity`.

Return type generator over `lmi.shell.LmiInstanceName`

lmi.scripts.software.**get_package_nevra**(*package*)
Get a nevra from an instance of LMI_SoftwareIdentity.

Parameters **package** (lmi.shell.LMIInstance or lmi.shell.LMIInstanceName) – Instance or instance name of LMI_SoftwareIdentity representing package to install.

Returns Nevra string of particular package.

Return type string

lmi.scripts.software.**get_repository**(*ns*, *repoid*)
Return an instance of repository identified by its identification string.

Parameters **repoid** (*string*) – Identification string of repository.

Returns Instance of LMI_SoftwareIdentityResource.

Return type lmi.shell.LMIInstance

lmi.scripts.software.**install_from_uri**(*ns*, *uri*, *force=False*, *update=False*)
Install package from *URI* on remote system.

Parameters

- **uri** (*string*) – Identifier of *RPM* package available via http, https, or ftp service.
- **force** (*boolean*) – Whether the installation shall be done even if installing the same (reinstalling) or older version than already installed.
- **update** (*boolean*) – Whether this is an update. Update fails if package is not already installed on system.

lmi.scripts.software.**install_package**(*ns*, *package*, *force=False*, *update=False*)
Install package on system.

Parameters

- **package** (lmi.shell.LMIInstance or lmi.shell.LMIInstanceName) – Instance or instance name of LMI_SoftwareIdentity representing package to install.
- **force** (*boolean*) – Whether the installation shall be done even if installing the same (reinstalling) or older version than already installed.
- **update** (*boolean*) – Whether this is an update. Update fails if package is not already installed on system.

Returns Software identity installed on remote system. It's an instance LMI_SoftwareIdentity.

Return type lmi.shell.LMIInstance

lmi.scripts.software.**list_available_packages**(*ns*, *allow_installed=False*, *allow_duplicates=False*, *repoid=None*)
Yields instances of LMI_SoftwareIdentity representing available packages.

Parameters

- **allow_installed** (*boolean*) – Whether to include available packages that are installed.
- **allow_duplicates** (*boolean*) – Whether to include duplicates packages (those having same name and architecture). Otherwise only the newest packages available for each (name, architecture) pair will be contained in result.
- **repoid** (*string*) – Repository identification string. This will filter available packages just for those provided by this repository.

Return type generator

lmi.scripts.software.**list_installed_packages**(*ns*)
Yields instances of LMI_SoftwareIdentity representing installed packages.

Return type generator

lmi.scripts.software.**list_package_files**(*ns*, *package*, *file_type=None*)
Get a list of files belonging to particular installed *RPM* package. Yields instances of LMI_SoftwareIdentityFileCheck.

Parameters

- **package** (lmi.shell.LMIInstance or lmi.shell.LMIInstanceName) – Instance or instance name of LMI_SoftwareIdentity.
- **file_type** (string, integer or None) – Either an index to FILE_TYPES array or one of: { "all", "file", "directory", "symlink", "fifo", "device" }.

Returns Instances of LMI_SoftwareIdentityFileCheck.

Return type generator over lmi.shell.LMIInstance

lmi.scripts.software.**list_repositories**(*ns*, *enabled=True*)
Yields instances of LMI_SoftwareIdentityResource representing software repositories.

Parameters **enabled** (boolean or None) – Whether to list only enabled repositories. If False only disabled repositories shall be listed. If None, all repositories shall be listed.

Returns Instances of LMI_SoftwareIdentityResource

Return type generator over lmi.shell.LMIInstance

lmi.scripts.software.**pkg_spec_to_filter**(*pkg_spec*)
Converts package specification to a set of keys, that can be used to query package properties.

Parameters **pkg_spec** (*string*) – Package specification (see *Package specification*). Only keys given in this string will appear in resulting dictionary.

Returns Dictionary with possible keys being a subset of following: {'name', 'epoch', 'version', 'release', 'arch'}. Values are non-empty parts of pkg_spec string.

Return type dictionary

lmi.scripts.software.**remove_package**(*ns*, *package*)
Uninstall given package from system.

Raises LmiFailed` will be raised on failure.

Parameters **package** (lmi.shell.LMIInstance or lmi.shell.LMIInstanceName) – Instance or instance name of LMI_SoftwareIdentity representing package to remove.

lmi.scripts.software.**render_failed_flags**(*failed_flags*)
Make one liner string representing failed flags list of file that did not pass the verification.

Parameters **failed_flags** (*list*) – Value of FailedFlags property of some LMI_SoftwareIdentityFileCheck.

Returns Verification string with format matching the output of rpm -V command.

Return type string

lmi.scripts.software.**set_repository_enabled**(*ns*, *repository*, *enable=True*)
Enable or disable repository.

Parameters

- **repository** (`lmi.shell.LMIInstance` or `lmi.shell.LMIInstanceName`) – Instance of `LMI_SoftwareIdentityResource`.
- **enable** (*boolean*) – New value of `EnabledState` property.

Returns Previous value of repository's `EnabledState`.

Return type boolean

`lmi.scripts.software.`**`verify_package`**(*ns*, *package*)

Returns the instances of `LMI_SoftwareIdentityFileCheck` representing files, that did not pass the verification.

Parameters **package** (`lmi.shell.LMIInstance` or `lmi.shell.LMIInstanceName`) – Instance or instance name of `LMI_SoftwareIdentity` representing package to verify.

Returns List of instances of `LMI_SoftwareIdentityFileCheck` with non-empty `FailedFlags` property.

Return type list

SSSD Script python reference

LMI SSSD provider client library.

This set of functions can list and manage SSSD's responders and domains.

`lmi.scripts.sssd.`**`debug_level`**(*level*)

Return hexadecimal representation of debug level.

Parameters **level** (*int*) – Debug level.

Return type string

Storage Script python reference

OpenLMI Storage Scripts module is a standard python module, which provides high-level functions to manage storage on remote hosts with installed *OpenLMI-Storage provider*.

It is built on top of *LMIShell*, however only very little knowledge about the *LMIShell* and CIM model are required.

All *LMI metacommands* are implemented using this python module. I.e. everything that LMI metacommand can do with storage you can do also in python using this module, which makes it a good start for LMI scripting.

Example:

```
# Connect to a remote system using lmishell
import lmi.shell
conn = lmi.shell.connect("remote.host.org", "root", "opensesame")

# Find a namespace we want to operate on, root/cimv2 is the most used.
ns = conn.root.cimv2

# Now use lmi.scripts.storage functions.

# For example, let's partition /dev/vda disk
from lmi.scripts.storage import partition
partition.create_partition_table(ns, 'vda', partition.PARTITION_TABLE_TYPE_GPT)
# Create one large partition on it
new_partition = partition.create_partition(ns, 'vda')
```

```
# Create a volume group with the partition
from lmi.scripts.storage import lvm
new_vg = lvm.create_vg(ns, ['vda1'], 'my_vg')
print 'New VG name: ', new_vg.Name

# Create a 100 MB logical volume on the volume group
MEGABYTE = 1024*1024
new_lv = lvm.create_lv(ns, new_vg, 'my_lv', 100 * MEGABYTE)

# Format the LV
from lmi.scripts.storage import fs
fs.create_fs(ns, [new_lv], 'xfs')
```

It is important to note that most of the module functions accept both `string` or `LMIInstance` as parameters. For example, these two lines would have the same effect in the example above:

```
new_lv = lvm.create_lv(ns, 'my_vg', 100*MEGABYTE)

new_lv = lvm.create_lv(ns, new_vg,  100*MEGABYTE)
```

The first one use plain `string` as a volume group name, while the other uses LMIShell's `LMIInstance` previously returned from `lvm.create_vg()`.

Common functions Common storage functionality.

lmi.scripts.storage.common.**escape_cql**(*s*)
Escape potentially unsafe string for CQL.

It is generally not possible to do anything really harmful in CQL (there is no DELETE nor DROP TABLE), but just to be nice, all strings passed to CQL should escape backslash '' and double quote '"'.

Parameters **s** (*string*) – String to escape.

Return type string

lmi.scripts.storage.common.**get_children**(*ns*, *obj*, *deep=False*)
Return list of all children of given LMIInstance.

For example:

- If `obj` is LMIInstance/LMI_VGStoragePool (=Volume Group), it returns all its Logical Volumes (=LMIInstance/LMI_LVStorageExtent).
- If `obj` is LMIInstance/LMI_StorageExtent of a disk, it returns all its partitions (=LMIInstance/CIM_GenericDiskPartition).
- If `obj` is LMIInstance/LMI_DiskPartition and the partition is Physical Volume of a Volume Group,, it returns the pool (LMIInstance/LMI_VGStoragePool).

Parameters

- **obj** (*LMIInstance/CIM_StorageExtent or LMIInstance/LMI_VGStoragePool or string*) – Object to find children of.
- **deep** (*Boolean*) – Whether all children of the object should be returned or only immediate ones.

lmi.scripts.storage.common.**get_devices**(*ns*, *devices=None*)
Returns list of block devices. If no devices are given, all block devices on the system are returned.

This functions just converts list of strings to list of appropriate LMIInstances.

Parameters **devices** (*list of LMIInstance/CIM_StorageExtent or list of strings*) – Devices to list.

Return type list of LMIInstance/CIM_StorageExtent.

lmi.scripts.storage.common.**get_parents**(*ns*, *obj*, *deep=False*)
Return list of all parents of given LMIInstance.

For example:

- If `obj` is LMIInstance/LMI_LVStorageExtent (=Logical Volume), it returns LMIInstance/LMI_VGStoragePool (=Volume Group).
- If `obj` is LMIInstance/LMI_VGStoragePool (=Volume Group), it returns all its Physical Volumes (=LMIInstance/CIM_StorageExtent).

Parameters

- **obj** (*LMIInstance/CIM_StorageExtent or LMIInstance/LMI_VGStoragePool or string*) – Object to find parents of.
- **deep** (*Boolean*) – Whether all parents of the object should be returned or only immediate ones.

lmi.scripts.storage.common.**size2str**(*size*, *human_friendly*)
Convert size (in bytes) to string.

Parameters

- **size** (*int*) – Size of something in bytes.
- **human_friendly** (*bool*) – If True, the returned string is returned in human-friendly units (KB, MB, ...).

Return type string

lmi.scripts.storage.common.**str2device**(*ns*, *device*)
Convert string with name of device to LMIInstance of the device. If LMIInstance is provided, nothing is done and the instance is just returned. If string is given, appropriate LMIInstance is looked up and returned. This functions throws an error when the device cannot be found.

The main purpose of this function is to convert parameters in functions, where both string and LMIInstance is allowed.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string with name of device*) – Device to convert.

Return type LMIInstance/CIM_StorageExtent

lmi.scripts.storage.common.**str2obj**(*ns*, *obj*)
Convert string with name of device or volume group to LMIInstance of the device or the volume group.

If LMIInstance is provided, nothing is done and the instance is just returned. If string is given, appropriate LMIInstance is looked up and returned. This functions throws an error when the device or volume group cannot be found.

The main purpose of this function is to convert parameters in functions, where both string and LMIInstance is allowed.

Parameters **obj** (*LMIInstance/CIM_StorageExtent or LMIInstance/LMI_VGStoragePool or string with name of device or pool*) – Object to convert.

Return type LMIInstance/CIM_StorageExtent or LMIInstance/LMI_VGStoragePool

lmi.scripts.storage.common.**str2size**(*size*, *additional_unit_size=None*, *additional_unit_suffix=None*)

Convert string from human-friendly size to bytes. The string is expected to be integer number, optionally with on of these suffixes:

- k, K - kilobytes, 1024 bytes,
- m, M - megabytes, 1024 * 1024 bytes,
- g, G - gigabytes, 1024 * 1024 * 1024 bytes,
- t, T - terabytes, 1024 * 1024 * 1024 * 1024 bytes,

Parameters

- **size** (*string*) – The size to convert.
- **additional_unit_size** (*int*) – Additional unit size for additional_unit_suffix, e.g. 4 * 1024*1024 for extent size.
- **additional_unit_suffix** (*string*) – Additional suffix, e.g. 'E' for extents.

Return type int

lmi.scripts.storage.common.**str2vg**(*ns*, *vg*)

Convert string with name of volume group to LMIInstance of the LMI_VGStoragePool.

If LMIInstance is provided, nothing is done and the instance is just returned. If string is provided, appropriate LMIInstance is looked up and returned.

This functions throws an error when the device cannot be found.

The main purpose of this function is to convert parameters in functions, where both string and LMIInstance is allowed.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – VG to retrieve.

Return type LMIInstance/LMI_VGStoragePool

Partitioning Partition management functions.

lmi.scripts.storage.partition.**create_partition**(*ns*, *device*, *size=None*, *partition_type=None*)

Create new partition on given device.

Parameters

- **device** (*LMIInstance/CIM_StorageExtent or string*) – Device which should be partitioned.
- **size** (*int*) – Size of the device, in blocks. See device's BlockSize to get it. If no size is provided, the largest possible partition is created.
- **partition_type** (*int*) – Requested partition type. See PARTITION_TYPE_xxx variables. If no type is given, extended partition will be automatically created as 4th partition on MS-DOS style partition table with a logical partition with requested size on it.

Return type LMIInstance/CIM_GenericDiskPartition.

lmi.scripts.storage.partition.**create_partition_table**(*ns*, *device*, *table_type*)

Create new partition table on a device. The device must be empty, i.e. must not have any partitions on it.

Parameters

- **device** (*LMIInstance/CIM_StorageExtent*) – Device where the partition table should be created.

- **table_type** (*int*) – Requested partition table type. See PARTITION_TABLE_TYPE_xxx variables.

lmi.scripts.storage.partition.**delete_partition**(*ns*, *partition*)
Remove given partition

Parameters **partition** (*LMIInstance/CIM_GenericDiskPartition*) – Partition to delete.

lmi.scripts.storage.partition.**get_disk_partition_table**(*ns*, *device*)
Returns LMI_DiskPartitionTableCapabilities representing partition table on given disk.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device which should be examined.

Return type LMIInstance/LMI_DiskPartitionConfigurationCapabilities.

lmi.scripts.storage.partition.**get_disk_partitions**(*ns*, *disk*)
Return list of partitions on the device (not necessarily disk).

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device which should be partitioned.

Return type List of LMIInstance/CIM_GenericDiskPartition.

lmi.scripts.storage.partition.**get_largest_partition_size**(*ns*, *device*)
Returns size of the largest free region (in blocks), which can accommodate a partition on given device. There must be partition table present on this device.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device which should be examined.

Return type int

lmi.scripts.storage.partition.**get_partition_disk**(*ns*, *partition*)
Return a device on which is located the given partition.

Parameters **partition** (*LMIInstance/CIM_GenericDiskPartition or string*) – Partition to examine.

Return type LMIInstance/CIM_StorageExtent.

lmi.scripts.storage.partition.**get_partition_tables**(*ns*, *devices=None*)
Returns list of partition tables on given devices. If no devices are given, all partitions on all devices are returned.

Parameters **devices** (*list of LMIInstance/CIM_StorageExtent or list of strings*) – Devices to list partition tables on.

Return type List of tuples (LMIInstance/CIM_StorageExtent, LMIInstance/LMI_DiskPartitionConfigurationCapabilities).

lmi.scripts.storage.partition.**get_partitions**(*ns*, *devices=None*)
Retrieve list of partitions on given devices. If no devices are given, all partitions on all devices are returned.

Parameters **devices** (*List of LMIInstance/CIM_StorageExtent or list of string*) – Devices to list partitions on.

Return type List of LMIInstance/CIM_GenericPartition.

LUKS Management LUKS management functions.

lmi.scripts.storage.luks.**add_luks_passphrase**(*ns*, *fmt*, *passphrase*, *new_passphrase*)
Adds new password to LUKS format. Each format can have up to 8 separate passwords and any of them can be used to open(decrypt) the format.

Any existing passphrase must be provided to add a new one. This proves the caller is authorized to add new passphrase (because it already knows one) and also this 'old' passphrase is used to retrieve encryption keys. This 'old' passphrase is not removed nor replaced when adding new passphrase!

Parameters

- **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to modify.
- **passphrase** (*string*) – Existing LUKS passphrase.
- **new_passphrase** (*string*) – New passphrase to add to the format.

lmi.scripts.storage.luks.**close_luks**(*ns*, *fmt*)
Closes clear-text block device previously opened by open_luks().

Parameters **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to close.

lmi.scripts.storage.luks.**create_luks**(*ns*, *device*, *passphrase*)
Format given device with LUKS encryption format. All data on the device will be deleted! Encryption key and algorithm will be chosen automatically.

Parameters

- **device** (*LMIInstance/CIM_StorageExtent or string*) – Device to format with LUKS data
- **passphrase** (*string*) – Password to open the encrypted data. This is not the encryption key.

Return type LMIInstance/LMI_EncryptionFormat

lmi.scripts.storage.luks.**delete_luks_passphrase**(*ns*, *fmt*, *passphrase*)
Delete passphrase from LUKS format.

Parameters

- **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to modify.
- **passphrase** (*string*) – The passphrase to remove

lmi.scripts.storage.luks.**get_luks_device**(*ns*, *fmt*)
Return clear-text device for given LUKS format. The format must be already opened by open_luks().

Parameters **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to inspect.

Return type LMIInstance/LMI_LUKSStorageExtent

Returns Block device with clear-text data or None, if the LUKS format is not open.

lmi.scripts.storage.luks.**get_luks_list**(*ns*)
Retrieve list of all encrypted devices.

Return type list of LMIInstance/LMI_EncryptionFormat.

lmi.scripts.storage.luks.**get_passphrase_count**(*ns*, *fmt*)
Each LUKS format can have up to 8 passphrases. Any of these passphrases can be used to decrypt the format and create clear-text device.

This function returns number of passphrases in given LUKS format.

Parameters **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to inspect.

Return type int

Returns Number of used passphrases.

lmi.scripts.storage.luks.**open_luks**(*ns*, *fmt*, *name*, *passphrase*)
Open encrypted LUKS format and expose it as a clear-text block device.

Parameters

- **fmt** (*LMIInstance/LMI_EncryptionFormat or string*) – The LUKS format to open.
- **name** (*string*) – Requested name of the clear-text block device. It will be available as /dev/mapper/<name>.
- **passphrase** (*string*) – Password to open the encrypted data.

Return type LMIInstance/LMI_LUKSStorageExtent

Returns The block device with clear-text data.

Logical Volume Management LVM management functions.

lmi.scripts.storage.lvm.**create_lv**(*ns*, *vg*, *name*, *size*)
Create new Logical Volume on given Volume Group.

Parameters

- **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to allocate the volume from.
- **name** (*string*) – Name of the logical volume.
- **size** (*int*) – Size of the logical volume in bytes.

Return type LMIInstance/LMI_LVStorageExtent

lmi.scripts.storage.lvm.**create_vg**(*ns*, *devices*, *name*, *extent_size=None*)
Create new Volume Group from given devices.

Parameters

- **device** – Devices to add to the Volume Group.
- **name** (*string*) – Name of the Volume gGoup.
- **extent_size** (*int*) – Extent size in bytes.

Return type LMIInstance/LMI_VGStoragePool

lmi.scripts.storage.lvm.**delete_lv**(*ns*, *lv*)
Destroy given Logical Volume.

Parameters **lv** (*LMIInstance/LMI_LVStorageExtent or string*) – Logical Volume to destroy.

lmi.scripts.storage.lvm.**delete_vg**(*ns*, *vg*)
Destroy given Volume Group.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to delete.

lmi.scripts.storage.lvm.**get_lv_vg**(*ns*, *lv*)
Return Volume Group of given Logical Volume.

Parameters **lv** (*LMIInstance/LMI_LVStorageExtent or string*) – Logical Volume to examine.

Return type LMIInstance/LMI_VGStoragePool

lmi.scripts.storage.lvm.**get_lvs**(*ns*, *vgs=None*)
Retrieve list of all logical volumes allocated from given volume groups.

If no volume groups are provided, all logical volumes on the system are returned.

Parameters **vgs** (*list of LMIInstance/LMI_VGStoragePool or list of strings*) – Volume Groups to examine.

Return type list of LMIInstance/LMI_LVStorageExtent.

lmi.scripts.storage.lvm.**get_tp_vgs**(*ns*, *tp*)
Return Volume Groups of given Thin Pool.

Alias for get_vg_tps.

lmi.scripts.storage.lvm.**get_tps**(*ns*)
Retrieve list of all thin pools on the system.

Return type list of LMIInstance/LMI_VGStoragePool

lmi.scripts.storage.lvm.**get_vg_lvs**(*ns*, *vg*)
Return list of Logical Volumes on given Volume Group.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to examine.

Return type list of LMIInstance/LMI_LVStorageExtent

lmi.scripts.storage.lvm.**get_vg_pvs**(*ns*, *vg*)
Return Physical Volumes of given Volume Group.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to examine.

Return type list of LMIInstance/CIM_StorageExtent

lmi.scripts.storage.lvm.**get_vg_tps**(*ns*, *vg*)
Return Thin Pools of given Volume Group.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to examine.

Return type list of LMIInstance/CIM_StoragePool

lmi.scripts.storage.lvm.**get_vgs**(*ns*)
Retrieve list of all volume groups on the system.

Return type list of LMIInstance/LMI_VGStoragePool

lmi.scripts.storage.lvm.**modify_vg**(*ns*, *vg*, *add_pvs=None*, *remove_pvs=None*)
Modify given Volume Group.

Add 'add_pvs' devices as Physical Volumes of the group. Remove 'remove_pvs' devices from the Volume Group.

Parameters

- **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to delete.
- **add_pvs** (*List of LMIInstances/LMI_VGStoragePools or strings*) – List of new devices to be added as Physical Volumes of the VG.
- **remove_pvs** (*List of LMIInstances/LMI_VGStoragePools or strings*) – List of Physical Volume to be removed from the VG.

MD RAID MD RAID management functions.

lmi.scripts.storage.raid.**create_raid**(*ns*, *devices*, *level*, *name=None*)
Create new MD RAID device.

Parameters

- **device** – Devices to add to the RAID.
- **level** (*int*) – RAID level.
- **name** (*string*) – RAID name.

Return type LMIInstance/LMI_MDRAIDStorageExtent

lmi.scripts.storage.raid.**delete_raid**(*ns*, *raid*)
Destroy given RAID device

Parameters **raid** (*LMIInstance/LMI_MDRAIDStorageExtent*) – MD RAID to destroy.

lmi.scripts.storage.raid.**get_raid_members**(*ns*, *raid*)
Return member devices of the RAID.

Parameters **raid** (*LMIInstance/LMI_MDRAIDStorageExtent*) – MD RAID to examine.

Return type List of LMIInstance/CIM_StorageExtent

lmi.scripts.storage.raid.**get_raids**(*ns*)
Retrieve list of all MD RAIDs on the system.

Return type list of LMIInstance/LMI_MDRAIDStorageExtent.

Filesystems and data formats Filesystem management functions.

lmi.scripts.storage.fs.**create_fs**(*ns*, *devices*, *fs*, *label=None*)
Format given devices with a filesystem. If multiple devices are provided, the format will span over all these devices (currently supported only for btrfs).

Parameters

- **devices** (*list of LMIInstance/CIM_StorageExtent or list of strings*) – Devices to format.
- **fs** (*string*) – Requested filesystem type (case-insensitive).
- **label** (*string*) – The filesystem label.

Return type LMIInstance/CIM_LocalFileSystem

lmi.scripts.storage.fs.**delete_format**(*ns*, *fmt*)
Remove given filesystem or data format from all devices, where it resides.

Parameters **fmt** (*LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat*) – Format to delete.

lmi.scripts.storage.fs.**get_device_format_label**(*ns*, *device*)
Return short text description of the format, ready for printing.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device to describe.

Return type string

lmi.scripts.storage.fs.**get_format_label**(*_ns*, *fmt*)
Return short text description of the format, ready for printing.

Parameters **fmt** (*LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat*) – Format to describe.

Return type string

lmi.scripts.storage.fs.**get_format_on_device**(*ns*, *device*, *format_type=3*)
Return filesystem or data format, which is on given device.

Parameters

- **device** (*LMIInstance/CIM_StorageExtent or string*) – Device to to examine.
- **format_type** (*int*) – Type of format to find.
  - FORMAT_ALL - return either CIM_LocalFileSystem or LMI_DataFormat.

- **FORMAT_FS - return only CIM_LocalFileSystem or None, if there is no** filesystem on the device.
- **FORMAT_DATA - return only LMI_DataFormat or None, if there is no** data format on the device.

Return type LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat

lmi.scripts.storage.fs.**get_formats**(*ns*, *devices=None*, *format_type=3*, *nodevfs=False*)
Retrieve list of filesystems on given devices. If no devices are given, all formats on all devices are returned.

Parameters

- **devices** (*list of LMIInstance/CIM_StorageExtent or list of strings*) – Devices to list formats on.
- **format_type** (*int*) – Type of formats to find.
  - FORMAT_ALL - return either CIM_LocalFileSystem or LMI_DataFormat.
  - **FORMAT_FS - return only CIM_LocalFileSystem or None, if there is no** filesystem on the device.
  - **FORMAT_DATA - return only LMI_DataFormat or None, if there is no** data format on the device.
- **nodevfs** (*bool*) – Whether non-device filesystems like tmpfs, cgroup, procfs etc. should be returned.

Return type list of LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat

lmi.scripts.storage.fs.**str2format**(*ns*, *fmt*)
Convert string with name of device to LMIInstance of the format on the device.

If LMIInstance is provided, nothing is done and the instance is just returned. If a string is given, appropriate LMIInstance is looked up and returned.

This functions throws an error when the device cannot be found.

Parameters **fmt** (*LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat or string*) – The format.

Retval LMIInstance/CIM_LocalFileSystem or LMIInstance/LMI_DataFormat

Printing Functions to display information about block devices.

lmi.scripts.storage.show.**device_show**(*ns*, *device*, *human_friendly*)
Print extended information about the device.

Parameters

- **part** – Device to show.
- **human_friendly** (*bool*) – If True, the device sizes are shown in human-friendly units (KB, MB, ...).

lmi.scripts.storage.show.**device_show_data**(*ns*, *device*, *human_friendly*)
Display description of data on the device.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device to show.

lmi.scripts.storage.show.**device_show_device**(*ns*, *device*, *human_friendly*)
Print basic information about storage device, common to all device types.

Parameters **device** (*LMIInstance/CIM_StorageExtent or string*) – Device to show.

lmi.scripts.storage.show.**format_show**(*ns*, *fmt*, *human_friendly*)
Display description of data on the device.

Parameters **fmt** (*LMIInstance/LMI_DataFormat or string*) – Format to show.

lmi.scripts.storage.show.**fs_show**(*ns*, *fmt*, *human_friendly*)
Display description of filesystem on the device.

Parameters **fmt** (*LMIInstance/CIM_LocalFileSystem or string*) – Filesystem to show.

lmi.scripts.storage.show.**lv_show**(*ns*, *lv*, *human_friendly*)
Print extended information about the Logical Volume.

Parameters **lv** (*LMIInstance/LMI_LVStorageExtent or string*) – Logical Volume to show.

lmi.scripts.storage.show.**partition_show**(*ns*, *part*, *human_friendly*)
Print extended information about the partition.

Parameters **part** (*LMIInstance/CIM_GenericDiskPartition or string*) – Partition to show.

lmi.scripts.storage.show.**partition_table_show**(*ns*, *disk*, *human_friendly*)
Print extended information about the partition table on given disk.

Parameters **disk** (*LMIInstance/CIM_StorageExtent or string*) – Device with partition table to show.

lmi.scripts.storage.show.**raid_show**(*ns*, *r*, *human_friendly*)
Print extended information about the RAID.

Parameters **r** (*LMIInstance/LMI_MDRAIDStorageExtent or string*) – RAID to show.

lmi.scripts.storage.show.**tlv_show**(*ns*, *tlv*, *human_friendly*)
Print extended information about the Thin Logical Volume.

Parameters **tlv** (*LMIInstance/LMI_LVStorageExtent or string*) – Thin Logical Volume to show.

lmi.scripts.storage.show.**vg_show**(*ns*, *vg*, *human_friendly*)
Print extended information about the Volume Group.

Parameters **vg** (*LMIInstance/LMI_VGStoragePool or string*) – Volume Group to show.

System Script python reference

Main interface function wrapped with `lmi` command is:

- `get_system_info()`

It accepts NS object as the first argument, an instance of `lmi.shell.LMINamespace`.

System Module API LMI system client library.

lmi.scripts.system.**format_memory_size**(*size*)
Returns formatted memory size.

Parameters **size** (*Number*) – Size in bytes

Returns Formatted size string.

Return type String

lmi.scripts.system.**get_all_instances**(*ns*, *class_name*)
Returns all instances of instance_name.

Parameters **instance_name** (*String*) – Instance name

Returns List of instances of instance_name

Return type List of `lmi.shell.LMIInstance`

`lmi.scripts.system.`**`get_colored_string`**(*msg*, *color*)
Returns colored message with ANSI escape sequences for terminal.

Parameters

- **msg** (*String*) – Message to be colored.
- **color** (*Integer*) – Color of the message [GREEN_COLOR, YELLOW_COLOR, RED_COLOR].

Returns Colored message.

Return type String

`lmi.scripts.system.`**`get_hostname`**(*ns*)

Returns Tabular data of system hostname.

Return type List of tuples

`lmi.scripts.system.`**`get_hwinfo`**(*ns*)

Returns Tabular data of system hw info.

Return type List of tuples

`lmi.scripts.system.`**`get_networkinfo`**(*ns*)

Returns Tabular data of networking status.

Return type List of tuples

`lmi.scripts.system.`**`get_osinfo`**(*ns*)

Returns Tabular data of system OS info.

Return type List of tuples

`lmi.scripts.system.`**`get_servicesinfo`**(*ns*)

Returns Tabular data of some system services.

Return type List of tuples

`lmi.scripts.system.`**`get_single_instance`**(*ns*, *class_name*)
Returns single instance of instance_name.

Parameters **instance_name** (*String*) – Instance name

Returns Instance of instance_name

Return type `lmi.shell.LMIInstance`

`lmi.scripts.system.`**`get_system_info`**(*ns*)

Returns Tabular data of all general system information.

Return type List of tuples

3.2 OpenLMI server components

On servers (= managed systems), OpenLMI leverages WBEM infrastructure we already have in Linux and only adds the missing pieces: providers.

See our overview for details what is a provider and how the whole CIM+WBEM infrastructure is supposed to work.

Table of contents:

3.2.1 Usage & Troubleshooting

Installation

Fedora, Red Hat Enterprise Linux & derived Linux distributions

In Fedora Linux, one just needs to install OpenLMI packages:

```
$ yum install openlmi-networking openlmi-storage <any other providers>
```

From source code

Please refer to README of individual providers, either in git or in released tarballs.

Configuration files

`/etc/openlmi/openlmi.conf` is OpenLMI master configuration file.

Each provider may introduce additional configuration files, see their documentation. If a provider uses its own configuration file, the provider-specific one is parsed first and all missing options are then read from OpenLMI master configuration file.

Using this approach, administators can set e.g. one namespace for all providers in `/etc/openlmi/openlmi.conf` and different log levels for some providers in their configuration files.

File format

Configuration files has simple .ini syntax, with `#` or `;` used for comments.

Default configuration:

```
[CIM]
Namespace=root/cimv2
SystemClassName=PG_ComputerSystem

[Log]
Level=ERROR
Stderr=false
```

Section	Option name	Default value	Description
CIM	Namespace	root/cimv2	Namespace where OpenLMI providers are registered.
CIM	SystemClassName	PG_ComputerSystem	Name of CIM_ComputerSystem class, which is used to represent the computer system. It will be used as SystemClassName property value of various classes. Different cimmoms can instrument variously named computer systems and some may not instrument any at all. Sfcb is an example of the later, it needs the sblim-cmpi-base package installed providing the basic set of providers containing Linux_ComputerSystem. So in case you run a Sfcb or you preferr to use providers from sblim-cmpi-base package, you need to change this to Linux_ComputerSystem.
Log	Level	ERROR	Chooses which messages are logged, either to CIMOM and (if configured) to standard error output. Available levels (sorted by severity) are: • CRITICAL • ERROR • WARNING • INFO • DEBUG • TRACE_WARNING • TRACE_INFO • TRACE_VERBOSE Levels below INFO (= TRACE_WARNING, TRACE_INFO and DEBUG) are useful mainly for debugging and bug reporting.
Log	Stderr	False	Toggles sending of log messages to standard error output of the CIMOM. Accepts boolean value (see the next section).

Treating boolean values

Options expecting boolean values treat following strings as valid *True* values: `true`, `1`, `yes` and `on`. While the following are considered *False*: `false`, `0`, `no` and `off`. These words are checked in a case-insensitive way. Any other value isn't considered valid [16].

Logging

If logging is enabled, all log messages with level INFO and above are sent to CIMOM using standard CMPI `CMLogMessage` function. Consult documentation of your CIMOM how to enable output of these messages into CIMOM logs.

Messages with TRACE_WARNING and below are sent to CIMOM using `CMTraceMessage` and should be visible in CIMOM tracing log. Again, please consult your CIMOM documentation how to enable tracing logs.

With `Stderr` configuration option enabled, all logs are sent both to CIMOM and to the standard error output of the CIMOM.

Logging in Pegasus

When using Pegasus CIMOM, the easiest way is to let Pegasus daemon run in foreground and send log messages to its standard error output.

Sample `/etc/openlmi/openlmi.conf`:

```
[CIM]
Namespace = root/cimv2
SystemClassName = PG_ComputerSystem

[Log]
Level = TRACE_INFO
Stderr = True
```

Run Pegasus in foreground, i.e. with `stderr` output sent to terminal:

```
$ /sbin/cimserver daemon=false
INFO:cimom_entry:get_providers:146 - Provider init.
INFO:TimerManager:_timer_loop:246 - Started Timer thread.
Level 8:cmpi_logging:trace_info:126 - Timer: Checking for expired, now=17634.607226.
Level 8:cmpi_logging:trace_info:126 - Timer: No timers scheduled, waiting forever.
INFO:cimom_entry:init_anaconda:118 - Initializing Anaconda
INFO:JobManager:_worker_main:877 - Started Job thread.
```

Of course, more advanced logging can be configured in runtime to send provider logs into trace files, see Pegasus documentation for details.

Note: OpenLMI providers will start logging only after they are started, i.e. when they are used for the first time.

[16] Default value will be used as a fallback. This applies also to other non-boolean options in case of invalid value.

3.2.2 Account Provider

OpenLMI Account is CIM provider which manages POSIX accounts. It allows to create, delete and modify users and groups.

The provider implements DMTF identity profile, for more details read *DMTF profile*.

Contents:

DMTF profile

The provider implements DMTF's Simple Identity Management Profile, version 1.0.1.

Profile adjustment

The settings classes are not implemented. Necessary settings are done directly in methods of *LMI_AccountManagementService*. *LMI_AccountManagementService* is subclass of *CIM_SecurityService*, because there is a change in method parameters as follows:

- *CreateAccount* does not take EmbeddedInstance as parameter, but a list of parameters.

Implementation

All mandatory classes are implemented.

Classes Implemented DMTF classes:

- *LMI_AccountCapabilities*
- *LMI_AccountInstanceCreationIndication*
- *LMI_AccountInstanceDeletionIndication*
- *LMI_AccountManagementCapabilities*
- *LMI_AccountManagementServiceCapabilities*
- *LMI_AccountManagementService*
- *LMI_AccountManagementServiceSettingData*
- *LMI_AccountOnSystem*
- *LMI_Account*
- *LMI_AccountSettingData*
- *LMI_AssignedAccountIdentity*
- *LMI_AssignedGroupIdentity*
- *LMI_EnabledAccountCapabilities*
- *LMI_Group*
- *LMI_HostedAccountManagementService*
- *LMI_Identity*
- *LMI_MemberOfGroup*

- *LMI_OwningGroup*
- *LMI_ServiceAffectsIdentity*
- *LMI_SettingsDefineAccountCapabilities*
- *LMI_SettingsDefineManagementCapabilities*

Methods Implemented:

- *CreateAccount*

Additional methods:

- *CreateGroup*

Usage

General manipulation of users and groups are done with the objects from following classes:

- *LMI_AccountManagementService*
- *LMI_Account*
- *LMI_Group*
- *LMI_MemberOfGroup*
- *LMI_Identity*
- *LMI_AccountInstanceCreationIndication*
- *LMI_AccountInstanceDeletionIndication*

Some common use cases are described in the following parts

Note: Examples are written for `lmishell` version `0.9`.

List users

List of users are provided by *LMI_Account*. Each one object of this class represents one user on the system. Both system and non-sytem users are directly in *LMI_Account* class:

```
# List user by name
print c.root.cimv2.LMI_Account.first_instance({"Name": "root"})
# List user by id
print c.root.cimv2.LMI_Account.first_instance({"UserID": "0"})
```

List groups

Similarly like users, groups are represented by objects of *LMI_Group* class:

```
# List group by name
print c.root.cimv2.LMI_Group.first_instance({"Name": "root"})
# List group by id
print c.root.cimv2.LMI_Group.first_instance({"InstanceID": "LMI:GID:0"})
```

List group members

LMI_Identity is class representing users and groups on the system. Group membership is represented by *LMI_MemberOfGroup* association. It associates *LMI_Group* and *LMI_Identity*, where *LMI_Identity* is associated by *LMI_AssignedAccountIdentity* with *LMI_Account*:

```
# Get users from root group
# 1) Get root group object
root_group = c.root.cimv2.LMI_Group.first_instance({"Name": "root"})
# 2) Get LMI_Identity objects associated with root group
identities = root_group.associators(
     AssocClass="LMI_MemberOfGroup", ResultClass="LMI_Identity")
# 3) go through all identites, get LMI_Account associated with identity and print user name
# Note: associators returns a list, but there is just one LMI_Account
for i in identities:
    print i.first_associator(
            AssocClass="LMI_AssignedAccountIdentity",
            ResultClass="LMI_Account").Name
```

Create user

For user creation we have to use *LMI_AccountManagementService*. There is *CreateAccount* method, which will create user with descired attributes:

```
# get computer system
cs = c.root.cimv2.PG_ComputerSystem.first_instance()
# get service
lams = c.root.cimv2.LMI_AccountManagementService.first_instance()
# invoke method, print result
lams.CreateAccount(Name="lmishell-user", System=cs)
```

Create group

Similarly like creating user, creating groups are don in *LMI_AccountManagementService*, using *CreateGroup* method:

```
# get computer system
cs = c.root.cimv2.PG_ComputerSystem.first_instance()
# get service
lams = c.root.cimv2.LMI_AccountManagementService.first_instance()
# invoke method, print result
print lams.CreateGroup(Name="lmishell-group", System=cs)
```

Delete user

User deletion is done with *DeleteUser* method on the desired *LMI_Account* object.

```
# get the desired user
acci = c.root.cimv2.LMI_Account.first_instance({"Name": "tobedeleted"})
# delete the user
acci.DeleteUser()
```

Note: Previous releases allowed to use `DeleteInstance` intrinsic method to delete `LMI_Account`. This method is now deprecated and will be removed from future releases of OpenLMI Account. The reason is that

`DeleteInstance` cannot have parameters; it is equivalent to call `DeleteAccount` without specifying parameters.

Delete group

Group deletion is done with *DeleteGroup* method on the desired *LMI_Group* object,

```
# get the desired group
grp = c.root.cimv2.LMI_Group.first_instance({"Name": "tobedeleted"})
# delete the group
grp.DeleteGroup()
```

Note: Previous releases allowed to use `DeleteInstance` intrinsic method to delete `LMI_Group`. This method is now deprecated and will be removed from future releases of OpenLMI Account. The reason is that we want to have consistent way to delete user and group.

Add user to group

Adding user to group is done with `CreateInstance` intrinsic method on the *LMI_MemberOfGroup* class, which requires reference to *LMI_Group* and *LMI_Identity*:

```
# We will add root user to pegasus group
# get group pegasus
grp = c.root.cimv2.LMI_Group.first_instance_name({"Name": "pegasus"})
# get user root
acc = c.root.cimv2.LMI_Account.first_instance({"Name": "root"})
# get identity of root user
identity = acc.first_associator_name(
        AssocClass='LMI_AssignedAccountIdentity',
        ResultClass="LMI_Identity")
# create instance of LMI_MemberOfGroup with the above references
c.root.cimv2.LMI_MemberOfGroup.create_instance({"Member":identity, "Collection":grp})
```

Remove user from group

Removing user from group is done with `DeleteInstance` intrinsic method on the desired *LMI_MemberOfGroup* object:

```
# We will remove root user from pegasus group
# get group pegasus
grp = c.root.cimv2.LMI_Group.first_instance_name({"Name": "pegasus"})
# get user root
acc = c.root.cimv2.LMI_Account.first_instance({"Name": "root"})
# get identity of root user
identity = acc.first_associator(
        AssocClass="LMI_AssignedAccountIdentity",
        ResultClass="LMI_Identity")
# iterate through all LMI_MemberOfGroup associated with identity and remove the one with our group
for mog in identity.references(ResultClass="LMI_MemberOfGroup"):
    if mog.Collection == grp:
        mog.delete()
```

Modify user

It is also possible to modify user details and it is done by `ModifyInstance` intrinsic method on the desired *LMI_Account* object:

```
# Change login shell of test user
acci = c.root.cimv2.LMI_Account.first_instance({"Name": "test"})
acci.LoginShell = '/bin/sh'
# propagate changes
acci.push()
```

Indications

OpenLMI Account supports the following indications:

- *LMI_AccountInstanceCreationIndication*
- *LMI_AccountInstanceDeletionIndication*

Both indications work only on the following classes:

- *LMI_Account*
- *LMI_Group*
- *LMI_Identity*

Please see LMIShell Indications API reference for an overview how indications work.

Creation Indication Client can be notified when instance of class has been created. It is done with *LMI_AccountInstanceCreationIndication*. The indication filter query must be in the following form: `SELECT * FROM LMI_AccountInstanceCreationIndication WHERE SOURCEINSTANCE ISA class_name`, where `class_name` is one of the allowed classes.

The following example creates filter, handler and subscription (lmi shell does that in one step), which will notify client when user is created:

```
# Notify when a user is created
c.subscribe_indication(
    Name="account_creation",
    Query='SELECT * FROM LMI_AccountInstanceCreationIndication WHERE SOURCEINSTANCE ISA LMI_Account',
    Destination="http://192.168.122.1:5988" # this is the destination computer, where all the indicat
)
```

Deletion Indication Client can be notified when instance is deleted. The same rules like in Creation Indication applies here:

```
# Notify when a user is deleted
c.subscribe_indication(
    Name="account_deletion",
    Query='SELECT * FROM LMI_AccountInstanceDeletionIndication WHERE SOURCEINSTANCE ISA LMI_Account',
    Destination="http://192.168.122.1:5988" # this is the destination computer, where all the indicat
)
```

Note: Both indications use the indication manager and polling.

Creation Indication example The following code snippet illustrates usage of indication listener and subscription. It is a complete minimal example of user creation. Once a new account is added, simple informational message is printed on the standard output.

```
#!/usr/bin/lmishell

from lmi.shell import LMIIndicationListener
import socket
import time
import random

def ind_handler(indication, **kwargs):
    print "User '%s' added" % indication["SourceInstance"]["Name"]

c = connect("localhost", "pegasus", "test")

indication_port = random.randint(12000, 13000)
listener = LMIIndicationListener("0.0.0.0", indication_port)
uniquename = listener.add_handler("account_watch-XXXXXXXX", ind_handler)
listener.start()

c.subscribe_indication(
    Name=uniquename,
    Query="select * from LMI_AccountInstanceCreationIndication where SourceInstance isa LMI_Account",
    Destination="http://%s:%d" % (socket.gethostname(), indication_port)
)

try:
    while True:
        time.sleep(0.1)
        pass

except KeyboardInterrupt:
    pass

c.unsubscribe_indication(uniquename)
```

Note: Press Ctrl+C to terminate the script. Also, remember to change the login credentials! The example picks a random port in the 12000 - 13000 range, no check for port occupancy is made, a conflict on a busy system is possible.

3.2.3 Fan Provider

Contents:

DMTF profiles

OpenLMI Fan provider implements *Fan Profile*

Fan Profile

Implemented *DMTF* version: `1.0.1`

Described by DSP1013

It defines the classes used to describe the fans and the possible redundancy of the fans in a managed system. The document also defines association classes that describe the relationship of the fan to the fan's physical aspects (such as FRU data) to the sensors monitoring the fans, to other cooling devices, to redundancy status, and to DMTF profile version information. The information in this specification is intended to be sufficient for a provider or consumer of this data to identify unambiguously the classes, properties, methods, and values that are mandatory to be instantiated and manipulated to represent and manage fans and redundant fans of managed systems and subsystems that are modeled using the DMTF CIM core and extended model definitions.

Not implemented features *DMTF* profile defines many classes that are not instrumented due to limitations of low level libraries giving informations about fans. Here is a list of not implemented classes:

`CIM_ManagedSystemElement` Models the piece of hardware being cooled by particular fan. It's associated with *LMI_Fan* through `CIM_AssociatedColling` which is also not instrumented.

`CIM_RedundancySet` Represents redundacy of fans belonging to particular computer system. It's associated with *LMI_Fan* through `CIM_MemberOfCollection` and `CIM_IsSpare` associations. There is no way how to detect whether the fan is spare or not.

Classes that shall be implemented There are still classes missing implementation and are planned to be delivered in future versions.

`CIM_SystemDevice` Associates *LMI_Fan* to `CIM_ComputerSystem`.

`CIM_EnabledLogicalElementCapacilities` Represents the capabilities of associated fan. It's associated to *LMI_Fan* through `CIM_ElementCapabilities`.

Not implemented optional features *Physical Asset* association from *LMI_Fan* to `CIM_PhysicalPackage` through `CIM_Realizes` association class is not instrumented. This is an optional feature. It may be implemented later.

Physical Asset is a related profile implemented by *OpenLMI Hardware* provider.

Class overview

Class-name	Parent_class	Type
LMI_Fan	*CIM_Fan*	Plain
LMI_FanSensor	*CIM_NumericSensor*	Plain
LMI_FanAssociatedSensor	*CIM_AssociatedSensor*	Association

LMI_Fan Represents the the fan installed and connected to computer. One of the most important keys is *DeviceID*. It's a *sys* path to kernel driver's abstraction for fan combined with its name.

Typical sys directory for fan looks like this:

```
/sys/class/hwmon/hwmon1/device/
-- driver -> ../../../bus/platform/drivers/thinkpad_hwmon
-- fan1_input
-- hwmon
|   -- hwmon1
|       -- device -> ../../../thinkpad_hwmon
|       -- power
|       |   -- async
|       |   -- autosuspend_delay_ms
|       |   -- control
|       |   -- runtime_active_kids
```

```
|       |   -- runtime_active_time
|       |   -- runtime_enabled
|       |   -- runtime_status
|       |   -- runtime_suspended_time
|       |   -- runtime_usage
|       -- subsystem -> ../../../../../class/hwmon
|       -- uevent
-- modalias
-- name
-- power
|   -- async
|   -- autosuspend_delay_ms
|   -- control
|   -- runtime_active_kids
|   -- runtime_active_time
|   -- runtime_enabled
|   -- runtime_status
|   -- runtime_suspended_time
|   -- runtime_usage
-- pwm1
-- pwm1_enable
-- subsystem -> ../../../bus/platform
-- uevent
```

Corresponding `DeviceID` is `/sys/class/hwmon/hwmon1/device/fan1`. The fan name is the prefix of `*_input` file which gives the current RPM (Revolutions per minute) value.

It has several other interesting properties:

> ***OtherIdentifyingInfo*** `[string []]` Has the name of chip controlling the fan as the first item.

LMI_FanSensor Represents a sensor measuring a speed of particular fan. It's exactly the same keys and values except for *CreationClassName* containg the name of corresponding class `LMI_Fan`.

It inherts many methods that are not supported because underlying library does not offer such functionality. Controlling of fans is very hardware dependent. Different drivers may provide different ways and possibilities to manage connected fans.

Usage

Examples for common use cases listed below are written in lmishell.

Set up

OpenLMI Fan provider uses lm-sensors to find, observe and manage installed fans. In order to make the fans exposed to it, one operation needs to be done:

```
sensors-detect
```

`sensors-detect` is a script shiped with `lm_sensors` package in *Fedora* which tries to load correct modules for various sensor devices found in system. It also writes a config used by `sensors` library which is utilised in this provider. Please refer to its *sensors-detect (8)* man-page.

Examples

Listing installed fans

```
c = connect("host", "user", "pass")
for fan in c.root.cimv2.LMI_Fan.instances():
    print(fan.ElementName)
```

See also:

LMI_Fan

Getting fan's speed Current value can be read from *CurrentReading* property. It's measured in *revolutions per minute*.

```
c = connect("host", "user", "pass")
for fan in c.root.cimv2.LMI_FanSensor.instances():
    print("%s:\t%s RPM" % (fan.Name, fan.CurrentReading))
```

See also:

LMI_FanSensor

3.2.4 Hardware Provider

OpenLMI Hardware is CIM provider which can provide hardware information.

The provider is partially implementing DMTF Computer System Profile with addition of multiple hardware related profiles. For more information see *DMTF profiles*.

Contents:

DMTF profiles

The provider is partially implementing DMTF's Computer System Profile, version 1.0.2, with addition of multiple hardware related profiles. Complete list of implemented profiles can be found below.

CPU Profile

CPU DMTF Profile, version 1.0.1.

Classes Implemented DMTF classes:

- *LMI_Processor*
- *LMI_ProcessorCapabilities*
- *LMI_ProcessorElementCapabilities*
- *LMI_ProcessorCacheMemory*
- *LMI_AssociatedProcessorCacheMemory*
- *LMI_ProcessorChip*
- *LMI_ProcessorChipRealizes*
- *LMI_ProcessorChipContainer*

- *LMI_ProcessorSystemDevice*

System Memory Profile

System Memory DMTF Profile, version 1.0.1.

Classes Implemented DMTF classes:

- *LMI_Memory*
- *LMI_MemoryPhysicalPackage*
- *LMI_PhysicalMemory*
- *LMI_PhysicalMemoryRealizes*
- *LMI_PhysicalMemoryContainer*
- *LMI_MemorySlot*
- *LMI_MemorySlotContainer*
- *LMI_MemoryPhysicalPackageInConnector*
- *LMI_MemorySystemDevice*

Physical Asset Profile

Physical Asset DMTF Profile, version 1.0.2.

Classes Implemented DMTF classes:

- *LMI_Chassis*
- *LMI_Baseboard*
- *LMI_BaseboardContainer*
- *LMI_PointingDevice*
- *LMI_PortPhysicalConnector*
- *LMI_PortPhysicalConnectorContainer*
- *LMI_SystemSlot*
- *LMI_SystemSlotContainer*
- *LMI_ChassisComputerSystemPackage*

Battery Profile

Battery DMTF Profile, version 1.0.0.

Classes Implemented DMTF classes:

- *LMI_Battery*
- *LMI_BatteryPhysicalPackage*
- *LMI_PhysicalBatteryContainer*
- *LMI_PhysicalBatteryRealizes*
- *LMI_BatterySystemDevice*

PCI Device Profile

PCI Device DMTF Profile, version 1.0.0.

Classes Implemented DMTF classes:

- *LMI_PCIDevice*
- *LMI_PCIDeviceSystemDevice*
- *LMI_PCIBridge*
- *LMI_PCIBridgeSystemDevice*

Disk Drive Profile

Storage Management Technical Specification, Part 3 Block Devices SNIA Profile, Clause 11: Disk Drive Lite Subprofile, version 1.6.0, revision 4.

Classes Implemented DMTF classes:

- *LMI_DiskPhysicalPackage*
- *LMI_DiskPhysicalPackageContainer*
- *LMI_DiskDrive*
- *LMI_DiskDriveRealizes*
- *LMI_DiskDriveSoftwareIdentity*
- *LMI_DiskDriveElementSoftwareIdentity*
- *LMI_DiskDriveATAProtocolEndpoint*
- *LMI_DiskDriveSAPAvailableForElement*
- *LMI_DiskDriveATAPort*
- *LMI_DiskDriveDeviceSAPImplementation*
- *LMI_DiskDriveSystemDevice*

Usage

OpenLMI Hardware provider contains hardware information, it does not implement any methods. List of provided information divided by DMTF profiles can be found below.

CPU Profile

CPU Profile provides information about CPU and associated cache:

- Processor
  - Number of CPUs, cores, threads
  - Model
  - Clock and FSB speeds
  - Data and Address width
  - Architecture
  - Flags
  - Family
  - Stepping
  - FRU data (Manufacturer, Model, Serial Number, Part Number)
- Processor Cache
  - Level
  - Size
  - Type (Data / Instruction / Unified)

Used Resources

- dmidecode program *[from dmidecode package]*
- lscpu program *[from util-linux package]*
- /proc/cpuinfo file
- /sys/devices/system/cpu/* files

System Memory Profile

System Memory Profile provides information about system memory and slots:

- Memory
  - Size
  - Speed (in both MHz and ns)
  - Size of standard memory page
  - All supported sizes of huge pages
  - Current state of transparent huge pages [Unsupported, Never, Madvise, Always]
  - Detection of NUMA layout
- Memory slots + modules
  - Number of slots and modules
  - In which slots are modules plugged in

  - Size of modules
  - Speed of modules
  - Data and Total width
  - Module type and form factor
  - FRU data

Used Resources

- dmidecode program *[from dmidecode package]*
- /proc/meminfo file
- /sys/devices/system/node/* files
- /sys/kernel/mm/hugepages/* files
- /sys/kernel/mm/transparent_hugepage/* files

Physical Asset Profile

Physical Asset Profile provides basic information about physical assets in system, usually with FRU data, currently for following hardware (with associations):

- System chassis
- Baseboard (motherboard)
- Chassis ports (USB, LAN, VGA..)
- Chassis slots (Media card slot, Express card slot..)
- Pointing devices on chassis (Touch pad, Track point..)

Used Resources

- dmidecode program *[from dmidecode package]*

Battery Profile

Battery Profile provides basic information about battery:

- Capacity
- Voltage
- Chemistry
- FRU data

Used Resources

- dmidecode program *[from dmidecode package]*

PCI Device Profile

PCI Device Profile provides information about PCI devices:

- PCI Devices:
  - Bus Number
  - Device Number
  - Function Number
  - PCI Device ID
  - PCI Device Name
  - Vendor ID
  - Vendor Name
  - Subsystem ID
  - Subsystem Name
  - Subsystem Vendor ID
  - Subsystem Vendor Name
  - Revision ID
  - Base Address
  - Cache Line Size
  - Capabilities
  - Class Code
  - Command Register
  - Device Select Timing
  - Interrupt Pin
  - Latency Timer
  - Expansion ROM Base Address
- PCI Bridges (all of the above, plus):
  - Bridge Type
  - Primary Bus Number
  - Secondary Bus Number
  - Subordinate Bus Number
  - Secondary Latency Timer
  - IO Base
  - IO Limit
  - Memory Base
  - Memory Limit
  - Prefetch Memory Base
  - Prefetch Memory Limit

Used Resources

- libpci library *[from pciutils package, pci/pci.h header file]*

Disk Drive Profile

Disk Drive Profile provides information about disk drives:

- Disk Drive:
  - Overall S.M.A.R.T. status
  - Temperature
  - Capacity
  - Manufacturer
  - Model
  - Serial Number
  - Firmware version
  - Form Factor (disk size: 2.5", 3.5"..)
  - RPM
  - Port Type (ATA/SATA/SATA2)
  - Max Port Speed
  - Current Port Speed
  - Disk Type (HDD/SSD)

Used Resources

- lsblk program *[from util-linux package]*
- smartctl program *[from smartmontools package]*
- /sys/class/block/*/device/vendor file
- /sys/class/block/*/queue/rotational file

3.2.5 Journald Provider

OpenLMI Journald is a CIM provider exposing systemd journald log records and basic means of iteration and log writing.

Journald is a daemon working with journals. Journal is a log, a set of log records, chronologically ordered. Records are structured, able to carry multiple (custom) data fields. By implementation, journald is able to work with multiple (separate) journals but we use the mixed way for the moment, which is typical in production use.

Classes used by the provider were chosen to mimic the sblim-cmpi-syslog provider set of classes allowing drop-in replacement in production tools. We haven't been able to find a profile it conforms to though. There's a related DMTF profile DSP1010 "Record Log Profile" which may be subject to extension of this provider in the future. As a benefit, by using the parent classes (e.g. *CIM_LogRecord*), one is able to mix log records from orthodox syslog and journald together.

Provider features

This is a short list of provider features:

- log records reading
- log record iteration using persistent iterators
- new records indication
- writing new log records

For the moment, global journal is used, all journal files are mixed together.

The provider also comes with a test suite covering most of its functionality.

Contents

Caveats

There are some specifics when working with journald and OpenLMI journald provider.

Number of LMI_JournalLogRecord instances enumerated limitation

Testing the provider showed up an issue with enumeration of *LMI_JournalLogRecord* instances. On the testing machine there was 199583 journal records, which is simply too much for the CIMOM, exceeding memory and the resulting XML reply limits.

An artificial limit has been set, currently to 1000 most recent records. This limit is defined by the `JOURNAL_MAX_INSTANCES_NUM` define in `Journal.h` source file. Please use iterators instead to get access to all records.

Iteration and iterators

Iteration is a different way of getting data through the log records. Comparing to the usual instance enumeration, this is a sequential-like access with ability to seek back and forth in the journal. Retrieving individual records might be slower than direct random access though memory consumption is kept on a low level.

Please check the *LMI_JournalMessageLog* class reference for detailed description of available iterator-related methods. Implemented iterator methods are *PositionToFirstRecord()*, *PositionAtRecord()*, *GetRecord()* and *CancelIteration()*. Only relative movement is supported by the *PositionAtRecord()* method.

A key element of the iteration process is the iteration identifier that is typically passed in the methods listed above. Only the *PositionToFirstRecord()* method is able to create new iteration identifier without the need of specifying one.

Iteration identifiers are specific to the provider and are opaque. They're are persistent to some extent, surviving unexpected CIMOM runtime cleanup. The only requirement for persistency to work is the journal record the iterator identifier previously pointed to to be available at the time the iterator is reused. I.e. it won't survive log rotation.

A remark for the *LMI_JournalMessageLog.GetRecord()* method: the outgoing RecordData argument carries string data encoded in an array of uint8 elements as defined by the model. This is quite limiting and also still very free-form on the other hand. To conform the definition, we put UTF-8 encoded string split by characters in the array and is up to clients to decode it back to a readable form.

New log records writing security concerns

The provider has an ability to send new messages to the log. This may be percieved as a security issue in someone's eyes as long as you can specify custom message format that is sent to the log. The only obstacle preventing anyone in sending spoof messages is the rather weak CIM authentication model.

However, as long as journald is a structured logging system, further information is stored along every log record. Messages sent through the OpenLMI Journald provider may be identified by supplemental fields such as `_COMM` and `_EXE`, pointing to a CIMOM that had been running the provider code or even the `CODE_FUNC` field, pointing to a specific function that invoked the journald library code.

Potential indications endless loop

Just a note for implementing a system processing the indications. Having no specific filter for the indication subscription and performing an action within the indication handler that involves a message being sent to syslog may result in an endless loop as long such action generates another indication for the fresh syslog message. Even a CIMOM in certain situations (i.e. debugging in verbose mode) may generate additional messages while sending an indication that in turn will generate another one.

Usage

The OpenLMI Journald provider depends on running journald daemon. See the systemd manual for how to enable the journald service.

Listing a log

This example shows simple enumeration through available *LMI_JournalLogRecord* instances in classic syslog-like format:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
for rec in c.root.cimv2.LMI_JournalMessageLog.first_instance().associators():
    print "%s %s %s" % (rec.MessageTimestamp.datetime.ctime(), rec.HostName, rec.DataFormat)
```

Note: Only a limited number of records are being enumerated and printed out, please see the *Number of LMI_JournalLogRecord instances enumerated limitation* remark.

Using WQL query for simple filtering

From its nature LMIShell can only do simple filtering by matching exact property values. However there's a posibility of constructing custom CQL or WQL queries bringing more flexibility in specific test conditions. The result from the query method call is a list of instances, similar to calling "`.associators()`" or "`.instances()`".

The following example uses WQL query to get a list of messages with syslog severity 3 (error) or higher:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
for rec in c.root.cimv2.wql("SELECT * FROM LMI_JournalLogRecord WHERE SyslogSeverity <= 3"):
    print "[severity %d] %s" % (rec.SyslogSeverity, rec.DataFormat)
```

Iterating through the log

This example uses iterator methods of the *LMI_JournalMessageLog* class to continuously go through the whole journal:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
inst = c.root.cimv2.LMI_JournalMessageLog.first_instance()
r = inst.PositionToFirstRecord()
iter_id = r.rparams['IterationIdentifier']
while True:
    x = inst.GetRecord(IterationIdentifier=iter_id, PositionToNext=True)
    if x.rval != 0:
        break
    print "".join(map(chr, x.rparams['RecordData']))
    iter_id = x.rparams['IterationIdentifier']
```

Sending new message to log

Simple example that uses *LMI_JournalLogRecord.create_instance()* CIM method to send a new message in the log:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
c.root.cimv2.LMI_JournalLogRecord.create_instance({"CreationClassName": "LMI_JournalLogRecord",
                                                   "LogCreationClassName": "LMI_JournalMessageLog",
                                                   "LogName": "Journal",
                                                   "DataFormat": ""})
```

Indications

The Journald provider comes with a *LMI_JournalLogRecordInstanceCreationIndication* class that can be used to receive indications when new log message is logged in the journal. This way user is notified about system events.

Please see LMIShell Indications API reference for an overview how indications work.

Simple indication listener The following piece of code sets up a simple indication listener and waits for any new messages. Press Ctrl+C to end the script.

```
#!/usr/bin/lmishell

from lmi.shell import LMIIndicationListener
import socket
import time
import random

def ind_handler(indication, **kwargs):
    print indication["SourceInstance"]["DataFormat"]


c = connect("localhost", "pegasus", "test")

indication_port = random.randint(12000, 13000)
ind_filter = c.root.interop.CIM_IndicationFilter.first_instance(
                 {"Name": "LMI:LMI_JournalLogRecord:NewErrorMessage"})
listener = LMIIndicationListener("0.0.0.0", indication_port)
uniquename = listener.add_handler("journald_watch-XXXXXXXX", ind_handler)
listener.start()

c.subscribe_indication(
    Name=uniquename,
```

```
    Filter=ind_filter,
    Destination="http://%s:%d" % (socket.gethostname(), indication_port)
)

try:
    while True:
        time.sleep(1)
        pass
except KeyboardInterrupt:
    pass

c.unsubscribe_indication(uniquename)
```

The above script makes use of pre-defined indication filters. There are three indication filters available by default:

New message event filter When used in indication subscription this will report all newly logged messages:

```
SELECT * FROM LMI_JournalLogRecordInstanceCreationIndication WHERE
    SourceInstance ISA LMI_JournalLogRecord
```

Filter name `"LMI:LMI_JournalLogRecord:NewMessage"`.

New error message event filter This filter can be used to report all newly logged messages having syslog severity value less than 4 ("Error"), meaning error messages including more critical ones:

```
SELECT * FROM LMI_JournalLogRecordInstanceCreationIndication WHERE
    SourceInstance ISA LMI_JournalLogRecord AND
    SourceInstance.LMI_JournalLogRecord::SyslogSeverity < 4
```

Filter name `"LMI:LMI_JournalLogRecord:NewErrorMessage"`.

New critical message event filter Similar to the last one except this omits error messages and only reports critical, alert and emergency messages (see RFC 5424 for syslog severity mapping):

```
SELECT * FROM LMI_JournalLogRecordInstanceCreationIndication WHERE
    SourceInstance ISA LMI_JournalLogRecord AND "
    SourceInstance.LMI_JournalLogRecord::SyslogSeverity < 3
```

Filter name `"LMI:LMI_JournalLogRecord:NewCriticalMessage"`.

Custom event filters Apart from pre-defined indication filters the Journald provider supports custom filters. This allows user to construct a very detailed filter to satisfy specific needs. The following excerpt from the last example will make the script to report any errors coming from the "sudo" command:

```
c.subscribe_indication(
    Name=uniquename,
    Query="SELECT * FROM LMI_JournalLogRecordInstanceCreationIndication WHERE "
          "SourceInstance ISA LMI_JournalLogRecord AND "
          "SourceInstance.LMI_JournalLogRecord::SyslogSeverity < 4 AND "
          "SourceInstance.LMI_JournalLogRecord::SyslogIdentifier = 'sudo'",
    Destination="http://%s:%d" % (socket.gethostname(), indication_port)
)
```

3.2.6 Locale Provider

OpenLMI Locale is CIM provider for managing Linux locale settings (using the systemd/localed D-Bus interface).

It allows to set system locale represented by environment variables (LANG, LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, LC_MESSAGES, LC_PAPER, LC_NAME, LC_ADDRESS, LC_TELEPHONE, LC_MEASUREMENT and LC_IDENTIFICATION), set the default key mapping of the X11 servers (keyboard layouts, model, variant and options) and the default key mapping for virtual console.

If you set a new system locale with SetLocale() method, all old system locale settings will be dropped, and the new settings will be saved to disk. It will also be passed to the system manager, and subsequently started daemons will inherit the new system locale from it.

Note that already running daemons will not learn about the new system locale.

Also note that setting key mapping with SetVConsoleKeyboard() method instantly applies the new keymapping to the console, while setting the key mapping of X11 server using SetX11Keyboard() method simply sets a default that may be used by later sessions.

Contents:

Usage

Some common use cases are described in the following parts.

Getting locale settings

Create connection, get instance (assuming the default namespace 'root/cimv2' is used):

```
c = connect("https://myhost")
# optionally create namespace alias
ns = c.root.cimv2
locale = ns.LMI_Locale.first_instance()
```

Print what you're interested in:

```
# get LANG setting
print locale.Lang
# get X11Layouts
print locale.X11Layouts
# get VConsoleKeymap
print locale.VConsoleKeymap
```

Or print everything:

```
# get all available settings
locale.doc()
```

Setting system locale

Set LANG and/or set individual locale variables. Lang, LCCType, LCAddress, LCNumeric, LCTelephone, LCCollate, LCPaper, LCMonetary, LCTime, LCMessages, LCIdentification, LCName and LCMeasurement properties correspond to likewise named Linux locale environmental variables:

```
# set LANG (LANG value is used also for all other locale categories by default)
locale.SetLocale(Lang="en_US.UTF-8")
# set LANG and set different value for LC_TELEPHONE
# note that SetLocale() clears previous setting - if you want to preserve
# LANG value, you have to set it again
locale.SetLocale(Lang="en_US.UTF-8",LCTelephone="cs_CZ.UTF-8")
```

Setting default key mapping of the X11 servers

Set default key mapping for X11 server:

```
locale.SetX11Keyboard(Layouts="de")
```

Optionally set keyboard model and variant:

```
locale.SetX11Keyboard(Layouts="us",Model="dellsk8125",Variant="qwertz")
```

Set more than one layout and set option for switching between them:

```
locale.SetX11Keyboard(Layouts="us,cz,de",Options="grp:alt_shift_toggle")
```

You can set Convert parameter to 'True', mapping for virtual console will be set also then (nearest console keyboard setting for the chosen X11 setting):

```
locale.SetX11Keyboard(Layouts="us",Convert="True")
```

Setting default key mapping of the virtual console

Set default key mapping for virtual console:

```
locale.SetVConsoleKeyboard(Keymap="us")
```

Again, setting Convert to 'True' will set the nearest X11 keyboard setting for the chosen console setting:

```
locale.SetVConsoleKeyboard(Keymap="us",Convert="True")
```

3.2.7 LogicalFile Provider

OpenLMI LogicalFile is a CIM provider which provides a way to read information about files and directories. The provider also allows to traverse the file hierarchy, create and remove empty directories.

The provider implements a part of the CIM System schema (sections "Local File Systems" and "Unix System").

Contents:

Usage

There are two basic types of classes in the LogicalFile provider.

CIM_LogicalFile subclasses:

- *LMI_FIFOPipeFile*
- *LMI_UnixDeviceFile*
- *LMI_UnixDirectory*

- *LMI_UnixSocket*
- *LMI_DataFile*
- *LMI_SymbolicLink*

Subclasses derived from *CIM_LogicalFile* represent basic types of files and their system independent properties, such as if the file is readable or its modification time. The classes' names are self-explanatory. *LMI_SymbolicLink* represents symbolic link files, *LMI_UnixDeviceFile* represents unix device files, etc.

The other type of class is *LMI_UnixFile*. It is used in the Unix-like environment. Its properties are tied to the system – Linux in our case. For example, the group id of the owner or the inode number are among those properties.

To provide ways to connect the file subclasses together, LogicalFile also defines a few associations.

Association classes:

- *LMI_RootDirectory*
- *LMI_FileIdentity*
- *LMI_DirectoryContainsFile*

LMI_RootDirectory is used to connect the computer system to its root directory.

LMI_FileIdentity associates the system-independent *CIM_LogicalFile* subclasses to their respective *LMI_UnixFile* equivalents that are dependent on the system.

LMI_DirectoryContainsFile serves as a tool to show contents of a directory. Note that directory is usually just a type of file.

Deviations from the schema

No classes that represent files have the `EnumerateInstances` method implemented. The reason for this is that it would be very resource intensive to list all the files on the given filesystem. Even more so, for example, all the symlinks on the filesystem. For that reason, every LogicalFile class implements only its `GetInstance` method.

The objectpath of the logical file classes consists of these properties:

- *CSCreationClassName*
- *CSName*
- *FSCreationClassName*
- *FSName*
- *CreationClassName* (*LFCreationClassName* for *LMI_UnixFile*)
- *Name* (*LFName* for *LMI_UnixFile*)

When getting an instance, it's usually required that all of the key properties are specified. However, it is impossible, or at least needlessly complicated, to know some of them when querying remote machines. For example, if I want to see information about the file '/home/user/myfile' on a remote computer, I don't want to specify the filesystem it resides on or the type of the file.

Therefore, the only mandatory key properties are *CSCreationClassName*, *CSName* and *Name* (of *LFName* in case of *LMI_UnixFile*). *FSName*, *FSCreationClassName* and *CreationClassName* are ignored. They are correctly filled in after the instance has been properly returned.

To have an entry point into the Unix filesystems, an association has been added. It binds the computer system and its root directory. See *LMI_RootDirectory*.

LMI_UnixFile has been extended to hold additional properties. Currently, those are *SELinuxCurrentContext* and *SELinuxExpectedContext*. Should there be need for more additions, this class can be easily extended.

Getting files

All further code assumes that a connection object has been created and the default namespace (root/cimv2) is used. Also, the system's instance must have been acquired.

```
# plain http connections will likely be refused
c = connect('https://myhost')
# namespace alias for convenience
ns = c.root.cimv2
system = ns.PG_ComputerSystem.first_instance()
```

Get an instance of the home directory:

```
name_dict =  {'CSCreationClassName':system.classname,
              'CSName':system.name,
              'CreationClassName':'ignored',
              'FSCreationClassName':'ignored',
              'FSName':'ignored',
              'Name':'/home/jsynacek'}
name = ns.LMI_UnixDirectory.new_instance_name(name_dict)
home = name.to_instance()
print home.Name
```

Get an instance of a temporary file and see its selinux contexts using the *LMI_FileIdentity*:

```
name_dict =  {'CSCreationClassName':system.classname,
              'CSName':system.name,
              'LFCreationClassName':'ignored',
              'FSCreationClassName':'ignored',
              'FSName':'ignored',
              'LFName':'/var/tmp/data_file'}
name = ns.LMI_UnixFile.new_instance_name(name_dict)
unixdata = name.to_instance()
data = unixdata.first_associator(AssocClass='LMI_FileIdentity')
print unixdata.SELinuxCurrentContext
print unixdata.SELinuxExpectedContext
print data.Readable
print data.Writeable
print data.Executable
```

Get an instance of a symlink and check where it points to:

```
name_dict =  {'CSCreationClassName':system.classname,
              'CSName':system.name,
              'LFCreationClassName':'ignored',
              'FSCreationClassName':'ignored',
              'FSName':'ignored',
              'LFName':'/home/jsynacek/test-link'}
name = ns.LMI_UnixFile.new_instance_name(name_dict)
unixsymlink = name.to_instance()
symlink = unixsymlink.first_associator(AssocClass='LMI_FileIdentity')
print symlink.TargetFile
```

Association classes examples

List a directory:

```
files = home.associators(AssocClass='LMI_DirectoryContainsFile')
for f in sorted(files, key=lambda x: x.Name):
    print f.Name
```

Get the root directory:

```
root = system.first_associator(AssocClass='LMI_RootDirectory')
print root.Name
```

Note: For a more complex example of how to use the LogicalFile provider, please refer to the OpenLMI LogicalFile script.

Configuration

Configuration is stored in `/etc/openlmi/logicalfile/logicalfile.conf`.

In addition to *common configuration options*, this provider can be configured to allow or deny various filesystem operations. Default configuration:

```
[LMI_UnixDirectory]
# Allow user to create directories. (default = True)
AllowMkdir=True

# Allow user to remove empty directories. (default = True)
AllowRmdir=True

[LMI_SymbolicLink]
# Allow user to create symbolic links. (default = False)
AllowSymlink=False
```

Options and their values are self-explanatory.

3.2.8 Power Management

OpenLMI Power Management Provider allows to manage power states of the managed system. Key functionality is ability to reboot, power off, suspend and hibernate managed system.

This provider is based on following DMTF standard:

- DSP1027 - Power State Management Profile

The knowledge of this standard is not necessary, but it can help a lot.

Table of Contents

Usage

Figure 3.2: Class diagram for Power Management provider.

Base class of this provider is *LMI_PowerManagementService*. This class has method *RequestPowerStateChange* that can be used for changing between power states.

For list of available power states, see property *PowerStatesSupported* of the class *LMI_PowerManagementCapabilities*

All example scripts are for `lmishell`. See it's documentation on OpenLMI page.

We also assume that `lmishell` is connected to the CIMOM and the connection is stored in `connection` variable:

```
connection = connect("server", "username", "password")
ns = connection.root.cimv2
```

Enumeration of available power states

To see the available power states on given managed system, use following:

```
capabilities = ns.LMI_PowerManagementCapabilities.first_instance()
for state in capabilities.PowerStatesSupported:
    print ns.LMI_PowerManagementCapabilities.PowerStatesSupportedValues.value_name(state)
```

Setting the power state

Let's say we want to power off the system gracefully:

```
# Check if the power state is available first
capabilities = ns.LMI_PowerManagementCapabilities.first_instance()
if not ns.LMI_PowerManagementCapabilities.PowerStatesSupportedValues.OffSoftGraceful in capabilities
    print "OffSoftGraceful state is not supported"
    return
# Get the PowerManagement service
service = ns.LMI_PowerManagementService.first_instance()
# Invoke the state change
service.RequestPowerStateChange(PowerState=ns.LMI_PowerManagementCapabilities.PowerStatesSupportedVal
```

Note that the job returned from this function is not much usable because when system is shutting down, the CIMOM is terminated as well.

3.2.9 Realmd Provider

OpenLMI Realmd is a CIM provider for managing the systems Active Direcory or Kerberos realms membership through the Realmd system service.

It provides only the basic functionality: join or leave a domain and query the domain membership.

Contents:

Usage

The OpenLMI Realmd provider allows for basic configuration of the managed systems Active Directory or Kerberos realms membership. It relies on the Realmd system service.

Querying a domain membership

To verify if the remote machine is part of the domain, it is enough to query the value of the *LMI_RealmdService.Domain* property: If non-NULL it contains the name of the joined domain:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
realmsrv = c.root.cimv2.LMI_RealmdService.first_instance()
dom = realmsrv.Domain
if (dom):
    print "Joined to the domain: " + dom
else:
    print "No domain joined."
```

Joining a domain

The *LMI_RealmdService.JoinDomain()* method can be used to join a domain. It takes three mandatory arguments: username and password for the authentication and the domain name:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
realmsrv = c.root.cimv2.LMI_RealmdService.first_instance()
realmsrv.JoinDomain(Password='ZisIzSECRET', User='admin', Domain='AD.EXAMPLE.COM')
```

Leaving a domain

Similarly to joining a domain the *LMI_RealmdService.LeaveDomain()* can be used to leave the joined domain. It requires the same arguments as the *JoinDomain()* method:

```
#!/usr/bin/lmishell
c = connect("localhost", "pegasus", "test")
realmsrv = c.root.cimv2.LMI_RealmdService.first_instance()
realmsrv.LeaveDomain(Password='ZisIzSECRET', User='admin', Domain='AD.EXAMPLE.COM')
```

3.2.10 SELinux Provider

OpenLMI SELinux is a CIM provider which provides a way to read and set SELinux values, such as booleans, ports, or file labels.

The provider doesn't implement any CIM standard schema.

Contents:

Introduction

SELinux provider model is displayed in the following figure. Classes with the blue mark are part of the provider.

Figure 3.3: SELinux provider model

Basic SELinux entities are represented by *LMI_SELinuxElement*. It is a basic class from which concrete SELinux items are derived. All SELinux elements use their *InstanceID* as a primary identifier. Concrete cases are describe below.

LMI_SELinuxBoolean represents an SELinux boolean on a system. Concrete boolean instances are uniquely identified by their *InstanceID* in the form of *LMI:LMI_SELinuxBoolean:<boolean name>*.

LMI_SELinuxPort is a class encompassing multiple individual network ports, or even their ranges. Its *InstanceID* is in the form of *LMI:LMI_SELinuxPort:<type>:<port name>*. Port type can be either *TCP* or *UDP*.

To read SELinux file labels, the *LMI_UnixFile* has to be used. This class is part of the LogicalFile provider.

LMI_SELinuxService is the main class that allows users to modify SELinux state on the system. The class also provides some basic information about SELinux. It is connected to the computer system on which the provider resides by *LMI_HostedSELinuxService*. All instances of *LMI_SELinuxElement* are associated with the service via *LMI_SELinuxServiceHasElement*.

Every method that is provided by *LMI_SELinuxService* returns an *LMI_SELinuxJob* instance, because the actions that are executed by those methods are expected to take a long time. Which of the concrete *LMI_SELinuxElement* instances are operated on by a job instance is determined by *LMI_AffectedSELinuxJobElement*.

Usage

All further code assumes that a connection object has been created and the default namespace (root/cimv2) is used. Also, the LMI_SELinuxService instance must have been acquired.

```
c = connect("https://myhost", "user", "secret")
service = c.root.cimv2.LMI_SELinuxService.first_instance()
system = c.root.cimv2.PG_ComputerSystem.first_instance()
```

As a convenience helper function for further use, *lmi_unixfile_instance_name* is defined. It provides an easy way to get file references for methods that require an LMI_UnixFile reference as a parameter.

```
def lmi_unixfile_instance_name(path):
    props = {"CSName":system.name,
             "CSCreationClassName":system.classname,
             "FSCreationClassName":"ignored",
             "FSName":"ignored",
             "LFCreationClassName":"ignored",
             "LFName":path}
    return c.root.cimv2.LMI_UnixFile.new_instance_name(props)
```

SELinux state

General information about SELinux is available via the *service* instance:

```
def state_to_str(state):
    if state == 0: return "Disabled"
    elif state == 1: return "Permissive"
    elif state == 2: return "Enabled"
    else: return "Unknown"

print "Policy version:   %s" % service.PolicyVersion
print "Policy type:      %s" % service.PolicyType
print "Current state:    %s" % state_to_str(service.SELinuxState)
print "Persistent state: %s" % state_to_str(service.SELinuxDefaultState)
```

Set service state, for example, set the default (persistent) state to Enforcing:

```
# 2 == Enforcing
service.SetSELinuxState({"NewState":2,
                         "MakeDefault":True})
```

Booleans

List all booleans and print their current and default values:

```
booleans = c.root.cimv2.LMI_SELinuxBoolean.instances()
for boolean in booleans:
    print "%-50s (%s, %s)" % (boolean.ElementName, boolean.State, boolean.DefaultState)
```

To enable the *httpd_use_sasl* boolean in the current runtime, but not permanently:

```
target = c.root.cimv2.LMI_SELinuxBoolean.new_instance_name({"InstanceID":"LMI:LMI_SELinuxBoolean:http
res = service.SetBoolean({"Target":target,
                          "Value":True,
                          "MakeDefault":False})
```

Ports

List all ports:

```
ports = c.root.cimv2.LMI_SELinuxPort.instances()
for port in sorted(ports):
    print "%-30s %-10s %s" % (port.ElementName,
                              "tcp" if port.Protocol else "udp",
                              ", ".join(port.Ports))
```

Label the TCP port 8080 with *http_port_t*:

```
target = c.root.cimv2.LMI_SELinuxPort.new_instance_name({"InstanceID":"LMI:LMI_SELinuxPort:TCP:http_p
service.SetPortLabel({"Target":target,
                      "PortRange":"8080"})
```

It is also possible to specify *PortRange* as an actual range, for example “8080-8090”.

File labels

To see what SELinux context a file holds, the LogicalFile provider is used:

```
target = lmi_unixfile_instance_name("/tmp/file")
file = target.to_instance()
print file.SELinuxCurrentContext
print file.SELinuxExpectedContext
```

Set a file context:

```
target = lmi_unixfile_instance_name("/root")
service.SetFileLabel({"Target":target,
                      "Label":"my_user_u:my_role_r:my_type_t"})
```

Restore SELinux contexts of all the files in */etc/* recursively:

```
# 1 == Restore
target = lmi_unixfile_instance_name("/etc/")
service.RestoreLabels({"Target":target,
                       "Action":1,
                       "Recursively":True})
```

3.2.11 Service Provider

OpenLMI Service is CIM provider for managing Linux system services (using the systemd D-Bus interface).

It allows to enumerate system services and get their status, start/stop/restart/... a service and enable/disable a service.

The provider is also able to do event based monitoring of service status (emit indication event upon service property change).

Contents:

Usage

Some common use cases are described in the following parts.

List services

List all services available on managed machine, print whether the service has been started (TRUE), or stopped (FALSE) and print status string of the service:

```
for service in c.root.cimv2.LMI_Service.instances():
    print "%s:\t%s" % (service.Name, service.Status)
```

List only enabled by default services (automatically started on boot). Note that value of EnabledDefault property is '2' for enabled services (and it's '3' for disabled services):

```
service_cls = c.root.cimv2.LMI_Service
for service in service_cls.instances():
    if service.EnabledDefault == service_cls.EnabledDefaultValues.Enabled:
        print service.Name
```

See available information about the 'cups' service:

```
cups = c.root.cimv2.LMI_Service.first_instance({"Name" : "cups.service"})
cups.doc()
```

Start/stop service

Start and stop 'cups' service, see status:

```
cups = c.root.cimv2.LMI_Service.first_instance({"Name" : "cups.service"})
cups.StartService()
print cups.Status
cups.StopService()
print cups.Status
```

Enable/disable service

Disable and enable 'cups' service, print EnabledDefault property:

```
cups = c.root.cimv2.LMI_Service.first_instance({"Name" : "cups.service"})
cups.TurnServiceOff()
print cups.EnabledDefault
cups.TurnServiceOn()
print cups.EnabledDefault
```

Indications

OpenLMI Service provider is able (using indication manager and polling) to emit indication event upon service (i. e. *LMI_Service* instance) property modification (*LMI_ServiceInstanceModificationIndication*).

This is useful mainly for being notified when a service has changed state (has been started, or stopped).

In order to receive indications, create instances of CIM_IndicationFilter (which indications should be delivered), CIM_IndicationHandler (what to do with those indications) and CIM_IndicationSubscription (links filter and handler together).

The following example in LMIShell does it all in one step:

```
c.subscribe_indication(
    Name="service_modification",
    QueryLanguage="DMTF:CQL",
    Query="SELECT * FROM LMI_ServiceInstanceModificationIndication WHERE SOURCEINSTANCE ISA LMI_Servi
    CreationNamespace="root/interop",
    SubscriptionCreationClassName="CIM_IndicationSubscription",
    FilterCreationClassName="CIM_IndicationFilter",
    FilterSystemCreationClassName="CIM_ComputerSystem",
    FilterSourceNamespace="root/cimv2",
    HandlerCreationClassName="CIM_IndicationHandlerCIMXML",
    HandlerSystemCreationClassName="CIM_ComputerSystem",
    Destination="http://localhost:12121"
)
```

Indications are sent to the location specified in 'Destination' argument.

3.2.12 Software Provider

Contents:

Introduction

OpenLMI Software provider allows to query and manipulate software package database on remote hosts. They utilize YUM (Yellowdog Updater Modified) which is a standard package manager for several *GNU/Linux* distributions. They provide the subset of its functionality.

RPM database, repositories and the package manager itself are modeled with *CIM* classes according to several *DMTF* profiles described *later*. To make a query on database, install, update a remove some *RPM* package means to trigger some operation on one or several *CIM* classes. This page explains the mapping of mentioned objects to corresponding classes.

Figure 3.4: This model shows classes representing various objects taking role in software management provided by *OpenLMI Software* provider.

Classes with the blue background belong to *Software Inventory Profile*. Classes painted yellow belong to *Software Update Profile* that builds on the former one. Classes painted red/pink are extensions not beloning to any *DMTF* profile.

Mapping of objects to *CIM* classes

***RPM* package** [*LMI_SoftwareIdentity*] Is represented by `LMI_SoftwareIdentity`. It's identified by a single key property called *LMI_SoftwareIdentity.InstanceID*. This is a composition of some *CIM* related prefix with package's *NEVRA* string. It's the similar string you may see, when listing package with `rpm` tool:

```
$ rpm -qa 'openlmi-*' vim-enhanced
openlmi-python-base-0.3.0_5_gf056906-2.fc21.noarch
openlmi-providers-0.3.0_5_gf056906-2.fc21.x86_64
openlmi-indicationmanager-libs-0.3.0_5_gf056906-2.fc21.x86_64
openlmi-account-0.3.0_5_gf056906-2.fc21.x86_64
openlmi-service-0.3.0_5_gf056906-2.fc21.x86_64
vim-enhanced-7.4.027-2.fc20.x86_64
openlmi-logicalfile-0.3.0_5_gf056906-2.fc21.x86_64
openlmi-storage-0.6.0-2.fc20.noarch
openlmi-python-providers-0.3.0_5_gf056906-2.fc21.noarch
openlmi-providers-debuginfo-0.3.0_5_gf056906-2.fc21.x86_64
openlmi-software-0.3.0_5_gf056906-2.fc21.noarch
```

except for *Epoch* part, which is omitted by `rpm` tool but is required to be present in `InstanceID` by instrumenting provider. To get the expected output, the above command needs to be modified:

```
$ rpm --qf '%{NAME}-%{EPOCH}:%{VERSION}-%{RELEASE}.%{ARCH}\n' -qa 'openlmi-*' | sed 's/(none)/0/
openlmi-python-base-0:0.3.0_5_gf056906-2.fc21.noarch
openlmi-providers-0:0.3.0_5_gf056906-2.fc21.x86_64
openlmi-indicationmanager-libs-0:0.3.0_5_gf056906-2.fc21.x86_64
openlmi-account-0:0.3.0_5_gf056906-2.fc21.x86_64
openlmi-service-0:0.3.0_5_gf056906-2.fc21.x86_64
vim-enhanced-2:7.4.027-2.fc20.x86_64
openlmi-logicalfile-0:0.3.0_5_gf056906-2.fc21.x86_64
openlmi-storage-0:0.6.0-2.fc20.noarch
openlmi-python-providers-0:0.3.0_5_gf056906-2.fc21.noarch
openlmi-providers-debuginfo-0:0.3.0_5_gf056906-2.fc21.x86_64
openlmi-software-0:0.3.0_5_gf056906-2.fc21.noarch
```

Some *RPM* packages do not define *Epoch* part, which means its 0 although `rpm` returns `(none)`.

When installing, updating or removing package, we operate upon an instance or object path of this class.

See also:

Identifying software identity

Repository [*LMI_SoftwareIdentityResource*] Is represented by `LMI_SoftwareIdentityResource`. What distinguishes particular repository from others on the same system is a *LMI_SoftwareIdentityResource.Name* key property. It's the name of repository written in square brackets in repository config. Not the configuration file name, not the `name` option, but a the name of section. See the example of `OpenLMI Nightly` repository:

```
$ cat /etc/yum.repos.d/openlmi-nightly.repo
[openlmi-nightly]
name=OpenLMI Nightly
baseurl=http://openlmi-rnovacek.rhcloud.com/rpm/rawhide/
gpgcheck=0
enabled = 1
```

The `Name` property of corresponding *Software Identity Resource* will be `openlmi-nightly`.

Installed file [*LMI_SoftwareIdentityFileCheck*] Is represented by `LMI_SoftwareIdentityFileCheck`. Represents a verification check of particular file installed by *RPM* package. It contains attributes being checked, like:

- `User ID`, `Group ID`
- `Checksum`
- `Link Target`
- `File Mode` and others

Each is present twice. One property represents the current value of installed file and the other the value stored in *RPM* package, that the file should have. The later properties have `Original` suffix. So for example:

- *UserID* vs *UserIDOriginal*
- *FileChecksum* vs *FileChecksumOriginal*

Mentioned attributes are compared when the package verification is done. Single file can also be easily checked. Either by running *LMI_SoftwareIdentityFileCheck.Invoke()* method on particular object path or by testing the *FailedFlags* property for emptiness. If its empty, the file or directory passed the verification test.

***RPM* database** [*LMI_SystemSoftwareCollection*] Is represented by `LMI_SystemSoftwareCollection`. Administrator probably won't be interested in this class. The *LMI_MemberOfSoftwareCollection* association class associates this collection with available and installed *Software Identities*. It can not be enumerated — due to the same reason as in case of *LMI_SoftwareIdentity* (see the explanation in *Package searching*).

***YUM* package manager** [*LMI_SoftwareInstallationService*] Is represented by `LMI_SoftwareInstallationService`. Allows to query the database, install, update, verify and remove *RPM* packages. All of this can be achieved by invocations of its methods:

FindIdentity() Allows to query the database for matching packages.

InstallFromSoftwareIdentity() Allows to install, update or remove *RPM* package represented by an instance of *Software Identity*.

InstallFromURI() Allows to install or update *RPM* package located with particular URI string.

VerifyInstalledIdentity(). Runs a verification check on given *Software Identity*.

See also:

Examples on using above methods:

- *Package installation*
- *Package update*
- *Package removal*
- *Package verification*

DMTF profiles

OpenLMI Software providers implement two *DMTF* profiles:

- Software Inventory Profile
- Software Update Profile

Software Inventory Profile

Implemented *DMTF* version: `1.0.1`

Described by DSP1023

The Software Inventory Profile describes the CIM schema elements required to provide an inventory of installed BIOS, firmware, drivers, and related software in a managed system. This profile also describes the CIM schema elements required to represent the software that can be installed on a managed system.

Not implemented optional features This implementation does not support:

Representing a Software Bundle Software bundle is represented by LMI_SoftwareIndentity instance having `"Software Bundle"` value present in its `Classifications` property. It shall prepresent software groups. It extends the profile for subclasses of `CIM_OrderedComponent`.

Representing Installation Dependencies Dependencies between software packages are also unimplemented. This also extends the profile for subclasses of `CIM_OrderedDependency` referencing *CIM-SoftwareIdentity* instances.

Deviations

Version Comparison Version comparison is based on different approach than in *Software Inventory Profile* where the following properties are present to uniquely specify package version:

- `uint16` *MajorVersion*
- `uint16` *MinorVersion*
- `uint16` *RevisionNumber*
- `uint16` *BuildNumber*

And also a *VersionString* property which is a composition of previous ones separed with dots.

Unfortunately versioning of RPM packages is incompatible with this scheme. Version of RPM package is composed of following properties:

- `uint32` *Epoch*
- `string` *Version*
- `string` *Release*

Where `Version` and `Release` can contain arbitrary set of characters [17]. These attributes were added to `LMI_SoftwareIdentity` class and will be filled for every RPM package. On the other hand `MajorVersion`, `MinorVersion`, `RevisionNumber` and `BuildNumber` will not be filled.

This implementetion composes `VersionString` in following way:

```
<Epoch>:<Version>-<Release>.<Architecture>
```

The algorithm for comparing two RPM packages version is following:

1. Compare the `Epoch` (which is a number) of both packages. The one with higher epoch is newer. If they match, continue to point 2.
2. Compare their `Version` attributes with rpmvercmp algorithm. Package with larger `Version` (according to rpmvercmp) is newer. If they match, continue to point 3.
3. Compare their `Release` attributes with rpmvercmp algorithm. Package with larger `Release` string is newer. Otherwise packages have the same version.

Relationships between *Software Identity* and *Managed Element* are not modeled. RPM package database does not provide such informations that would allow to associate particular package with a piece of hardware it relates to.

[17] Precisely `Release` must match following regular expression `r"[\\w.+{}]+"`. `Version` allows also tilde character: `r"[~\\w.+{}]+"`.

Querying for packages Since enumeration of *Software Identities* is disabled due to a huge amount of data generated by large package database, the query execution on them is also disallowed [18]. The only way how to search for packages is using the method *LMI_SoftwareInstallationService.FindIdentity*.

Identifying software identity *InstanceID* key property is the one and only identification string of *LMI-SoftwareIdentity* instances representing RPM packages. It's composed of following strings:

```
LMI:LMI_SoftwareIdentity:<Name>-<Epoch>:<Version>-<Release>.<Architecture>
```

Where the prefix `"LMI:LMI_SoftwareIdentity:"` is compared case-insensitively. The rest is also known as a *NEVRA*. When calling `GetInstance()` on this class, the `"<Epoch>:"` part can be omitted in the `InstanceID` key property of passed `InstanceName` in case the epoch is zero.

Example Take for example package `vim-enhanced` installed on Fedora 18:

```
$ yum info vim-enhanced
Installed Packages
Name        : vim-enhanced
Arch        : x86_64
Epoch       : 2
Version     : 7.4.027
Release     : 2.fc18
Size        : 2.1 M
Repo        : installed
From repo   : updates-testing
```

The output has been shortened. This package is represented by an instance of `LMI_SoftwareIdentity` with `InstanceID` equal to:

```
LMI:LMI_SoftwareIdentity:vim-enhanced-2:7.4.027-2.fc18.x86_64
```

Profile extensions List of additional attributes of `LMI_SoftwareIdentity`:

- version properties mentioned above (version_properties)
- `string Architecture` - Target machine architecture. Packages with architecture independent content will have `"noarch"` value set.

List of additional attributes of `LMI_SoftwareIdentityResource`:

`Cost` [sint32] Relative cost of accessing this repository.

`GPGCheck` [boolean] Whether the GPG signature check should be performed.

`TimeOfLastUpdate` [datetime] Time of repository's last update on server.

Class overview

[18] Because internally the query is executed upon the list obtained by enumeration of instances.

Class-name	Parent_class	Type
LMI_SoftwareIdentity	*CIM_SoftwareIdentity*	Plain
LMI_SystemSoftwareCollection	*CIM_SystemSpecificCollection*	Plain
LMI_SoftwareIdentityResource	*CIM_SoftwareIdentityResource*	Plain
LMI_HostedSoftwareCollection	*CIM_HostedCollection*	Association
LMI_InstalledSoftwareIdentity	*CIM_InstalledSoftwareIdentity*	Association
LMI_HostedSoftwareIdentityResource	*CIM_HostedAccessPoint*	Association
LMI_ResourceForSoftwareIdentity	*CIM_SAPAvailableForElement*	Association
LMI_MemberOfSoftwareCollection	*CIM_MemberOfCollection*	Aggregation

See also:

Class model in *Introduction* where above classes are coloured blue.

Software Update Profile

Implemented *DMTF* version: `1.0.0`

Described by DSP1025.

The Software Update Profile describes the classes, associations, properties, and methods used to support the installation and update of BIOS, firmware, drivers and related software on a managed element within a managed system.

Implemented optional features This implementation supports:

Advertising the Location Information of a Software Identity This optional feature provides association of *Software Identity* to its resource. In other words each available package is associated to a corresponding repository defined in configuration files of *YUM*. Repositories are represented with *LMI_SoftwareIdentityResource* and are associated to *LMI_SoftwareIdentity* via *LMI_ResourceForSoftwareIdentity*.

Not implemented features Following methods are not implemented:

- *CIM_SoftwareInstallationService.InstallFromByteStream*
- *LMI_SoftwareInstallationService.CheckSoftwareIdentity*

Profile extensions

RPM package verification *Software Inventory* and *Softare Update* profiles don't allow for software verification. That is quite useful and desired operation done on RPM packages. Following additions has been added to provide such a functionality.

Following classes have been added:

LMI_SoftwareIdentityFileCheck Represents single file contained and installed by *RPM* package. It contains properties allowing for comparison of installed file attributes with those stored in a package database. In case those attributes do not match, file fails the verification test.

LMI_SoftwareIdentityChecks Associates *Software Identity File Check* to corresponding *Software Identity*.

Following methods have been added:

LMI_SoftwareInstallationService.VerifyInstalledIdentity This allows to run verification test on particular *Software Identity* and returns a list of files that failed.

Package searching On modern Linux distributions we have thousands of software packages available for installation making it nearly impossible for *CIMOM* to enumerate them all because it consumes a lot of resources. That's why the `EnumerateInstances()` and `EnumerateInstanceNames()` calls have been disabled *Software Identities*. As a consequence the `ExecQuery()` call is prohibited also.

But the ability to search for packages is so important that a fallback solution has been provided. Method *FindIdentity()* has been added to *LMI_SoftwareInstallationService* allowing to create complex queries on package database.

Class overview

Class-name	Parent_class	Type
LMI_SoftwareInstallationService	*CIM_SoftwareInstallationService*	Plain
LMI_SoftwareJob	*LMI_ConcreteJob*	Plain
LMI_SoftwareInstallationJob	*LMI_SoftwareJob*	Plain
LMI_SoftwareVerificationJob	*LMI_SoftwareJob*	Association
LMI_SoftwareMethodResult	*LMI_MethodResult*	Association
LMI_SoftwareIdentityFileCheck	*CIM_FileSpecification*	Association
LMI_SoftwareInstallationServiceAffectsElement	*CIM_ServiceAffectsElement*	Association
LMI_SoftwareIdentityChecks		Aggregation
LMI_HostedSoftwareInstallationService	*CIM_HostedService*	Plain
LMI_AffectedSoftwareJobElement	*CIM_AffectedJobElement*	Plain
LMI_OwningSoftwareJobElement	*LMI_OwningJobElement*	Plain
LMI_AssociatedSoftwareJobMethodResult	*LMI_AssociatedJobMethodResult*	Plain

See also:

Class model in *Introduction* where above classes are coloured blue.

Configuration

There are various options affecting behaviour of *OpenLMI Software* provider. All of them can be fine-tuned using two configuration files. The main one is located at:

```
/etc/openlmi/software/software.conf
```

The other one is a global configuration file for all providers in *OpenLMI* project and serves as a fallback, for options not specified in the main one. It's located in:

```
/etc/openlmi/openlmi.conf
```

Since this is a common setup for all *OpenLMI* providers, administator can specify options common to all in the global configuration file, while the values specific for particular provider can be overriden in its main one (`/etc/openlmi/${provider}/${provider}.conf`).

Treating boolean values

Options expecting boolean values treat following strings as valid `True` values:

- `True`
- `1`
- `yes`
- `on`

While the following are considered `False`:

- `0`
- `no`
- `False`
- `off`

These words are checked in a case-insensitive way. Any other value isn't considered valid [19].

Options

Follows a list of valid options with sections enclosed in square brackets.

CIM options

[CIM] Namespace [defaults to `root/cimv2`] Is a *CIM* namespace, where *CIM* classes of this provider are registered.

[CIM] SystemClassName [defaults to `PG_ComputerSystem`] Sets the class name used to refer to computer system. Different cimmoms can instrument variously named computer systems and some may not instrument any at all. Sfcb is an example of the later, it needs the `sblim-cmpi-base` package installed providing the basic set of providers containing `Linux_ComputerSystem`. So in case you run a `Sfcb` or you preferr to use providers from `sblim-cmpi-base` package, you need to change this to `Linux_ComputerSystem`.

YUM options Options related to the use of *YUM* API and its configuration.

[Yum] LockWaitInterval [defaults to 0.5] Number of seconds to wait before next try to lock yum package database. This applies, when yum database is locked by another process.

[Yum] FreeDatabaseTimeout = 60 [defaults to 60] Number of seconds to keep package cache in memory after the last use (caused by user request). Package cache takes up a lot of memory.

Log options

[Yum] Level [defaults to `ERROR`] Can be set to one of the following:

- `CRITICAL`
- `ERROR`
- `WARNING`
- `INFO`
- `DEBUG`
- `TRACE_WARNING`
- `TRACE_INFO`
- `TRACE_VERBOSE`

[19] Default value will be used as a fallback. This applies also to other non-boolean options in case of invalid value.

It specifies the minimum severity of messages that shall be logged. Messages having `DEBUG` or more severe level are sent to *CIMOM* using standard function `CMLogMessage()`. Tracing messages (whose level names start with `TRACE_` use the `CMTraceMessage()` instead.

Please consult the documentation of your *CIMOM* to see, how these messages can be treated and logged to different facilities.

Note: This does not have any effect if the `[Log] FileConfig` option is set.

`[Yum] Stderr` [defaults to `False`] Whether to enable logging to standard error output. This does not affect logging to *CIMOM* which stays enabled independently of this option.

This is mostly usefull when debugging with *CIMOM* running on foreground.

Note: This does not have any effect if the `[Log] FileConfig` option is set.

See also:

Since this accepts boolean values, refer to Treating boolean values for details.

`[Yum] FileConfig` [defaults to empty string] This option overrides any other logging option. It provides complete control over what is logged, when and where. It's a path to a logging configuration file with format specified in: http://docs.python.org/2/library/logging.config.html#configuration-file-format Path can be absolute or relative. In the latter case it's relative to a directory of this configuration file.

***YumWorkerLog* options** This section is targeted mostly on developpers of *OpenLMI Software* provider. *YUM* API is accessed exclusively from separated process called `YumWorker`. Because separated process can not send its log messages to *CIMOM*, its logging configuration needs to be configured extra.

`[YumWorkerLog] OutputFile` [defaults to empty string] This is an absolute or relative path to a file, where the logging will be done. Without this option set, logging of `YumWorker` is disabled (assuming the `[YumWorkerLog] FileConfig` option is also unset).

`[YumWorkerLog] Level` [defaults to `DEBUG`] This has generally the same meaning as `Level` in previous section (Log options). Except this affects only logging of `YumWorker` process.

`[YumWorkerLog] FileConfig` [defaults to empty string] Similar to the `FileConfig` option in Log options. This overrides any other option in this section.

Usage

Examples for common use cases listed below are written in lmishell. Where appropriate, an example for `lmi` meta-command, which is a part of *OpenLMI-Scripts* project, is added. Please refer to its documentation for installation notes and usage.

Note: Examples below are written for `openlmi-tools` version `0.9`.

Listing installed packages

Simple Simple but very slow way:

```
c = connect("host", "user", "pass")
cs = c.root.cimv2.PG_ComputerSystem.first_instance()
for identity in cs.associators(
        AssocClass="LMI_InstalledSoftwareIdentity",
        Role="System",
        ResultRole="InstalledSoftware",
        ResultClass="LMI_SoftwareIdentity"):
    print(identity.ElementName)
```

Note: Here we use `PG_ComputerSystem` as a class representing computer system. It is part of `sblim-cmpi-base` package, which is obsoleted. If you use *Pegasus* as your *CIMOM* you may safely switch to `PG_ComputerSystem`.

See also:

LMI_InstalledSoftwareIdentity

Faster This is much faster. Here we enumerate association class *LMI_InstalledSoftwareIdentity* and get information from its key properties.

```
c = connect("host", "user", "pass")
for iname in c.root.cimv2.LMI_InstalledSoftwareIdentity.instance_names():
    print(iname.InstalledSoftware.InstanceID
            [len("LMI:LMI_SoftwareIdentity:"):])
```

Note: Whole instance is not available. To get it from association instance name, you need to add:

```
iname.InstalledSoftware.to_instance()
```

`lmi` meta-command

```
lmi -h $HOST sw list pkgs
```

Listing repositories

lmishell

```
c = connect("host", "user", "pass")
for repo in c.root.cimv2.LMI_SoftwareIdentityResource.instance_names():
    print(repo.Name)
```

See also:

LMI_SoftwareIdentityResource

`lmi` meta-command

```
lmi -h $HOST sw list pkgs
```

Listing available packages

lmishell Enumerating of *LMI_SoftwareIdentity* is disabled due to a huge amount of data being generated. That's why we enumerate them for particular repository represented by *LMI_SoftwareIdentityResource*.

```
c = connect("host", "user", "pass")
for repo in c.root.cimv2.LMI_SoftwareIdentityResource.instances():
    if repo.EnabledState != c.root.cimv2.LMI_SoftwareIdentityResource. \
            EnabledStateValues.Enabled:
        continue                    # skip disabled repositories
    print(repo.Name)
    for identity in repo.associator_names(
            AssocClass="LMI_ResourceForSoftwareIdentity",
            Role="AvailableSAP",
            ResultRole="ManagedElement",
            ResultClass="LMI_SoftwareIdentity"):
        print("  " + identity.InstanceID[len("LMI:LMI_SoftwareIdentity:"):])
```

Note: This is not the same as running:

```
yum list available
```

which outputs all available, not installed packages. The example above yields available packages without any regard to their installation status.

See also:

LMI_ResourceForSoftwareIdentity

`lmi` meta-command

```
lmi -h $HOST sw list --available pkgs
```

Listing files of package

Let's list files of packages `openlmi-tools`. Note that package must be installed on system in order to list its files.

lmishell We need to know exact *NEVRA* [20] of package we want to operate on. If we don't know it, we can find out using *FindIdentity()* method. See example under Searching for packages.

```
c = connect("host", "user", "pass")
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:openlmi-tools-0:0.5-2.fc18.noarch"})
for filecheck in identity.to_instance().associator_names(
        AssocClass="LMI_SoftwareIdentityChecks",
        Role="Element",
        ResultRole="Check",
        ResultClass="LMI_SoftwareIdentityFileCheck"):
    print("%s" % filecheck.Name)
```

See also:

LMI_SoftwareIdentityFileCheck

`lmi` meta-command

```
lmi -h $HOST sw list files openlmi-tools
```

[20] Stands for **N**ame, **E**poch, **V**ersion, **R**elease, **A**rchitecture. Please refer to *Identifying software identity* for more details.

Searching for packages

If we know just a fraction of informations needed to identify a package, we may query package database in the following way.

lmishell

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
# let's find all packages with "openlmi" in Name or Summary without
# architecture specific code
ret = service.FindIdentity(Name="openlmi", Architecture="noarch")
for identity in ret.rparams["Matches"]:
    # we've got only references to instances
    print identity.Name[len("LMI:LMI_SoftwareIdentity:"):]
```

See also:

FindIdentity() method

Please don't use this method to get an instance of package you know precisely. If you know all the identification details, you may just construct the instance name this way:

```
c = connect("host", "user", "pass")
iname = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:openlmi-software-0:0.1.1-2.fc20.noarch"})
identity = iname.to_instance()
```

lmi meta-command See help on `sw` command for more information on this.

```
lmi -h $HOST sw list pkgs openlmi
```

Package installation

There are two approaches to package installation. One is synchronous and the other asynchronous.

Synchronous installation This is a very simple and straightforward approach. We install package by creating a new instance of *LMI_InstalledSoftwareIdentity* with a reference to some available software identity.

```
c = connect("host", "user", "pass")
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
    {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-3.fc19.x86_64"})
cs = c.root.cimv2.PG_ComputerSystem.first_instance_name()
installed_assoc = c.root.cimv2.LMI_InstalledSoftwareIdentity.create_instance(
    properties={
            "InstalledSoftware" : identity,
            "System"            : cs
})
```

If the package is already installed, this operation will fail with the `pywbem.CIMError` exception being raised initialized with `CIM_ERR_ALREADY_EXISTS` error code.

Asynchronous installation Method *InstallFromSoftwareIdentity()* needs to be invoked with desired options. After the options are checked by provider, a job will be returned representing installation process running at background. Please refer to Asynchronous Jobs for more details.

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-5.fc19.x86_64"})
cs = c.root.cimv2.PG_ComputerSystem.first_instance_name()
ret = service.InstallFromSoftwareIdentity(
        Source=identity,
        Target=cs,
        # these options request to install available, not installed package
        InstallOptions=[4]     # [Install]
        # this will force installation if package is already installed
        # (possibly in different version)
        #InstallOptions=[4, 3] # [Install, Force installation]
)
```

The result can be checked by polling resulting job for finished status:

```
finished_statuses = {
      c.root.cimv2.CIM_ConcreteJob.JobState.Completed
    , c.root.cimv2.CIM_ConcreteJob.JobState.Exception
    , c.root.cimv2.CIM_ConcreteJob.JobState.Terminated
    }
job = ret.rparams["Job"].to_instance()
while job.JobStatus not in finished_statuses:
    # wait for job to complete
    time.sleep(1)
    job.refresh()
print c.root.cimv2.LMI_SoftwareJob.JobStateValues.value_name(job.JobState)
# get an associated job method result and check the return value
print "result: %s" % job.first_associator(
        AssocClass='LMI_AssociatedSoftwareJobMethodResult').__ReturnValue
# get installed software identity
installed = job.first_associator(
        Role='AffectingElement',
        ResultRole='AffectedElement',
        AssocClass="LMI_AffectedSoftwareJobElement",
        ResultClass='LMI_SoftwareIdentity')
print "installed %s at %s" % (installed.ElementName, installed.InstallDate)
```

You may also subscribe to indications related to *LMI_SoftwareInstallationJob* and listen for events instead of the polling done above

As you can see, you may force the installation allowing for reinstallation of already installed package. For more options please refer to the documentation of this method.

Combined way We can combine both approaches by utilizing a feature of lmishell. Method above can be called in a synchronous way (from the perspective of script's code). It's done like this:

```
# note the use of "Sync" prefix
ret = service.SyncInstallFromSoftwareIdentity(
        Source=identity,
        Target=cs,
        # these options request to install available, not installed package
        InstallOptions=[4]     # [Install]
        # this will force installation if package is already installed
```

```
        # (possibly in different version)
        #InstallOptions=[4, 3] # [Install, Force installation]
)
print "result: %s" % ret.rval
```

The value of *LMI_SoftwareMethodResult* `.__ReturnValue` is placed to the `ret.rval` attribute. Waiting for job's completion is taken care of by lmishell. But we lose the reference to the job itself and we can not enumerate affected elements (that contain, among other things, installed package).

Installation from URI This is also possible with:

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
cs = c.root.cimv2.PG_ComputerSystem.first_instance_name()
ret = service.to_instance().InstallFromSoftwareURI(
        Source="http://someserver.com/fedora/repo/package.rpm",
        Target=cs,
        InstallOptions=[4])  # [Install]
```

Supported *URI* schemes are:

- `http`
- `https`
- `ftp`
- `file`

In the last cast, the file must be located on the remote system hosting the *CIMOM*.

See also:

InstallFromURI() method

Please refer to Asynchronous installation above for the consequent procedure and how to deal with `ret` value.

`lmi` meta-command

```
lmi -h $HOST sw install sblim-sfcb
```

Package removal

Again both asynchronous and synchronous approaches are available.

Synchronous removal The aim is achieved by issuing an opposite operation than before. The instance of *LMI_InstalledSoftwareIdentity* is deleted here.

```
c = connect("host", "user", "pass")
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-3.fc19.x86_64"})
installed_assocs = identity.to_instance().reference_names(
        Role="InstalledSoftware",
        ResultClass="LMI_InstalledSoftwareIdentity")
if len(installed_assocs) > 0:
    for assoc in installed_assocs:
        assoc.to_instance().delete()
        print("deleted %s" % assoc.InstalledSoftware.InstanceID)
```

```
else:
    print("no package removed")
```

Asynchronous removal

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-5.fc19.x86_64"})
cs = c.root.cimv2.PG_ComputerSystem.first_instance_name()
ret = service.InstallFromSoftwareIdentity(
        Source=identity,
        Target=cs,
        InstallOptions=[9])  # [Uninstall]
```

Again please refer to Asynchronous installation for examples on how to deal with the `ret` value.

`lmi` meta-command

```
lmi -h $HOST sw remove sblim-sfcb
```

Package update

Only asynchronous method is provided for this purpose. But with the possibility of synchronous invocation.

`lmishell` Example below shows the synchronous invocation of asynchronous method.

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-5.fc19.x86_64"})
cs = c.root.cimv2.PG_ComputerSystem.first_instance_name()
ret = service.SyncInstallFromSoftwareIdentity(
        Source=identity,
        Target=cs,
        InstallOptions=[5]       # [Update]
        # to force update, when package is not installed
        #InstallOptions=[4, 5]   # [Install, Update]
)
print "installation " + ("successful" if rval == 0 else "failed")
```

`lmi` meta-command

```
lmi -h $HOST sw update sblim-sfcb
```

Package verification

Installed *RPM* packages can be verified. Attributes of installed files are compared with those stored in particular *RPM* package. If some value of attribute does not match or the file does not exist, it fails the verification test. Following attributes come into play in this process:

- File size - in case of regular file
- User ID

- Group ID
- Last modification time
- Mode
- Device numbers - in case of device file
- Link Target - in case the file is a symbolic link
- Checksum - in case of regular file

lmishell It's done via invocation of *VerifyInstalledIdentity()*. This is an asynchronous method. We can not use synchronous invocation if we want to be able to list failed files.

```
c = connect("host", "user", "pass")
service = c.root.cimv2.LMI_SoftwareInstallationService.first_instance()
identity = c.root.cimv2.LMI_SoftwareIdentity.new_instance_name(
        {"InstanceID" : "LMI:LMI_SoftwareIdentity:sblim-sfcb-0:1.3.16-5.fc19.x86_64"})
results = service.VerifyInstalledIdentity(
        Source=identity,
        Target=ns.PG_ComputerSystem.first_instance_name())
nevra = (    identity.ElementName if isinstance(identity, LMIInstance)
        else identity.InstanceID[len('LMI:LMI_SoftwareIdentity:'):])
if results.rval != 4096:
    msg = 'failed to verify identity "%s (rval=%d)"' % (nevra, results.rval)
    if results.errorstr:
        msg += ': ' + results.errorstr
    raise Exception(msg)

job = results.rparams['Job'].to_instance()

# wait by polling or listening for indication
wait_for_job_finished(job)

if not LMIJob.lmi_is_job_completed(job):
    msg = 'failed to verify package "%s"' % nevra
    if job.ErrorDescription:
        msg += ': ' + job.ErrorDescription
    raise Exception(msg)

# get the failed files
failed = job.associators(
        AssocClass="LMI_AffectedSoftwareJobElement",
        Role='AffectingElement',
        ResultRole='AffectedElement',
        ResultClass='LMI_SoftwareIdentityFileCheck')
for iname in failed:
    print iname.Name    # print their paths
```

Polling, as a way of waiting for job completion, has been already shown in the example under Asynchronous installation.

See also:

LMI_SoftwareIdentityFileCheck

`lmi` meta-command

```
lmi -h $HOST sw verify sblim-sfcb
```

Enable and disable repository

lmishell

```
c = connect("host", "user", "pass")
repo = c.root.cimv2.LMI_SoftwareIdentityResource.first_instance_name(
        key="Name",
        value="fedora-updates-testing")
# disable repository
repo.to_instance().RequestStateChange(
    RequestedState=c.root.cimv2.LMI_SoftwareIdentityResource. \
        RequestedStateValues.Disabled)
repo = c.root.cimv2.LMI_SoftwareIdentityResource.first_instance_name(
        key="Name",
        value="fedora-updates")
# enable repository
repo.to_instance().RequestStateChange(
    RequestedState=c.root.cimv2.LMI_SoftwareIdentityResource. \
        RequestedStateValues.Enabled)
```

lmi meta-command

```
lmi -h $HOST sw disable fedora-updates-testing
lmi -h $HOST sw enable fedora-updates
```

Supported event filters

There are various events related to asynchronous job you may be interested about. All of them can be subscribed to with static filters presented below. Usage of custom query strings is not supported due to a complexity of its parsing. These filters should be already registered in *CIMOM* if *OpenLMI Software* providers are installed. You may check them by enumerating `LMI_IndicationFilter` class located in `root/interop` namespace. All of them apply to two different software job classes you may want to subscribe to:

LMI_SoftwareInstallationJob Represents a job requesting to install, update or remove some package.

LMI_SoftwareVerificationJob Represents a job requesting verification of installed package.

Filters below are written for *LMI_SoftwareInstallationJob* only. If you deal with the other one, just replace the class name right after the `ISA` operator and classname in filter's name.

Percent Updated Indication is sent when the *LMI_SoftwareJob.PercentComplete* property of a job changes.

```
SELECT * FROM LMI_SoftwareInstModification WHERE
    SourceInstance ISA LMI_SoftwareInstallationJob AND
    SourceInstance.CIM_ConcreteJob::PercentComplete <>
    PreviousInstance.CIM_ConcreteJob::PercentComplete
```

Registered under filter name `"LMI:LMI_SoftwareInstallationJob:PercentUpdated"`.

Job state change Indication is sent when the *LMI_SoftwareJob.JobState* property of a job changes.

```
SELECT * FROM LMI_SoftwareInstModification WHERE
    SourceInstance ISA LMI_SoftwareInstallationJob AND
    SourceInstance.CIM_ConcreteJob::JobState <>
    PreviousInstance.CIM_ConcreteJob::JobState
```

Registered under filter name "LMI:LMI_SoftwareInstallationJob:Changed".

Job Completed This event occurs when the state of job becomes COMPLETED/OK [21].

```
SELECT * FROM LMI_SoftwareInstModification WHERE
    SourceInstance ISA LMI_SoftwareInstallationJob AND
    SourceInstance.CIM_ConcreteJob::JobState = 17
```

Registered under filter name "LMI:LMI_SoftwareInstallationJob:Succeeded".

Error This event occurs when the state of job becomes COMPLETED/Error [22].

```
SELECT * FROM LMI_SoftwareInstModification WHERE
    SourceInstance ISA LMI_SoftwareInstallationJob AND
    SourceInstance.CIM_ConcreteJob::JobState = 10
```

Registered under filter name "LMI:LMI_SoftwareInstallationJob:Failed".

New Job This event occurs when the new instance of *LMI_SoftwareJob* is created.

```
SELECT * FROM LMI_SoftwareInstCreation WHERE
     SourceInstance ISA LMI_SoftwareInstallationJob
```

Registered under filter name "LMI:LMI_SoftwareInstallationJob:Created".

3.2.13 SSSD Provider

OpenLMI SSSD is a CIM provider for managing the System Security Services Daemon.

It provides only the basic functionality: managing SSSD components and providing information about active domains.

Contents:

3.2.14 Storage Provider

Overview

OpenLMI-Storage is a CIM provider which manages storage on a Linux machine. It exposes remotely accessible object-oriented API using WBEM set of protocols and technologies.

[21] This is a composition of values in *OperationalStatus* array. It corresponds to value Completed of *JobState* property.
[22] This is a composition of values in *OperationalStatus* array. It corresponds to value Exception of *JobState* property.

Clients

The API can be accessed by any WBEM-capable client. OpenLMI already provides:

- Python module *lmi.scripts.storage*, part of *OpenLMI scripts*.
- Command line tool: *LMI metacommand*, with *'storage'* subcommand.

Features

- Enumerate all block devices.
- Partition a block device.
- Manage MD RAID and LVM.
- Format a block device with a filesystem (xfs, ext2/3/4, ...)
- Manage mounts.

Currently, OpenLMI-Storage manages local block devices, i.e. block devices which are present in `/dev/` directory. This includes also attached iSCSI, FC and FCoE devices, as long as appropriate block device is present.

In future, it may include configuration of iSCSI and FC initiators, multipath and other remote-storage management.

Examples

There is plenty of examples how to use OpenLMI-Storage provider remotely from LMIShell:

- *Create a partition table on a device*.
- *Create a new partition*.
- *Create software RAID5 with 3 devices*.
- *Format a device with ext3 filesystem*.
- *Mount a filesystem*.

Documentation

The provider is inspired by SNIA SMI-S, but it differers in several important areas. Application developers who are familiar with SMI-S should read *SMI-S profiles* chapter.

Application developers and/or sysadmins should skip whole SMI-S chapter and start at *OpenLMI-Storage concept*.

Table of contents

SMI-S profiles

This chapter lists SMI-S profiles implemented by OpenLMI-Storage. The implementation does not follow SMI-S strictly and deviates from it where SMI-S model cannot be used. Each such deviation is appropriately marked.

OpenLMI-Storage implements following profiles:

SMI-S Disk Partition Subprofile

Profile adjustment The Disk Partition Subprofile does not reflect real-world MBR partition tables:

- The profile specifies, there can be up to 4 primary partitions (correct), one of them can be extended (correct) and up to 4 logical partitions can be instantiated on this extended partition (wrong, number of logical partitions is not limited).
- The profile specifies that logical partition metadata is on the beginning of the extended partition (see Figure 7 in the profile). In reality, each logical partition has its own metadata sector just before the partition. In addition, there can be number of empty sectors between the logical partition metadata and the partition beginning, which are left as result of alignment rules.

As result of this deficiency, some adjustments were necessary:

- The *LMI_DiskPartition* representing a logical partition *includes* the metadata sector and any alignment sectors.
- *NumberOfBlocks* property *includes* the metadata and any alignment sectors.
- *ConsumableBlocks* includes only the real usable data on partition.

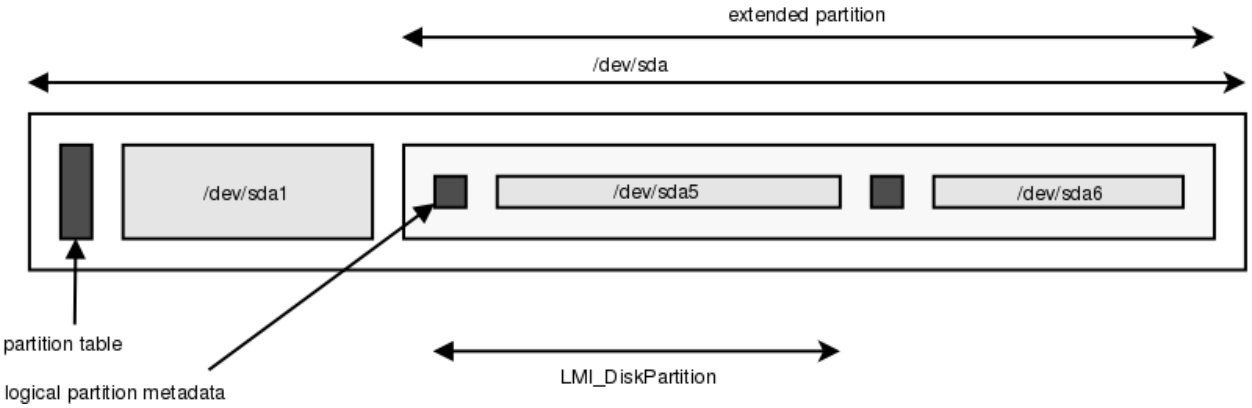

Figure 3.5: Correct overview of logical partitions.

GPT partition tables do not have these issues and are generally preferred over MBR ones.

Implementation All mandatory classes are implemented. However, *CreateOrModifyPartition* method is *not* implemented. This function might be added in future.

The only way, how to create partitions is proprietary *LMI_CreateOrModifyPartition*, which fits actual partitioning better.

Classes Implemented SMI-S classes:

- *LMI_PartitionBasedOn*
- *LMI_DiskPartition*
- *LMI_DiskPartitionConfigurationCapabilities*
- *LMI_DiskPartitionConfigurationService*
- *LMI_DiskPartitionElementCapabilities*
- *LMI_GenericDiskPartition*
- *LMI_InstalledPartitionTable*

- *LMI_StorageExtent*

Additional implemented classes:

- *LMI_DiskPartitionConfigurationSetting*
- *LMI_DiskPartitionElementSettingData*

Not implemented classes:

- *CIM_GPTDiskPartition*
- *CIM_LogicalDisk*
- *CIM_VTOCDiskPartition*
- *CIM_SystemDevice*
- *CIM_HostedService*

Methods Implemented:

- *SetPartitionStyle*
- *LMI_CreateOrModifyPartition*

Not implemented:

- *CreateOrModifyPartition*

Warning: Mandatory indications are not implemented.
Anaconda does not provide such functionality and it would be very CPU-intensive to periodically scan for new/deleted partitions.

SMI-S Block Services Package

This package is core of SMI-S. It describes how devices (disks) are grouped together into pools with different capabilities and even hierarchy of pools can be built.

> A StoragePool is a storage element; its storage capacity has a given set of capabilities. Those 'StorageCapabilities' indicate the 'Quality of Service' requirements that can be applied to objects created from the StoragePool.

Storage on Linux does not use pool concept except Volume Groups, therefore we allow to create storage devices directly from other storage devices, e.g. create MD RAID from partitions.

Primordial pool At the lowest level of hierarchy of SMI-S storage pools are primordial devices and pools.

> A primordial StoragePool is a type of StoragePool that contains unformatted, unprepared, or unassigned capacity. Storage capacity is drawn from the primordial StoragePool to create concrete StoragePools. A primordial StoragePool aggregates storage capacity not assigned to a concrete StoragePool. StorageVolumes and LogicalDisks are allocated from concrete StoragePools.
>
> At least one primordial StoragePool shall always exists on the block storage system to represent the unallocated storage on the storage device.

OpenLMI-Storage uses raw disk as primordial. Everything else (partitions, RAIDs, logical volumes, ...) are not primordial.

Logical disks In SMI-S, only LogicalDisks instances can be used by the OS. I.e. if an admin wants to build a filesystem e.g. on RAIDCompositeExtent, in SMI-S it's necessary to allocate a LogicalDisk from it.

We find this approach useless and we don't allocate LogicalDisks for devices, which can be used by the OS. In fact, any block device can be used by the OS, therefore it would make sense to make `LMI_StorageExtent` as subclass of `CIM_LogicalDisk`.

Implementation

Classes Implemented SMI-S classes:

- *LMI_VGAssociatedComponentExtent*
- *LMI_MDRAIDBasedOn*
- *LMI_LVBasedOn*
- *LMI_LVAllocatedFromStoragePool*
- *LMI_LVElementCapabilities*
- *LMI_VGElementCapabilities*
- *LMI_MDRAIDElementCapabilities*
- *LMI_MDRAIDElementSettingData*
- *LMI_LVElementSettingData*
- *LMI_VGElementSettingData*
- *LMI_StorageExtent*
- *LMI_LVStorageExtent*
- *LMI_MDRAIDStorageExtent*
- *LMI_StorageConfigurationService*
- *LMI_VGStoragePool*
- *LMI_VGStorageCapabilities*
- *LMI_LVStorageCapabilities*
- *LMI_MDRAIDStorageCapabilities*
- *LMI_VGStorageSetting*
- *LMI_MDRAIDStorageSetting*
- *LMI_LVStorageSetting*

Methods Implemented:

- *CreateOrModifyStoragePool* (creates Volume Group from list of block devices).
- *CreateOrModifyElementFromElements* (creates MD RAID from list of block devices).
- *CreateOrModifyElementFromStoragePool* (creates logical Volumes from a Volume Group).
- *CreateOrModifyMDRAID*
- *CreateOrModifyVG*
- *CreateOrModifyLV*

Warning: Mandatory indications are **not** implemented.

SMI-S Extent Composition Subprofile

This profile provides lot of examples how to create various RAID levels and how to composite hierarchy of StoragePools in general. It does not introduce any new method or class.

SMI-S File Storage Profile

This profile is fully implemented. See the next chapter for its usage and mapping to `LMI_` classes.

SMI-S Filesystem Profile

OpenLMI-Storage implements the Filesystem Profile with these adjustments:

- Local Access is not implemented, we use LMI_MountService to mount local filesystems:
  - SMI-S expects that one filesystem can be mounted only once using Local Access, which is not true on Linux, we might mount one filesystem multiple times.
  - Mounting a filesystem is totally different operation to creating/modifying of a filesystem, these two functions should be separated. Therefore we introduce LMI_MountService to mount various filesystems.
- Directory Services are not implemented.

Implementation All mandatory classes and methods are implemented.

Classes Implemented SMI-S classes:

- *LMI_FileSystemSetting*
- *LMI_FileSystemElementSettingData*
- *LMI_HostedFileSystem*
- *LMI_LocalFileSystem*
- *CIM_LogicalFile* using separate *LogicalFile provider* from OpenLMI-Providers package.

Not implemented classes:

- `'CIM_FileStorage`
- `SNIA_LocalAccessAvailable`
- `SNIA_LocalFileSystem`
- `SNIA_LocallyAccessibleFileSystemSetting`
- and all related references.

Methods There are no methods in this profile.

> **Warning:** Mandatory indications are not implemented.
> Blivet does not provide such functionality and it would be very CPU-intensive to periodically scan for modified filesystems.

SMI-S Filesystem Manipulation Profile

OpenLMI-Storage implements the Filesystem Profile with these adjustments:

- Local Access is not implemented, we use LMI_MountService to mount local filesystems:
  - SMI-S expects that one filesystem can be mounted only once using Local Access, which is not true on Linux, we might mount one filesystem multiple times.
  - Mounting a filesystem is totally different operation to creating/modifying of a filesystem, these two functions should be separated.
- Directory Services are not implemented.

Implementation SNIA-specific classes and methods (with `SNIA_` prefix) are not implemented to avoid any copyright problems - SNIA MOF files have a license which does not allow us to implement it in open source project.

We implement our `LMI_` counterparts, inspired by CIM_StorageService and CIM_StorageSetting. The major difference to `CIM_` and `SNIA_FileSystemConfigurationService` is that all methods accepts a Setting argument as reference and not as embedded instance to match the rest of the methods (mainly in Block Services profile).

Classes Implemented SMI-S classes:

- *LMI_FileSystemConfigurationElementCapabilities*
- *LMI_FileSystemElementSettingData*
- *LMI_HostedFileSystem*
- *LMI_HostedStorageService*
- *LMI_FileSystemCapabilities*
  - not derived from `SNIA_FileSystemCapabilities`!
- *LMI_FileSystemConfigurationCapabilities*
  - not derived from `SNIA_FileSystemConfigurationCapabilities`!
- *LMI_FileSystemConfigurationService*
  - not derived from `SNIA_FileSystemConfigurationService`!
- *LMI_FileSystemSetting*
  - not derived from `SNIA_FileSystemSetting`!
- *LMI_LocalFileSystem*
  - not derived from `SNIA_LocalFileSystem`!

Not implemented classes:

- `SNIA_FileSystemCapabilities`
- `SNIA_FileSystemConfigurationCapabilities`

- SNIA_FileSystemConfigurationService
- SNIA_FileSystemSetting
- SNIA_LocalFileSystem
- SNIA_LocalAccessAvailable
- SNIA_LocallyAccessibleFileSystemCapabilities
- SNIA_LocallyAccessibleFileSystemSetting
- and all related references.

Methods Implemented:

- *LMI_CreateSetting*
- *LMI-CreateFileSystem*
  - Similar to plain CIM CreateFileSystem, with these modifications:
    * Goal parameters is passed as reference and not as embedded instance, i.e. all *LMI_FileSystemSetting* instances reside on server and are created using *LMI_CreateSetting*
    * Multiple extents can be passed in InExtents parameter. The method then creates one filesystem on multiple devices. Currently only btrfs supports this behavior, other filesystems can be created only on one device.
- *DeleteFileSystem*

Not implemented:

- CreateGoalSettings
- GetRequiredStorageSize
- SNIA_CreateFileSystem
- SNIA_ModifyFileSystem
- *CreateFileSystem*
- *ModifyFileSystem*

Warning: Mandatory indications are not implemented.
Blivet does not provide such functionality and it would be very CPU-intensive to periodically scan for modified filesystems.

SMI-S Job Control Subprofile

OpenLMI-Storage implements the Job Control Subprofile with these adjustments:

- All indications are implemented, however the CQL query is different. SMI-S uses optional CQL extensions, e.g. ANY keyword, and our CIMOMs do not support that. Therefore all the CQL queries for OperationalStatus[*] were reworked to use JobState property.

Implementation All mandatory classes and methods are implemented.

Classes Implemented SMI-S classes:

- *LMI_AffectedStorageJobElement*
- *LMI_AssociatedStorageJobMethodResult*
- *LMI_StorageJob*
- *StorageMethodResult*
- *LMI_OwningStorageJobElement*

Methods

- *GetErrors*
- *GetError*
- *RequestStateChange*

Indications See list of indications in *Jobs* chapter.

SMI-S Block Server Performance Subprofile

This profile provides I/O statistics for various *CIM_StorageExtent* subclasses.

OpenLMI-Storage implements the Block Server Performance Subprofile with these adjustments:

- Applications cannot create custom manifests, i.e. *LMI_BlockStatisticsService.AddOrModifyManifest* is not implemented.
- We provide *LMI_BlockStorageStatisticalData* for *every CIM_StorageExtent* subclass and not only for disk drives. *LMI_BlockStorageStatisticalData.ElementType* property is always set to `9`, i.e. `Extent`.
- There is no sampling interval. OpenLMI always reports current values when returning *LMI_BlockStorageStatisticalData* instance.

Note: Even though properties in *LMI_BlockStorageStatisticalData* are 64-bit, they are tracked as 32-bit on systems with 32-bit kernel. They can wrap pretty quickly on modern hardware.

For example, on i686 with iSCSI drive on 10Gb/s link, the KBytesRead counter can wrap in approximately 27 minutes.

With 64-bit kernels, these counters are tracked in 64-bits and they wrap once in a few years.

Implementation All mandatory classes and methods are implemented.

Classes Implemented SMI-S classes:

- *LMI_BlockStorageStatisticalData*
- *LMI_StorageElementStatisticalData*
- *LMI_StorageStatisticsCollection*
- *LMI_MemberOfStorageStatisticsCollection*
- *LMI_HostedStorageStatisticsCollection*
- *LMI_BlockStatisticsService*

- *LMI_BlockStatisticsCapabilities*
- *LMI_BlockStatisticsManifest*
- *LMI_BlockStatisticsManifestCollection*
- *LMI_MemberOfBlockStatisticsManifestCollection*
- *LMI_AssociatedBlockStatisticsManifestCollection*

Methods Implemented methods:

- *LMI_BlockStatisticsService.GetStatisticsCollection*

The OpenLMI-Storage CIM API follows following principles:

- Each block device is represented by exactly one *CIM_StorageExtent*.
  - For example RAID devices are created using *LMI_StorageConfigurationService*. *CreateOrModifyElementFromElements*, without any pool being involved.
  - No *CIM_LogicalDisk* is created for devices consumed by the OS, i.e. when there is a filesystem on them.
  - Actually, all block devices can be used by the OS and it might be useful to have *LMI_StorageExtent* as subclass of *CIM_LogicalDisk*.

Warning: This violates SMI-S, each block device should have **both** a StorageExtent + LogicalDisk associated from it to be usable by the OS.

- *CIM_StoragePool* is used only for real pool objects - volume groups.
- PrimordialPool is not present. It might be added in future to track unused disk drives and partitions.

The implementation is not complete, e.g. mandatory Server Profile is not implemented at all. The list will get updated.

Storage API concept

OpenLMI-Storage provides CIM API. Some CIM knowledge is required and this guide assumes that reader can routinely read and modify remote CIM objects and call their intrinsic and extrinsic methods.

No SMI-S knowledge is necessary, but it can help a lot.

CIM API concepts

Storage API is based on several design patterns, which are common in CIM and SMI-S.

Separation of state and configuration If *foo* is configurable, CIM uses two classes to describe it:

- `CIM_Foo`: *state* of foo.
- `CIM_FooSetting`: *configuration* of foo.

That means, each *foo* on managed system is represented by one `CIM_Foo` instance and one `CIM_FooSetting` instance. They are connected together using `CIM_FooElementSettingData` association instance.

If there is no `CIM_FooSetting instance` for a `CIM_Foo`, it indicates that the *foo* is not configurable.

For example, a local filesystem is represented by:

- one instance of `CIM_LocalFileSystem`, which contains *state* of the filesystem – nr. of inodes, nr. of free inodes, total space on the filesystem, free space, etc.
- one instance of `CIM_LocalFileSystemSetting`, which contains *configuration* of the filesystem – inode size, journal size, ...

Sometimes, state and configuration overlap. In our filesystem example, `BlockSize` is property of both `CIM_LocalFileSystem` and `CIM_LocalFileSystemSetting`. Logically, the `BlockSize` should be only in `CIM_LocalFileSystemSetting`. But if a filesystem was not configurable, there would be no `CIM_LocalFileSystemSetting` for it and therefore any management application would not have access to its `BlockSize`, which is important feature of the filesystem.

Configuration service In CIM world, managed elements cannot be configured directly by editing the associated `CIM_FooSetting` with the configuration of *foo*. Instead, there is `CIM_FooConfigurationService` singleton, which has method to create, modify and sometimes also delete *foos*.

Change of configuration If an application want to change configuration of a *foo*, it must create new auxiliary `CIM_FooSetting` instance with requested new configuration and associate this new `CIM_FooSetting` with the `CIM_Foo` it wants to configure. The application does not need to completely fill the auxiliary `CIM_FooSetting`, in most cases it is enough to edit only the properties that it wants to change, the rest of properties can be `NULL`.

For example, to change `CIM_LocalFileSystemSetting` of a `CIM_LocalFileSystem`, the application must create new `CIM_LocalFileSystemSetting`, fill its properties it wants to change and then call `CIM_FileSystemConfigurationService.SNIA_ModifyFileSystem()` method.

The auxiliary `CIM_LocalFileSystemSetting` created by the application can be reused by the application to change configuration of different `CIM_LocalFileSystem` instances.

Creation of instances The `CIM_FooSetting` is also used to create new objects. If an application wants to create new *foo*, it creates new auxiliary `CIM_FooSetting`, which describes configuration of the *foo* to create. The application can then call specific API method to create the *foo* and new `CIM_Foo` is created, with its own associated `CIM_FooSetting`. The associated `CIM_FooSetting` is basically a copy of the auxiliary CIM_FooSetting created by the application. Therefore the application can reuse one auxiliary `CIM_FooSetting` instance to create or modify multiple *foos*.

For example, to create a filesystem on a block device, the application must create `CIM_LocalFileSystemSetting`, set its properties as it wants and call `CIM_FileSystemConfigurationService.SNIA_CreateFileSystem`.

Capabilities The DMTF and SMI-S describe various methods and configuration properties of various classes. Implementations of the standards can implement only some of these methods and properties. Therefore `CIM_FooConfigurationCapabilities` describes what methods and kinds of *foo* our implementation of `CIM_FooConfigurationService` supports.

For example, if our `CIM_FileSystemConfigurationService` supports xfs and ext3 filesystems and only `SNIA_CreateFileSystem` and `SNIA_ModifyFileSystem` method calls, it will be reflected in its associated `CIM_FileSystemConfigurationCapabilities`.

In addition, if there are several different kind of *foos* supported by the implementation, each such kind can have its own `CIM_FooCapabilities` instance to describe all available configuration options and their value ranges.

For example, if our `CIM_FileSystemConfigurationService` is able to create xfs and ext3 filesystems, there are two `CIM_LocalFileSystemCapabilities` instances, one for xfs and the second for ext3. The xfs-related

instance describes valid inode sizes for xfs, while the ext3-related instance describes valid inode sizes for ext3. Since we can subclass CIM_LocalFileSystemCapabilities, the xfs-related instance can have additional xfs-specific properties and so can have also the ext3-related instance.

The supported properties and their ranges can be either defined directly in the CIM_FooCapabilities (which is the most common case) or using CIM_FooSetting attached to CIM_FooCapabilities using CIM_SettingsDefineCapabilities association. The associated CIM_FooSetting can then define minimum, maximum or default values of the configuration properties. Consult DMTF description of CIM_SettingsDefineCapabilities association in this case.

This is the case of filesystem configuration, the capabilities of xfs and ext3 filesystem is defined using CIM_LocalFileSystemSetting.

Figure 3.6: Example CIM_FileSystemConfigurationService with capabilities and settings, which define the capabilities.

There are slight variations on this concept across DMTF and SMI-S profiles as the standards evolved, sometimes are CIM_FooConfigurationCapabilities and CIM_FooCapabilities merged into one class, sometimes the capabilities are associated directly to managed elements, sometimes the capabilities as defined using setting instances etc. Still, the concept is the same - *capabilities* define what configuration options are supported by the implementation and its valid values or value ranges. Different implementations will have different capabilities. *Setting* instances then describe specific configuration of one managed element.

Predefined configurations To simplify management applications, the implementation can provide several CIM_FooSetting instances for the most typical *foo* configurations. These instances are associated to CIM_FooCapabilities. Application then does not need to manually create auxiliary CIM_FooSetting instance and fill its properties, it can directly use the preconfigured ones.

For example, an implementation can provide one typical CIM_LocalFileSystemSetting instance for generic xfs filesystem and one CIM_LocalFileSystemSetting instance for xfs filesystem tuned for Gluster, which needs larger inode size for better performance.

Document conventions

Throughout this document we use following conventions.

Examples All example scripts are for lmishell. See it's documentation on OpenLMI page.

We also assume that following script has been run to connecto to a CIMOM and initialize basic variables:

```
MEGABYTE = 1024*1024
connection = connect("localhost", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()
partitioning_service = ns.LMI_DiskPartitionConfigurationService.first_instance()
filesystem_service = ns.LMI_FileSystemConfigurationService.first_instance()
encryption_service = ns.LMI_ExtentEncryptionConfigurationService.first_instance()
```

Device hierarchy

The API manages all block devices in machine's local /dev/ directory, i.e. also remote disks (iSCSI, FcoE, ...), as long as there is appropriate device in local /dev/.

The API exposed by OpenLMI-Storage is object-oriented. Each block device present on the managed system is represented as instance of *CIM_StorageExtent* class. The instance has properties like DeviceID, Name, BlockSize and NumberOfBlocks, which describe the block device.

CIM_StorageExtent has several subclasses, such as *LMI_DiskPartition* (=MS DOS partition) or *LMI_LVStorageExtent* (=Logical Volume), which add properites specific for the particular device type.

Each block device is represented by instance of *CIM_StorageExtent* or its subclasss.

LMI_StorageExtent represents all devices, which do not have any specific *CIM_StorageExtent* subclass.

Each volume group is represented by *LMI_VGStoragePool*.

Instances of *LMI_VGStoragePool*, *CIM_StorageExtent* and its subclasses compose an oriented graph of devices on the system. Devices are connected with these associations or their subclasses:

- *CIM_BasedOn* and is subclasses associates a block device to all devices, on which it directly depends on, for example a partition is associated to a disk, on which it resides, and MD RAID is associated to all underlying devices, which compose the RAID.
- *LMI_VGAssociatedComponentExtent* associates volume groups with its physical extents.
- *LMI_LVAllocatedFromStoragePool* associates logical volumes to their volume groups.

Figure 3.7: Example of two logical volumes allocated from volume group created on top of MD RAID with three devices.

All other storage objects, like partition tables, filesystems and mounts are designed in similar way - all these are instances of particular classes.

These storage elements are managed (i.e. created / modified and deleted) by subclasses of *CIM_Service* such as *LMI_FileSystemConfigurationService*. These services are not system services in systemd or UNIX SysV sense, it is just API collecting bunch of methods related to a particular topic, e.g. filesystem management in our example.

These services are described in *OpenLMI-Storage API* chapter.

Device identification

On modern Linux, block devices can be identified in number of ways. Some of them are stable across reboots, some other are nice to remeber and it is also possible to configure block device names using udev rules.

For example, all these paths refer to the same block device:

- `/dev/disk/by-id/ata-Samsung_SSD_840_Series_S19MNSAD500335K`
- `/dev/disk/by-id/wwn-0x50025385a0031e7c`
- `/dev/sda`
- `/dev/systemdisk` (using an udev rule)

OpenLMI does not assume any site policy, it's up to system administrator to write udev rules if default /dev/sdX and /dev/disk/by-id/XYZ is not sufficient.

As many things in Linux are configurable and tunable, term *SHOULD* below means *unless explicitly reconfigured*.

CIM_StorageExtent When OpenLMI builds *CIM_StorageExtent* for a block device, it fills following properties:

DeviceID

OpenLMI internal identifier of a block device. Even if it looks like a device path, it should be opaque for applications and applications should not parse it / interpret it in any way. Its format may change in future versions of OpenLMI.

This is the primary key how to identify a *CIM_StorageExtent*.

- Guaranteed to be unique in the managed system.
- SHOULD be persistent across reboots.

InstanceID

OpenLMI internal identifier of a block device group. This property has been added to have the same way how to identify *CIM_StorageExtent* and *LMI_VGStoragePool*.

- Guaranteed to be unique in the managed system.
- SHOULD be persistent across reboots.

Name

Canonical path to the device, such as as `/dev/sda`, `/dev/mapper/test-test1`, `/dev/md/blivet00`. This is the Linux default device name.

- Guaranteed to be unique in the managed system.
- Not persistent across reboots.

ElementName

Name of the block device, logical volume, RAID etc, such as as `sda` for disk, `test1` for logical volume, `blivet00` for MD RAID.

- Not unique in the managed system.
- Not persistent across reboots.
- Usually assigned by system administrator when the device is created (logical volume, MD RAID, ...)

Names

Array of all paths, under which this device is known in the system. All these paths are links to one block device. For disk from the example above, it's content would be:

```
[
    '/dev/disk/by-id/ata-Samsung_SSD_840_Series_S19MNSAD500335K',
    '/dev/disk/by-id/wwn-0x50025385a0031e7c',
    '/dev/sda',
    '/dev/systemdisk;
]
```

Applications can use any of these properties to find a block device (using CQL or WQL).

Note: OpenLMI tries as hard as possible to have *DeviceID* and *InstanceID* properties really stable across reboots. Unfortunately, some hardware does not provide unique identifier for disks - typically in virtualized environment, there may be cases where DeviceID may be just `/dev/vda` and it may change when the virtual machine reorders the virtual disks after reconfiguration.

LMI_VGStoragePool Although volume groups are not exactly block devices, there are several ways how to identify *LMI_VGStoragePool* instances:

InstanceID

OpenLMI internal identifier of a volume group. It should be opaque for applications, i.e. applications should not parse it / interpret it in any way.

- Guaranteed to be unique in the managed system.
- SHOULD be persistent across reboots.

PoolID, *ElementName*

Name of the volume group.

- Guaranteed to be unique among all volume groups on the managed system. However, there can be other ManagedElements, such as logical volumes, with the same ElementName.
- SHOULD be persistent across reboots.

Name

Canonical path to the volume group, such as as `/dev/mapper/mygroup`. This property has been added to have the same way how to identify *CIM_StorageExtent* and *LMI_VGStoragePool*.

- Guaranteed to be unique in the managed system.
- Not persistent across reboots.

Overwrite policy

Before OpenLMI-Storage overwrites or deletes a device, it first checks if the device is **unused**.

Unused device:

- Is not mounted.
- Is not part of running device, e.g. MD RAID, Volume Group or LUKS.

If a device is used, any operation which would overwrite or delete it returns `CIM_Error` with error message "Device XYZ is mounted" or "Device XYZ is used by ABC". It is up to the application to first unmount the device, close the LUKS/dm-crypt device, stop the RAID or remove it from running Volume Group etc.

Asynchronous jobs

Most of storage manipulation methods, for example *CreateOrModifyVG*, can be time-consuming. Therefore the methods only check input parameters and return immediately with a reference to *LMI_StorageJob* instance. The operation itself is performed asynchronously on the server in a separate thread.

The returned *LMI_StorageJob* instance can be then used to either pull the operation status or applications can subscribe for job events and get an indication when status of a job changes.

Currently, only one job is being executed at a time, all others are enqueued and executed later.

Job status The job status is exposed in *OperationalStatus* and *JobState* properties. Their combination compose unique job status:

Job is	OperationalStatus	JobState
Queued	Dormant	New
Suspended	OK	Suspended
Running	OK	Running
Finished OK	Completed, OK	Completed
Failed	Completed, Error	Exception
Cancelled	Stopped	Terminated

Job.RequestStateChange method can be used to suspend, resume and cancel a job, while following rules apply:

- Only Queued job can be suspended.
- Only Suspended job can be resumed.
- Only Queued or Suspended job can be cancelled.

Note: Running job cannot be terminated in any way.

Figure 3.8: Job state machine.

By default, all job instances disappear automatically after 60 seconds after they reach any final state. This can be overridden by setting *TimeBeforeRemoval* and *DeleteOnCompletion* properties of a job.

Return value and output parameters Return value and output parameters of an asynchronous method call are stored in *LMI_StorageJob.JobOutParameters* property, which is EmbeddedObject of a class, which has property for each output parameter of the asynchronous method. The method return value itself is available there too, as `__ReturnValue` property.

For compatibility with SMI-S, the output parameters are also included in *LMI_StorageMethodResult.PostCallIndication* property, which is associated to the job. The property itself is embedded instance of *CIM_InstMethodCall* class. Return value is stored in its *ReturnValue* property. Output parameters are stored in its *MethodParameters* property.

LMI_AffectedStorageJobElement association can be also used to find created/modified element of a *LMI_StorageJob* instance.

Figure 3.9: Instance diagram of a job before finishing.

Figure 3.10: Instance diagram of a job after finishing.

Supported event filters

- PercentComplete property of a job changed:

```
SELECT * FROM LMI_StorageInstModification
    WHERE SourceInstance ISA LMI_StorageJob
        AND SourceInstance.CIM_ConcreteJob::PercentComplete
            <> PreviousInstance.CIM_ConcreteJob::PercentComplete
```

- State of a job changed:

```
SELECT FROM LMI_StorageInstModification
    WHERE SourceInstance ISA CIM_ConcreteJob
        AND SourceInstance.CIM_ConcreteJob::JobState <> PreviousInstance.CIM_ConcreteJob::JobSta
```

- A job reaches state "Completed/OK":

```
SELECT * FROM LMI_StorageInstModification
    WHERE SourceInstance ISA LMI_StorageJob
        AND SourceInstance.CIM_ConcreteJob::JobState = 7
```

- A job reaches state "Completed/Error":

```
SELECT * FROM LMI_StorageInstModification
    WHERE SourceInstance ISA LMI_StorageJob
        AND SourceInstance.CIM_ConcreteJob::JobState = 10
```

- New job was created:

```
SELECT * FROM LMI_StorageInstCreation WHERE SourceInstance ISA LMI_StorageJob
```

Note: All other indication filter queries will be rejected.

Usage

Block devices cannot be directly manipulated using intrinsic or extrinsic methods of *CIM_StorageExtent* or *LMI_VGStoragePool*.

Please use appropriate `ConfigurationService` to create, modify or delete devices or volume groups.

Partitioning

Disks or any other block devices with partition tables have their *LMI_StorageExtent* or its subclass associated to *LMI_DiskPartitionConfigurationCapabilities* using *LMI_InstalledPartitionTable*.

A GPT partition present on a block device are represented as *LMI_GenericDiskPartition*.

A MS-DOS partition present on a block device are represented as *LMI_DiskPartition*.

Both MS-DOS and GPT partitions are associated to the parent device using *LMI_PartitionBasedOn*. This BasedOn association contains also start and end sectors of the partitions. Note that logical partitions are associated with the extended partition where they are located, see the diagram below. Following instance diagram shows `/dev/sda` disk with MS-DOS partition table and:

- 3 primary partitions
- 1 extended partition
  - 2 logical partitions

Especially note that the extended partition `/dev/sda4` *contains* an extended partition table and all logical partitions are based on this extended partition. This is for compatibility with SMI-S and also it better illustrates physical composition of the partitions on the disk.

However, to create a partition on the device, applications can use both `/dev/sda` or `/dev/sda4` as value of `Extent` parameter in *LMI_CreateOrModifyPartition*, call.

Useful methods

LMI_CreateOrModifyPartition Creates a partition of given size on a device with GPT or MS-DOS partition table. It can automatically create extended and logical partitions when there is no space in the partition table for a primary partition.

CreateOrModifyPartition Creates a partition on a device with GPT or MS-DOS partition table. This method is provided for compatibility with SMI-S. Instead of providing requested size of the new partition, exact location of partition must be specified, which may result in suboptimal performance of the partition.

SetPartitionStyle Creates partition table on a device of requested size. If the size is not specified, the largest possible partition is created.

FindPartitionLocation Finds start and end sector where a partition would be created and returns size of the partition.

LMI_DeletePartition Destroys a partition.

Use cases

List supported partition table types Currently GPT and MS-DOS partition tables are supported. More types can be added later. Enumerate instances of *LMI_DiskPartitionConfigurationCapabilities* class to get list of all of them, together with their basic properties like partition table size and maximum number of partitions:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

part_styles = ns.LMI_DiskPartitionConfigurationCapabilities.instances()
for style in part_styles:
    print style.Caption
    print "Partition table size:", style.PartitionTableSize, "block(s)"
```

Create partition table Use *SetPartitionStyle* method.

Sample code to create GPT partition table on `/dev/sda`:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
partitioning_service = ns.LMI_DiskPartitionConfigurationService.first_instance()

# Find the disk
sda = ns.LMI_StorageExtent.first_instance({"Name": "/dev/sda"})

# Find the partition table style we want to create there
gpt_caps = ns.LMI_DiskPartitionConfigurationCapabilities.first_instance(
        {"InstanceID": "LMI:LMI_DiskPartitionConfigurationCapabilities:GPT"})

# Create the partition table
partitioning_service.SetPartitionStyle(
        Extent=sda,
        PartitionStyle=gpt_caps)
```

MS-DOS partition tables are created with the same code, just using different *LMI_DiskPartitionConfigurationCapabilities* instance.

Create partition Use *LMI_CreateOrModifyPartition* method.

Following code creates several partitions on `/dev/sda`. The code is the same for GPT and MS-DOS partitions:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
```

```
partitioning_service = ns.LMI_DiskPartitionConfigurationService.first_instance()
MEGABYTE = 1024*1024

# Define helper function
def print_partition(partition_name):
    partition = partition_name.to_instance()
    print "Created partition", partition.DeviceID, \
            "with", partition.NumberOfBlocks * partition.BlockSize, "bytes."

# Find the disk
sda = ns.LMI_StorageExtent.first_instance({"Name": "/dev/sda"})

# create 4 partitions with 100 MB each
for i in range(4):
    (ret, outparams, err) = partitioning_service.SyncLMI_CreateOrModifyPartition(
            Extent=sda,
            Size = 100 * MEGABYTE)
    print_partition(outparams['Partition'])

# Create partition with the whole remaining space - just omit 'Size' parameter
(ret, outparams, err) = partitioning_service.SyncLMI_CreateOrModifyPartition(
        Extent=sda)

print_partition(outparams['Partition'])
```

On an empty disk with GPT partition table this code creates:

- 4 partitions with 100 MB each.
- One partition with the largest continuous unpartitioned space on the disk.

On an empty disk with MS-DOS partition table, the code creates:

- 3 primary partitions, 100 MB each.
- One extended partition with the largest continuous unpartitioned space.
- One 100 MB logical partitions.
- One logical partition with the largest continuous free space on the extended partition.

The resulting partitions can be seen in the diagram above.

List all partitions on a disk Enumerate *LMI_PartitionBasedOn* associations of the disk.

Following code lists all partitions on `/dev/sda`, together with their location:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Find the disk
sda = ns.LMI_StorageExtent.first_instance({"Name": "/dev/sda"})

based_ons = sda.references(ResultClass="LMI_PartitionBasedOn")
for based_on in based_ons:
    print "Found partition", based_on.Dependent.DeviceID, \
        "at sectors", based_on.StartingAddress, based_on.EndingAddress
# TODO: check extended partition
```

Find the largest continuous unpartitioned space on a disk Using side-effect of *FindPartitionLocation*, we can find size of the largest partition that can be created on `/dev/sda`:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Find the disk
sda = ns.LMI_StorageExtent.first_instance({"Name": "/dev/sda"})
# Find LMI_DiskPartitionConfigurationCapabilities associated to the disk
sda_partition_capabilities = sda.associators(
        AssocClass='LMI_InstalledPartitionTable') [0]

# Call its FindPartitionLocation without 'Size' parameter
# - the largest available space is returned.
(ret, outparams, err) = sda_partition_capabilities.FindPartitionLocation(
        Extent=sda)

print "Largest space for a partition:", outparams['size']
```

Delete partition Call *LMI_DeletePartition*:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
partitioning_service = ns.LMI_DiskPartitionConfigurationService.first_instance()

sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
(ret, outparams, err) = partitioning_service.LMI_DeletePartition(
        Partition=sda1)
```

Future direction In future, we might implement:

- *LMI_CreateOrModifyPartition* would also modify existing partitions, for example resize them.

MD RAID

MD RAID devices are represented by *LMI_MDRAIDStorageExtent* class.

Configuration of a MD RAID device is represented by instance of *LMI_MDRAIDStorageSetting* associated to it. Currently this instance is there only for compatibility with SMI-S, but in future it may be extended to allow detailed configuration of the RAID.

Members of the MD RAID are associated to the *LMI_MDRAIDStorageExtent* instance by *LMI_MDRAIDBasedOn* association. Following instance diagram shows RAID5 `/dev/md/myRAID` with three devices:

Note the *Level* property in *LMI_MDRAIDStorageExtent*, which was added to simplify RAID level calculation, in SMI-S the data redundancy and striping is determined by *DataRedundancy*, *ExtentStripeLength* and *PackageRedundancy* properties.

Currently the MD RAID support is limited to creation and removal of RAIDs. It is not possible to modify existing RAID, e.g. add or remove devices to/from it and/or manage RAID spares.

Useful methods

CreateOrModifyMDRAID Creates a MD RAID of given level with given devices. Optionally, RAID name can be specified and in future also more detailed RAID configuration.

CreateOrModifyElementFromElements Creates a MD RAID in SMI-S way. It is necessary to provide correct Goal setting, which can be calculated e.g. by *CreateMDRAIDStorageSetting*

CreateMDRAIDStorageSetting This is helper method to calculate *LMI_StorageSetting* for given list of devices and given RAID level for *CreateOrModifyElementFromElements*.

DeleteMDRAID Destroys a MD RAID. There is no SMI-S function for this.

Use cases

Create MD RAID Use *CreateOrModifyMDRAID* method. Following example creates MD RAID level 5 named '/dev/md/myRAID' with three members:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find the devices we want to add to MD RAID
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})

# Create the RAID
(ret, outparams, err) = storage_service.SyncCreateOrModifyMDRAID(
        ElementName = "myRAID",
        InExtents= [sda1, sdb1, sdc1],
        Level=storage_service.CreateOrModifyMDRAID.LevelValues.RAID5)
raid = outparams['TheElement'].to_instance()
print "RAID", raid.DeviceID, \
        "level", raid.Level, \
        "of size", raid.BlockSize * raid.NumberOfBlocks, \
        "created"
```

The result is the same as shown in diagram above.

Create MD RAID in SMI-S way SMI-S applications can use *CreateOrModifyElementFromElements* method. Following example creates MD RAID level 5 named '/dev/md/myRAID' with three members:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find the devices we want to add to MD RAID
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})
```

```
# Calculate LMI_StorageSetting, e.g. using our helper method
# (SMI-S application can of course use standard caps.CreateSetting()
# and edit it manually)
caps = ns.LMI_MDRAIDStorageCapabilities.first_instance()
(ret, outparams, err) = caps.CreateMDRAIDStorageSetting(
        InExtents=[sda1, sdb1, sdc1],
        Level=caps.CreateMDRAIDStorageSetting.LevelValues.RAID5)
setting = outparams ['Setting'].to_instance()

# Create the RAID
(ret, outparams, err) = storage_service.SyncCreateOrModifyElementFromElements(
        InElements=[sda1, sdb1, sdc1],
        Goal=setting,
        ElementType = storage_service.CreateOrModifyElementFromElements.ElementTypeValues.StorageExte
raid = outparams['TheElement'].to_instance()
print "RAID", raid.DeviceID, \
        "level", raid.Level, \
        "of size", raid.BlockSize * raid.NumberOfBlocks, \
        "created"
```

List members of MD RAID Enumerate *LMI_MDRAIDBasedOn* associations of the MD RAID extent.

Following code lists all members od `/dev/md/myRAID`:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Find the disk
md = ns.LMI_StorageExtent.first_instance({"Name": "/dev/md/myRAID"})

devices = md.associators(AssocClass="LMI_MDRAIDBasedOn")
for dev in devices:
    print "Found device", dev.DeviceID
```

Delete MD RAID Call *DeleteMDRAID* method:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

md = ns.LMI_MDRAIDStorageExtent.first_instance({"Name": "/dev/md/myRAID"})
(ret, outparams, err) = storage_service.SyncDeleteMDRAID(TheElement=md)
```

Future direction In future, we might implement:

- Modification of existing MD RAIDs, for example adding/removing devices.
- Management of spare devices.
- Detailed information of device status, synchronization progress etc.
- Indications of various events, like RAID failed member, synchronization errors etc.

Logical Volume management

Volume Groups (VG) and Thin Pools (TP) are represented by *LMI_VGStoragePool* class. To differentiate between the two, *SpaceLimitDetermination* and *SpaceLimit* are both set or both empty.

If both are set, an instance of the class is a thin pool. *SpaceLimitDetermination* is always set to 4 (limitless thin pool, meaning that it can be overcommited) and *SpaceLimit* is set to the capacity of the storage allocated to the pool. Also, *RemainingManagedSpace* will be set to the remaining space on the pool. Due to the current limitation of the underlying storage library, if the pool is overcommited, its *RemainingManagedSpace* value is set to 0.

If both *SpaceLimitDetermination* and *SpaceLimit* are empty, the instance of the *LMI_VGStoragePool* class is a regular volume group.

Every *LMI_VGStoragePool* instance has associated one instance of *LMI_VGStorageSetting* representing its configuration (e.g. volume group extent size) and one instance of *LMI_LVStorageCapabilities*, representing its ability to create logical volumes (for SMI-S applications). Every *LMI_VGStoragePool* instance, if it is a thin pool, is associated with its thin logical volumes (if they exist) using *LMI_VGAllocatedFromStoragePool*.

Physical Volumes (PV) are associated to VGs using *LMI_VGAssociatedComponentExtent* association.

Logical Volumes (LV) and Thin Logical Volumes (TLV) are represented by *LMI_LVStorageExtent* class. If an instance of the class is a thin logical volume, *ThinlyProvisioned* is set to True.

Each *LMI_LVStorageExtent* instance is associated to its respective VG/TP using *LMI_LVAllocatedFromStoragePool* association.

In addition, LVs are associated to all PVs using *LMI_LVBasedOn* association. Following instance diagram shows one Volume Group `/dev/myGroup` based on three Physical Volumes `/dev/sda1`, `/dev/sdb1` and `/dev/sdc1` and two Logical Volumes `myVol1` and `myVol2`.

Note that the diagram is simplified and does not show *LMI_LVBasedOn* association, which associates every `myVolY` to `/dev/sdX1`.

The next instance diagram displays the Volume Group `/dev/myGroup` (see previous diagram) that has `myThinPool`, sized 100 MiB, associated to it. This Thin Pool is used to provision the 10 GiB Thin Logical Volume `/dev/mapper/myGroup-myThinVolume`. The VG/TP pair is connected with an *LMI_VGAllocatedFromStoragePool* association. *LMI_LVAllocatedFromStoragePool* association joins the TP/TLV pair.

Currently the LVM support is limited to creation and removal of VGs and LVs and to adding/removing devices to/from a VG. It is not possible to modify existing LV, e.g. or resize LVs. In future OpenLMI may be extended to have more configuration options in *LMI_VGStorageSetting* and *LMI_LVStorageSetting*.

Useful methods

CreateOrModifyVG Creates a Volume Group with given devices. The devices are automatically formatted with Physical Volume metadata. Optionally, the Volume Group extent size can be specified by using Goal parameter of the method.

This method can be also used to add/remove PVs to/from VG.

CreateOrModifyThinPool Creates or modifies a Thin Pool.

CreateOrModifyThinLV Create or modifies a Thin Logical Volume.

CreateOrModifyStoragePool Creates a Volume Group in SMI-S way.

CreateVGStorageSetting This is helper method to calculate *LMI_VGStorageSetting* for given list of devices for *CreateOrModifyStoragePool* method.

CreateOrModifyLV Creates a Logical Volume from given VG.

CreateOrModifyElementFromStoragePool Creates a Logical Volume in SMI-S way.

DeleteLV Destroys a Logical Volume or a Thin Logical Volume.

ReturnToStoragePool Destroys a Logical Volume in SMI-S way.

DeleteVG Destroys a Volume Group or a Thin Pool.

DeleteStoragePool Destroys a Volume Group in SMI-S way.

Use cases

Create Volume Group Use *CreateOrModifyVG* method. Following example creates a VG '/dev/myGroup' with three members and with default extent size (4MiB):

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find the devices we want to add to VG
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})

# Create the VG
(ret, outparams, err) = storage_service.SyncCreateOrModifyVG(
        ElementName="myGroup",
        InExtents=[sda1, sdb1, sdc1])
vg = outparams['Pool'].to_instance()
print "VG", vg.PoolID, \
        "with extent size", vg.ExtentSize, \
        "and",  vg.RemainingExtents, "free extents created."
```

The resulting VG is the same as shown in diagram above, except it does not have any LVs yet.

Create Thin Pool The VG from the previous example can be used to create a TP on. This example script creates a Thin Pool 'myThinPool' on the VG 'myGroup'. The TP is 100 MiB in size:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()
MEGABYTE = 1024*1024

# Find the volume group
vg = ns.LMI_VGStoragePool.first_instance({"InstanceID":"LMI:VG:myGroup"})

# Allocate a thin pool out of it
(ret, outparams, err) = storage_service.SyncCreateOrModifyThinPool(
        ElementName="myThinPool",
```

```
        InPool=vg.path,
        # 100 MiB
        Size=100 * MEGABYTE)
tp = outparams["Pool"].to_instance()
print "TP %s with %d MiB remaining" % \
        (tp.Name, tp.RemainingManagedSpace / MEGABYTE)
```

Create Volume Group in SMI-S way SMI-S applications can use *CreateOrModifyStoragePool* method. Following example creates a VG '/dev/myGroup' with three members and with default extent size (4MiB):

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find the devices we want to add to VG
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})

# Create the VG
(ret, outparams, err) = storage_service.SyncCreateOrModifyStoragePool(
        InExtents=[sda1, sdb1, sdc1],
        ElementName="myGroup")
vg = outparams['Pool'].to_instance()
print "VG", vg.PoolID, \
        "with extent size", vg.ExtentSize, \
        "and",  vg.RemainingExtents, "free extents created."
```

The resulting VG is the same as shown in diagram above, except it does not have any LVs yet.

Add and remove devices to/from a Volume Group *CreateOrModifyStoragePool* can be used to modify exising VG. Its 'InExtents' parameter specifies new list of Physical Volumes of the VG. When an PV is being removed from a VG, all its data are safely moved to a free PV.

Continuing with previous example, let's remove '/dev/sda1' from the VG and add '/dev/sdd1' to it:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find all the devices we want to be in VG
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})
sdd1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdd1"})

new_pvs = [sdb1, sdc1, sdd1]              # Without sda1!

# Find the VG
vg = ns.LMI_VGStoragePool.first_instance({"Name": "/dev/mapper/myGroup"})

# Set the list of PVs of the VG.
```

```
# All existing PVs, which are not listed in InExtents parameter will
# be removed from the VG. All new devices listed in InExtents parameter
# are added to the VG. All data in the VG are moved from the PVs being
# removed to a free PV, no data is lost.

(ret, outparams, err) = storage_service.SyncCreateOrModifyVG(
        InExtents=new_pvs,
        pool=vg.path)
```

Create Volume Group with specific extent size Use *CreateVGStorageSetting* to create *LMI_VGStorageSetting*, modify its *ExtentSize* property with desired extent size and finally call *CreateOrModifyVG* with the setting as `Goal` parameter. Following example creates a VG '/dev/myGroup' with three members and with 1MiB extent size (4MiB):

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()
MEGABYTE = 1024*1024

# Find the devices we want to add to VG
# (filtering one CIM_StorageExtent.instances()
# call would be faster, but this is easier to read)
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sdb1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
sdc1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdc1"})

# Create the LMI_VGStorageSetting
vg_caps = ns.LMI_VGStorageCapabilities.first_instance()
(ret, outparams, err) = vg_caps.CreateVGStorageSetting(
        InExtents = [sda1, sdb1, sdc1])
setting = outparams['Setting'].to_instance()
# Modify the LMI_VGStorageSetting
setting.ExtentSize = MEGABYTE
settinh.push()

# Create the VG
# (either of CreateOrModifyStoragePool or CreateOrModifyVG
# can be used with the same result)
(ret, outparams, err) = storage_service.SyncCreateOrModifyStoragePool(
        InExtents=[sda1, sdb1, sdc1],
        ElementName="myGroup",
        Goal=setting)
vg = outparams['Pool'].to_instance()
print "VG", vg.PoolID, \
        "with extent size", vg.ExtentSize, \
        "and",  vg.RemainingExtents, "free extents created."
```

List Physical Volumes of a Volume Group Enumerate *VGAssociatedComponentExtent* associations of the VG.

Following code lists all PVs of `/dev/myGroup`:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Find the VG
```

```
vg = ns.LMI_VGStoragePool.first_instance({"Name": "/dev/mapper/myGroup"})
pvs = vg.associators(AssocClass="LMI_VGAssociatedComponentExtent")
for pv in pvs:
    print "Found PV", pv.DeviceID
```

Create Logical Volume Use *CreateOrModifyLV* method. Following example creates two 100MiB volumes:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()
MEGABYTE = 1024*1024

# Find the VG
vg = ns.LMI_VGStoragePool.first_instance({"Name": "/dev/mapper/myGroup"})

# Create the LV
(ret, outparams, err) = storage_service.SyncCreateOrModifyLV(
        ElementName="Vol1",
        InPool=vg,
        Size=100 * MEGABYTE)
lv = outparams['TheElement'].to_instance()
print "LV", lv.DeviceID, \
        "with", lv.BlockSize * lv.NumberOfBlocks,\
        "bytes created."

# Create the second LV
(ret, outparams, err) = storage_service.SyncCreateOrModifyLV(
        ElementName="Vol2",
        InPool=vg,
        Size=100 * MEGABYTE)
lv = outparams['TheElement'].to_instance()
print "LV", lv.DeviceID, \
        "with", lv.BlockSize * lv.NumberOfBlocks, \
        "bytes created."
```

The resulting LVs are the same as shown in diagram above.

Create Thin Logical Volume The following example assumes that a TP was already created (see Create Thin Pool).

There already is a TP (100 MiB) in the system. This snippet of code creates a 10 GiB Thin Logical Volume and prints some information about it. Note that this TLV causes the underlying TP to be overcommited:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

# Find the thin pool
tp = ns.LMI_VGStoragePool.first_instance({"ElementName":"myThinPool"})

(ret, outparams, err) = storage_service.SyncCreateOrModifyThinLV(
        ElementName="myThinLV",
        ThinPool=tp.path,
        # 10 GiB
        Size=10 * GIGABYTE)
tlv = outparams["TheElement"].to_instance()
```

```
print "TLV %s of size %d GiB" % \
       (tlv.Name, tlv.BlockSize * tlv.NumberOfBlocks / GIGABYTE)
```

Create Logical Volume in SMI-S way Use *CreateOrModifyElementFromStoragePool* method. The code is the same as in previous sample, just different method is used:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()
MEGABYTE = 1024*1024

# Find the VG
vg = ns.LMI_VGStoragePool.first_instance({"Name": "/dev/mapper/myGroup"})

# Create the LV
(ret, outparams, err) = storage_service.SyncCreateOrModifyElementFromStoragePool(
        ElementName="Vol1",
        InPool=vg,
        Size=100 * MEGABYTE)
lv = outparams['TheElement'].to_instance()
print "LV", lv.DeviceID, \
        "with", lv.BlockSize * lv.NumberOfBlocks,\
        "bytes created."

# Create the second LV
(ret, outparams, err) = storage_service.SyncCreateOrModifyElementFromStoragePool(
        ElementName="Vol2",
        InPool=vg,
        Size=100 * MEGABYTE)
lv = outparams['TheElement'].to_instance()
print "LV", lv.DeviceID, \
        "with", lv.BlockSize * lv.NumberOfBlocks, \
        "bytes created."
```

Delete VG Call *DeleteVG* method:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

vg = ns.LMI_VGStoragePool.first_instance({"Name": "/dev/mapper/myGroup"})
(ret, outparams, err) = storage_service.SyncDeleteVG(
        Pool = vg)
```

Delete LV Call *DeleteLV* method:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
storage_service = ns.LMI_StorageConfigurationService.first_instance()

lv = ns.LMI_LVStorageExtent.first_instance({"Name": "/dev/mapper/myGroup-Vol2"})
```

```
(ret, outparams, err) = storage_service.SyncDeleteLV(
        TheElement=lv)
```

Future direction In future, we might implement:

- Modification of existing VGs and LVs, for example renaming VGs and LVs and resizing LVs.
- LVs with stripping and mirroring.
- Clustered VGs and LVs.
- Snapshots.
- Indications of various events.

File system management

Local file systems, both supported and unsupported, are represented by *LMI_LocalFileSystem* class and its subclasses.

Each *LMI_LocalFileSystem* instance of supported filesystems have associated one instance of *LMI_FileSystemSetting* representing its configuration (e.g. inode size).

Supported filesystems are: ext2, ext3, ext4, xfs, btrfs. Only supported filesystems can be created! Actual set of supported filesystems can be obtained from *LMI_FileSystemConfigurationCapabilities* instance associated to *LMI_FileSystemConfigurationService*. Following instance diagram shows four block devices:

- `/dev/sda1` and `/dev/sda2` with btrfs filesystem spanning both these devices.
- `/dev/sda3` with ext3 filesystem.
- `/dev/sda4` with msdos filesystems. The msdos filesystem is unsupported, therefore it has no *LMI_FileSystemSetting* associated.

Note: Currently the filesystem support is limited:

- Filesystems can be only created and deleted, it is not possible to modify existing filesystem.
- There is no way to set specific filesystem options when creating one. Simple `mkfs.<filesystem type>` is called, without any additional parameters.
- btrfs filesystem can be only created or destroyed. There is currently no support for btrfs subvolumes, RAIDs, and dynamic addition or removal of block devices.
- The *LMI_LocalFileSystem* instances do not report free and used space on the filesystems.

These limitations will be addressed in future releases.

Useful methods

LMI_CreateFileSystem Formats a StorageExtent with filesystem of given type. Currently the Goal parameter is not used, i.e. no filesystem options can be specified.

DeleteFileSystem Destroys a file system (*LMI_LocalFileSystem*) or other metadata, such as Physical Volume metadata or MD RAID metadata present (*LMI_DataFormat*) on a device.

Only unmounted filesystems and unused metadata can be deleted.

Use cases

Create File System Use *LMI_CreateFileSystem* method. Following example formats `/dev/sda3` with ext3:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
filesystem_service = ns.LMI_FileSystemConfigurationService.first_instance()

# Find the /dev/sda3 device
sda3 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda3"})

# Format it
(ret, outparams, err) = filesystem_service.SyncLMI_CreateFileSystem(
        FileSystemType=filesystem_service.LMI_CreateFileSystem.FileSystemTypeValues.EXT3,
        InExtents=[sda3])
```

The resulting filesystem is the same as shown in diagram above.

Create btrfs File System with two devices Use the same *LMI_CreateFileSystem* method as above. Following example formats `/dev/sda1` and `dev/sda2` as one btrfs volume:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
filesystem_service = ns.LMI_FileSystemConfigurationService.first_instance()

# Find the /dev/sda1+2 devices
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
sda2 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda2"})

# Format them
(ret, outparams, err) = filesystem_service.SyncLMI_CreateFileSystem(
        FileSystemType=filesystem_service.LMI_CreateFileSystem.FileSystemTypeValues.BTRFS,
        InExtents=[sda1, sda2])
```

The resulting filesystem is the same as shown in diagram above.

Delete filesystem Use *LMI_CreateFileSystem* method:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
filesystem_service = ns.LMI_FileSystemConfigurationService.first_instance()

sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
fs = sda1.first_associator(ResultClass='LMI_LocalFileSystem')
(ret, outparams, err) = filesystem_service.SyncDeleteFileSystem(
        TheFileSystem = fs)
```

Note that with one btrfs on multiple block devices, the whole btrfs volume is destroyed.

Future direction In future, we might implement:

- Add advanced options to *LMI_CreateFileSystem*
- Allow (some) filesystem modification, e.g. amount of reserved space for root user.
- Indications of various events, like filesystem is getting full.

Block device performance

OpenLMI-Storage provider reports I/O statistics of all block devices. Every instance of *CIM_StorageExtent* or its subclass has associated *LMI_BlockStorageStatisticalData* instance, which reports current I/O statistics like nr. of kbytes read/written etc.

Following instance diagram shows two block devices and their associated statistics:

There are many more classes related to block device performance, but these are provided mainly for compatibility with SMI-S. See following instance diagram, which shows the same two block devices, but now with all SMI-S classes:

The only useful method is *LMI_BlockStatisticsService.GetStatisticsCollection*, which returns I/O statistics of **all** block devices as semicolon-separated-list. The order of fields in this list is described in *LMI_BlockStatisticsManifest.CSVSequence* property.

Note: Even though properties in *LMI_BlockStorageStatisticalData* are 64-bit, they are tracked as 32-bit on 32-bit systems like `i686` or `ppc` by Linux kernel. They can wrap pretty quickly on modern hardware.

For example, with iSCSI drive on 10Gb/s link, the KBytesRead counter can wrap in around 27 minutes.

On 64-bit systems, these counters are tracked in 64-bits in Linux kernel and they wrap once in a few years.

Useful methods

LMI_BlockStatisticsService.GetStatisticsCollection Return I/O statistics of **all** block devices as CSV-formatted string. (CSV = semicolon-separated list).

Note that this method is currently synchronous and does not return a `Job`.

Use cases

Get I/O statistics of a block device Find *LMI_BlockStorageStatisticalData* associated to appropriate *CIM_StorageExtent*:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Find the /dev/sda3 device
sda3 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda3"})

# Find its statistics
stat = sda3.first_associator(ResultClass="LMI_BlockStorageStatisticalData")
print "KBytesRead:", stat.KBytesRead
```

Get I/O statistics of all block devices I Enumerate all *LMI_BlockStorageStatisticalData* instances on the system:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
```

```
# Find all LMI_BlockStorageStatisticalData instances
stats = ns.LMI_BlockStorageStatisticalData.instances()
for stat in stats:
    print "Device", stat.ElementName, "KBytesRead:", stat.KBytesRead
```

This approach can return huge list of *LMI_BlockStorageStatisticalData* instances on systems with lot of block devices.

Get I/O statistics of all block devices II Use *LMI_BlockStatisticsService.GetStatisticsCollection* method to get all statistics in one method call:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

# Print column headers
manifest = ns.LMI_BlockStatisticsManifest.first_instance()
print ";".join(manifest.CSVSequence)

# Print the real data
service = ns.LMI_BlockStatisticsService.first_instance()
(ret, outparams, err) = service.GetStatisticsCollection()
stats = outparams['Statistics']
for stat in stats:
    print stat
```

Note that this method is currently synchronous and does not return a `Job`.

Mounting

Note: Currently, only basic mounting/unmounting works. Persistency and mount flags (i.e. bind) are not implemented, yet. These limitations will be addressed in the future releases.

Every mount is represented by an *LMI_MountedFileSystem* instance. Each instance can have one or two *LMI_MountedFileSystemSetting* instances associated to it via *LMI_MountedFileSystemElementSettingData* (one for the currently mounted filesystem and one for a persistent entry in /etc/fstab). This association class has two important properties – *IsCurrent* and *IsNext* . Their meaning is described in detail in the On modes section.

LMI_MountedFileSystemSetting is used for representing mount options (e.g. whether to mount read-write or read-only).

The setting instance can also exists on its own. This means that it's not connected with *LMI_MountedFileSystem* by any association. Such situation can happen after *CreateSetting* is called. According to its *ChangeableType* property, it is either deleted after an hour (ChangeableType = Transient), or has to be associated or deleted manually (Changeable = Persistent).

Local filesystems are represented by *LMI_LocalFileSystem* class and its subclasses. Filesystems are associated to *LMI_MountedFileSystem* via *LMI_AttachedFileSystem* .

Note: Currently, only local filesystems are supported.

When a filesystem is currently mounted, the directory where the *LMI_MountedFileSystem* instance is attached at is represented by an *LMI_UnixDirectory* instance. These two instances are connected through an *LMI_MountPoint* association instance.

The following diagram shows a local ext4 partition /dev/sda2 currently mounted at /boot. The filesystem is specified by its UUID. No persitent entry in /etc/fstab is managed.

The next figure shows a local ext3 partition /dev/sda1 mounted at /home and also made persistent in /etc/fstab, both with slightly different mount options. The filesystem is specified by its UUID. Notice that the mount options are represented by two different *LMI_MountedFileSystemSetting* instances. The final diagram represents a state where a local ext4 partition /dev/sda4, filesystem of which is specified by its UUID, is mounted at /var/log and also has the respective entry written in /etc/fstab. Note that both settings (current mount and the persistent entry) are the same, as is indicated by IsNext and IsCurrent being set to 1.

Note: TODO: bind mount examples, remote fs examples

Using the mounting API

On modes When calling *CreateMount* or *DeleteMount* methods, one of their arguments is a mode. The mode is an enumeration that denotes values of two different properties of the *LMI_MountedFileSystemElementSettingData* association. They are *IsNext* and *IsCurrent*. They determine if the mount operation performs mount only, adds a persistent entry to /etc/fstab, or both.

The following table displays possible values and their respective meanings of *IsNext* and *IsCurrent* .

	Value	Meaning
IsNext	1	This property indicates if the associated setting will be applied as mount options on next reinitialization, i.e. on reboot. In mounting this means persistency, an entry in /etc/fstab.
	2	No entry in /etc/fstab.
IsCurrent	1	This property indicates if the associated setting represents current mount options of the MountedFileSystem.
	2	The device is not mounted.

Supported modes of *CreateMount*, *ModifyMount* and *DeleteMount* methods and their meaning are described in the following table. See decription of the methods for details.

Mode	IsNext	IsCurrent
1	1	1
2	1	Not affected.
4	2	2
5	2	Not affected.
32768	Not affected.	1
32769	Not affected.	2

Methods

CreateMount Mounts a device to the specified mountpoint.

ModifyMount Modifies (remounts) the specified filesystem.

DeleteMount Unmounts the specified filesystem.

All the methods are asynchronous.

DeleteMount() note If, after *DeleteMount*, *IsNext* and *IsCurrent* are both set to 2 (device was unmounted and its persistent entry removed), the corresponding *LMI_MountedFileSystem*, *LMI_MountedFileSystemSetting* and their association are removed. This implies that there cannot be any *LMI_MountedFileSystemElementSettingData* with both *IsNext* and *IsCurrent* set to 2.

Use cases Typical use of the mounting API could be like the following:

Use an *LMI_MountedFileSystemCapabilities* instance to create a setting instance using the *CreateSetting* method. This method creates an instance of *LMI_MountedFileSystemSetting* class with default property values.

Modify the setting instance as needed. This is done using the ModifyInstance intrinsic method. This step is optional if the admin is satisfied with the default set of values.

Use an *LMI_MountConfigurationService* to create a mount using the *CreateMount* method or modify a mount using the *ModifyMount* method. You can also use an *LMI_MountConfigurationService* to unmount a mount using the *DeleteMount* .

Example 1 This example demonstrates mounting /dev/sda partition with a customized setting.

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace

cap = ns.LMI_MountedFileSystemCapabilities.first_instance()

# Create an LMI_MountedFileSystemSetting instance
(rc, out, err) = cap.CreateSetting()
setting_name = out['Setting']
setting = setting_name.to_instance()

# Modify the setting instance with requested options
setting.AllowWrite = False
setting.InterpretDevices = False
setting.push()

# Find the filesystem to mount
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sda1"})
fs = sda1.first_associator(ResultClass='LMI_LocalFileSystem')

# Mount it
# Mode == 32768 -> only mount, no fstab entry
mount_service = ns.LMI_MountConfigurationService.first_instance()
(rc, out, err) = mount_service.SyncCreateMount(
     Goal=setting,
     FileSystemType='ext4',
     Mode=32768,
     FileSystem=fs,
     MountPoint='/mnt/test',
     FileSystemSpec='/dev/sda1')
```

Example 2 In this example, /mnt, that was mounted in Example 1, is unmounted.

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
mount_service = ns.LMI_MountConfigurationService.first_instance()

mnt = ns.LMI_MountedFileSystem.first_instance({"MountPointPath": "/mnt/test"})

if not mnt:
    raise BaseException("Mountpoint does not exist: /mnt/test")

(rc, out, err) = mount_service.SyncDeleteMount(
```

```
    Mount=mnt,
    Mode=32769)
```

Note: Currently, only basic mounting/unmounting works. Persistency and mount flags (i.e. bind) are not implemented, yet. These limitations will be addressed in the future releases.

Storage encryption

OpenLMI supports Linux Unified Key Setup (LUKS) to encrypt block devices. This means any device can be formatted with LUKS, which destroys all data on the device and allows for encryption of the device future content. The block device then contains *encrypted* data. To see unencrypted (clear-text) data, the LUKS format must be *opened*. This operation creates new block device, which contains the *clear-text* data. This device is just regular block device and can be formatted with any filesystem. All write operations are automatically encrypted and stored in the LUKS format data.

To hide the clear-text data, the clear text device must be *closed*. This destroys the clear-text device, preserving only encrypted content in the LUKS format data.

The data are encrypted by a key, which is accessible using a pass phrase. There can be up to 8 different pass phrases per LUKS format. Any of them can be used to open the format and to unencrypt the data.

Note: There is currently no way how to specify which algorithm, key or key size will be used to actually encrypt the data. *cryptsetup* defaults are applied.

CIM_StorageExtent can be recognized by *LMI_LUKSFormat* resides on it.

If the *LMI_LUKSFormat* is opened, the new clear-text device is created as *LMI_LUKSStorageExtent*, which has `BasedOn` association to the original *CIM_StorageExtent*.

All operations with LUKS format can be done using *LMI_ExtentEncryptionConfigurationService*. Following instance diagram shows one encrypted partition. The LUKS is not opened, which means that there is no clear-text device on the system. Following instance diagram shows one encrypted partition with opened LUKS. That means any data written

Figure 3.11: Instance diagram of closed LUKS format on a partition.

to `/dev/mapper/cleartext` are automatically encrypted and stored on the partition.

Figure 3.12: Instance diagram of opened LUKS format on a partition.

Useful methods

CreateEncryptionFormat Formats a StorageExtent with LUKS format. All data on the device are destroyed.

OpenEncryptionFormat *Opens* given LUKS format and shows its clear-text in *LMI_LUKSStorageExtent*.

CloseEncryptionFormat *Closes* given LUKS format and destroys its previously opened *LMI_LUKSStorageExtent*.

AddPassphrase, ***DeletePassphrase*** Manage pass phrases for given LUKS format.

Use cases

Create encrypted file system. Use *CreateEncryptionFormat* to create LUKS format, open it and create ext3 filesystem on it:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
encryption_service = ns.LMI_ExtentEncryptionConfigurationService.first_instance()
filesystem_service = ns.LMI_FileSystemConfigurationService.first_instance()

# Find the /dev/sda1 device
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})

# Format it
(ret, outparams, err) = encryption_service.SyncCreateEncryptionFormat(
    InExtent=sda1, Passphrase="opensesame")
luks_format = outparams['Format'].to_instance()

# 'Open' it as /dev/mapper/secret_data
(ret, outparams, err) = encryption_service.SyncOpenEncryptionFormat(
    Format=luks_format,
    Passphrase="opensesame",
    ElementName="secret_data")
clear_text_extent = outparams['Extent'].to_instance()

# Format the newly created clear-text device
(ret, outparams, err) = filesystem_service.SyncLMI_CreateFileSystem(
FileSystemType=filesystem_service.LMI_CreateFileSystem.FileSystemTypeValues.EXT3,
InExtents=[clear_text_extent])
```

The resulting situation is the same as shown in *the second diagram* above.

Close opened LUKS format *CloseEncryptionFormat* can be used to destroy the clear-text device so only encrypted data is available. The clear-text device must be unmounted first!

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
encryption_service = ns.LMI_ExtentEncryptionConfigurationService.first_instance()

# Find the LUKS format
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
luks_format = sda1.first_associator(AssocClass="LMI_ResidesOnExtent")

# Close it
(ret, outparams, err) = encryption_service.SyncCloseEncryptionFormat(
    Format=luks_format)
```

The resulting situation is the same as shown in *the first diagram* above.

Pass phrase management Pass phrases can be added or deleted using *AddPassphrase* and *DeletePassphrase* methods.

Following code can be used to replace weak 'opensesame' password with something stronger:

```
# Connect to the remote system and prepare some local variables
connection = connect("remote.host.org", "root", "opensesame")
ns = connection.root.cimv2  # ns as NameSpace
encryption_service = ns.LMI_ExtentEncryptionConfigurationService.first_instance()
```

```
# Find the LUKS format
sda1 = ns.CIM_StorageExtent.first_instance({"Name": "/dev/sdb1"})
luks_format = sda1.first_associator(AssocClass="LMI_ResidesOnExtent")

# Add a pass phrase
(ret, outparams, err) = encryption_service.AddPassphrase(
        Format=luks_format,
        Passphrase="opensesame",
        NewPassphrase="o1mcW+O27F")

# Remove the old weak one
(ret, outparams, err) = encryption_service.DeletePassphrase(
        Format=luks_format,
        Passphrase="opensesame")
```

There are 8 so called key slots, which means each LUKS formats supports up to 8 different pass phrases. Any of the pass phrases can be used to open the LUKS format. Status of these key slots can be found in *LMI_LUKSFormat.SlotStatus* property.

Note: Previous releases allowed to use `DeleteInstance` intrinsic method to delete various `CIM_StorageExtents`. This method is now deprecated and will be removed from future releases of OpenLMI-Storage. The reason is that `DeleteInstance` cannot be asynchronous and could block the whole provider for a long time.

Configuration

Configuration is stored in `/etc/openlmi/storage/storage.conf`.

In addition to *common configuration options*, this provider can be configured to allow or deny various filesystem operations. Default configuration:

```
[Log]
# Toggles logging of detailed debug messages in Blivet.
DebugBlivet=False

[Storage]
# Path to temporary directory. The provider (usually running as root) need
# read/write access there. When SELinux or other security enhancement
# mechanism is used, **only** the provider should have read/write access
# to this directory.
Tempdir=/tmp
```

Options and their values are self-explanatory.

Persistent setting

OpenLMI-Storage stores persistent data in `/var/lib/openlmi-storage/`. Typically, various *CIM_SettingData* instances with *ChangeableType* `Changeable - Persistent` are stored here.

3.2.15 Networking Provider

OpenLMI-Networking is CIM provider which manages local network devices.

This provider is based on following DMTF standards:

- DSP1116 - IP Configuration Profile
- DSP1035 - Host LAN Network Port Profile

The knowledge of these standards is not necessary, but it can help a lot.

Application developers should first get familliar with *Networking API concepts* and then look at *usage of OpenLMI-Networking*.

Content:

Networking API concepts

OpenLMI-Networking provides CIM API. Some CIM knowledge is required and this guide assumes that reader can routinely read and modify remote CIM objects and call their intrinsic and extrinsic methods.

Hardware representation

There is instance of subclass of *CIM_NetworkPort* for each physical network device present in the system, e.g. *LMI_EthernetPort* for ethernet ports.

Instances of class *LMI_LANEndpoint* represent communication endpoints, identified by MAC address to which the network port will respond. It's associated to the corresponding instance of *CIM_NetworkPort* subclass via instance of *LMI_NetworkDeviceSAPImplementation*.

Current network configuration

LMI_IPNetworkConnection instances represents the network connection in the system, e.g. "eth0", "p1p1". Instances of this class are associated to the *LMI_LANEndpoint* via *LMI_EndpointForIPNetworkConnection*.

Note: There are usually 1:1:1 relation between instances of CIM_NetworkPort subclasses, LMI_LANEndpoint instances and LMI_IPNetworkConnection instance in this provider. The classes are implemented for sake of compatibility with DMTF profiles.

LMI_IPProtocolEndpoint - there is instance of this class for each IP address on any network device and the instance is associated with *LMI_IPNetworkConnection* via *LMI_NetworkSAPSAPDependency* and with *LMI_LANEndpoint* via *LMI_BindsToLANEndpoint*.

Default gateway is represented by instance of *LMI_NetworkRemoteServiceAccessPoint* with attribute `AccessContext` equal to `2 (Default Gateway)`.

Instances of class *LMI_IPVersionSettingData* represent IPv4 or IPv6 support. If there is instance of this class associated with *CIM_ComputerSystem* it means that the system supports IPv4 and/or IPv6 (depending on value of the ProtocolIFType property). Instances of this class can be associated also to *LMI_IPNetworkConnection*. It means that the network connection supports IPv4 and/or IPv6.

Settings

The OpenLMI networking provider is based on concept of `setting`. `Setting` is a set of configuration options that can be applied to an interface. Each setting is represented by instance of *LMI_IPAssignmentSettingData* and

it is aggregator for detailed configuration represented by instances of following classes: *LMI_DHCPSettingData*, *LMI_DNSSettingData*, *LMI_ExtendedStaticIPAssignmentSettingData*. These detailed settings are associated with the master setting via *LMI_OrderedIPAssignmentComponent* where the master has role `GroupComponent`.

`Settings` available for given port are associated by *LMI_IPElementSettingData*. Its property `IsCurrent` is `1` when the setting is currently active. Property `IsDefault` is `1` when the setting is automatically activated.

Altering and applying settings

Method *LMI_CreateIPSetting* of the *LMI_IPNetworkConnectionCapabilites* class can be used to create new setting. The setting will be tied to *LMI_IPNetworkConnection* that is associated with given *LMI_IPNetworkConnectionCapabilites*.

Singleton class *LMI_IPConfigurationService* provides method *ApplySettingToIPNetworkConnection* that applies *LMI_IPAssignmentSettingData* to *LMI_IPNetworkConnection*.

Bridging and bonding

Current state Instance of the *LMI_LinkAggregator8023ad* class represents currently active bond. It's associated to the *LMI_LAGPort8023ad* representing bonded interface via *LMI_LinkAggregationBindsTo*.

Instance of the *LMI_SwitchService* class represents currently active bridge. It's associated to the *LMI_SwitchPort* representing bridged interface via *LMI_SwitchesAmong*".

Creating bridge/bond Creating bridge/bond setting is the same as creating any other setting, just the `Type` parameter of the *LMI_CreateIPSetting* is different (`Bonding` or `Bridging`).

Bonding/bridging setting details can be altered by changing the properties of *LMI_BondingMasterSettingData* (or *LMI_BridgingMasterSettingData*) instance that is returned from the *LMI_CreateIPSetting* method.

For activating bridge/bond setting, use *ApplySettingToIPNetworkConnection* of the *LMI_IPConfigurationService* class.

For deletion of the bridge/bond setting just delete the "master" setting (the one created by *LMI_CreateIPSetting*). Deleting other settings will just remove the slave from the settings.

Enslaving First network interface is enslaved to the given bond/bridge setting automatically (depending on what *LMI_IPNetworkConnectionCapabilities* is the *LMI_CreateIPSetting* method called). Other interface can be enslaved by using *LMI_CreateSlaveSetting* method of the *LMI_IPNetworkConnectionCapabilities*.

Alter the *LMI_BondingSlaveSettingData* (or *LMI_BridgingSlaveSettingData*) instance to change the properties of bond/bridge slave.

Usage

All example scripts are for `lmishell`. See it's documentation on OpenLMI page.

We also assume that `lmishell` is connected to the CIMOM and the connection is stored in `connection` variable and variable `ns` points to cimv2 namespace:

```
connection = connect("server", "username", "password")
ns = connection.root.cimv2
```

Enumeration of network devices

Obtaining a list of network devices can be done by executing following commands in `lmishell`:

```
for device in ns.LMI_IPNetworkConnection.instances():
    print device.ElementName
```

Get parameters of network devices

Obtaining parameters of network device might be a little bit tricky. DMTF standards split network device to three classes and one might need to traverse between them through associations, see *Networking API concepts*.

Following example prints name, its status, MAC address, link technology and maximal speed for each device.

MAC address is not in the *LMI_IPNetworkConnection* class and must be accessed through *LMI_EndpointForIPNetworkConnection* association to *LMI_LANEndpoint* class, same for MaxSpeed and LinkTechnology, those are in *CIM_NetworkPort* subclasses, associated through *LMI_NetworkDeviceSAPImplementation* class:

```
for device in ns.LMI_IPNetworkConnection.instances():
    # print device name
    print device.ElementName,
    # print operating status
    print ns.LMI_IPNetworkConnection.OperatingStatusValues.value_name(device.OperatingStatus),

    # MAC address in not part of LMI_IPNetworkConnection but LMI_LANEndpoint class,
    # which is associated through LMI_EndpointForIPNetworkConnection
    lanendpoint = device.first_associator(AssocClass="LMI_EndpointForIPNetworkConnection")

    # print MAC address
    print lanendpoint.MACAddress,

    # LinkTechnology is part of CIM_NetworkPort subclasses, we need to traverse
    # through LMI_NetworkDeviceSAPImplementation association
    networkport = lanendpoint.first_associator(AssocClass="LMI_NetworkDeviceSAPImplementation")

    # print link technology
    print ns.CIM_NetworkPort.LinkTechnologyValues.value_name(networkport.LinkTechnology),

    # network speed might not be defined
    if networkport.MaxSpeed:
        # Convert bps to Mbps
        print "%dMbps" % (networkport.MaxSpeed // (1024*1024)),
    else:
        print "unknown",
    print
```

Get current IP configuration

Current IP addresses are in the *LMI_IPProtocolEndpoint* class associated to given *LMI_IPNetworkConnection*:

```
device = ns.LMI_IPNetworkConnection.first_instance({'ElementName': 'eth0'})
for endpoint in device.associators(AssocClass="LMI_NetworkSAPSAPDependency", ResultClass="LMI_IPProto
    if endpoint.ProtocolIFType == ns.LMI_IPProtocolEndpoint.ProtocolIFTypeValues.IPv4:
        print "IPv4: %s/%s" % (endpoint.IPv4Address, endpoint.SubnetMask)
```

```
    elif endpoint.ProtocolIFType == ns.LMI_IPProtocolEndpoint.ProtocolIFTypeValues.IPv6:
        print "IPv6: %s/%d" % (endpoint.IPv6Address, endpoint.IPv6SubnetPrefixLength)
```

Default gateway is represented by instance of *LMI_NetworkRemoteServiceAccessPoint* with `AccessContext` equal to `DefaultGateway`:

```
for rsap in device.associators(AssocClass="LMI_NetworkRemoteAccessAvailableToElement", ResultClass="
    if rsap.AccessContext == ns.LMI_NetworkRemoteServiceAccessPoint.AccessContextValues.DefaultGatewa
        print "Default Gateway: %s" % rsap.AccessInfo
```

For the list of DNS servers we need to traverse the object model a little bit. First get *LMI_IPProtocolEndpoint* instances associated with given *LMI_IPNetworkConnection* via *LMI_NetworkSAPSAPDependency*. Then use the same association to get instances of *LMI_DNSProtocolEndpoint*. Finally instances of *LMI_NetworkRemoteServiceAccessPoint* with `AccessContext` equal to `DNS Server` associated through *LMI_NetworkRemoteAccessAvailableToElement* have the DNS server address in the `AccessInfo` property.

Note that there might be more possible path to get to the RemoteServiceAccessPath and you might get duplicated entries. The `set` is used here to deduplicate the list of DNS servers:

```
dnsservers = set()
for ipendpoint in device.associators(AssocClass="LMI_NetworkSAPSAPDependency", ResultClass="LMI_IPPro
    for dnsedpoint in ipendpoint.associators(AssocClass="LMI_NetworkSAPSAPDependency", ResultClass="
        for rsap in dnsedpoint.associators(AssocClass="LMI_NetworkRemoteAccessAvailableToElement", Re
            if rsap.AccessContext == ns.LMI_NetworkRemoteServiceAccessPoint.AccessContextValues.DNSSe
                dnsservers.add(rsap.AccessInfo)
print "DNS:", ", ".join(dnsservers)
```

Bring up / take down a network device

Note: Changing the state of a network device is not recommended! Just disconnect the active setting.

Use method *RequestStateChange* of the *LMI_LANEndpoint* object. `RequestedState` parameter can be either `Enabled` or `Disabled`:

```
lanendpoint = ns.LMI_LANEndpoint.first_instance({ "ElementName": "eth0" })
lanendpoint.RequestStateChange(RequestedState=ns.LMI_LANEndpoint.RequestedStateValues.Enabled)
```

Enumerate available settings

One setting is a set of configuration options that are applicable to a network interface. This setting is represented by a *LMI_IPAssignmentSettingData* instances that have `AddressOrigin` equal to `Cumulative Configuration`:

```
for settingdata in ns.LMI_IPAssignmentSettingData.instances():
    if settingdata.AddressOrigin == ns.LMI_IPAssignmentSettingData.AddressOriginValues.cumulativecon
        print "Setting: %s" % settingdata.Caption
```

Obtaining setting details

Setting configuration is spread between the instances of *LMI_IPAssignmentSettingData* subclasses associated with the "master" setting:

```
settingdata = ns.LMI_IPAssignmentSettingData.first_instance({ "Caption": "eth0" })
for setting in settingdata.associators(AssocClass="LMI_OrderedIPAssignmentComponent"):
    if setting.classname == "LMI_DHCPSettingData":
        if setting.ProtocolIFType == ns.LMI_IPAssignmentSettingData.ProtocolIFTypeValues.IPv4:
            print "IPv4 DHCP"
        else:
            print "IPv6 DHCPv6"
    elif setting.classname == "LMI_ExtendedStaticIPAssignmentSettingData":
        for i in range(len(setting["IPAddresses"])):
            if setting["ProtocolIFType"] == ns.LMI_IPAssignmentSettingData.ProtocolIFTypeValues.IPv4:
                print "Static IPv4 address: %s/%s, Gateway %s" % (
                        setting["IPAddresses"][i],
                        setting["SubnetMasks"][i],
                        setting["GatewayAddresses"][i])
            else:
                print "Static IPv6 address: %s/%d, Gateway %s" % (
                        setting["IPAddresses"][i],
                        setting["IPv6SubnetPrefixLengths"][i],
                        setting["GatewayAddresses"][i])
    elif (setting.classname == "LMI_IPAssignmentSettingData" and
        setting["AddressOrigin"] == ns.LMI_IPAssignmentSettingData.AddressOriginValues.Stateless):
            print "IPv6 Stateless"
```

Create new setting

New setting is created by calling *LMI_CreateIPSetting* method on the instance of *LMI_IPNetworkConnectionCapabilities*, which is associated with *LMI_IPNetworkConnection* through *LMI_IPNetworkConnectionElementCapabilities*. It also has the `ElementName` property same as is the name of the network interface.

Created setting can be modified by using `ModifyInstance` intrinsic method (`push()` in the lmishell).

Let's say we want to create a new setting with static IPv4 and stateless IPv6 configuration for given network interface:

```
capability = ns.LMI_IPNetworkConnectionCapabilities.first_instance({ 'ElementName': 'eth0' })
result = capability.LMI_CreateIPSetting(Caption='eth0 Static',
        IPv4Type=capability.LMI_CreateIPSetting.IPv4TypeValues.Static,
        IPv6Type=capability.LMI_CreateIPSetting.IPv6TypeValues.Stateless)
setting = result.rparams["SettingData"].to_instance()
for settingData in setting.associators(AssocClass="LMI_OrderedIPAssignmentComponent"):
    if setting.ProtocolIFType == ns.LMI_IPAssignmentSettingData.ProtocolIFTypeValues.IPv4:
        # Set static IPv4 address
        settingData.IPAddresses = ["192.168.1.100"]
        settingData.SubnetMasks = ["255.255.0.0"]
        settingData.GatewayAddresses = ["192.168.1.1"]
        settingData.push()
```

Set DNS servers for given setting

DNS server for given setting is stored in the *DNSServerAddresses* property of class *LMI_DNSSettingData*.

Following code adds IPv4 DNS server to the existing setting:

```
setting = ns.LMI_IPAssignmentSettingData.first_instance({ "Caption": "eth0 Static" })
for settingData in setting.associators(AssocClass="LMI_OrderedIPAssignmentComponent"):
    if (settingData.classname == "LMI_DNSSettingData" and
```

```
            settingData.ProtocolIFType == ns.LMI_IPAssignmentSettingData.ProtocolIFTypeValues.IPv4):
        settingData.DNSServerAddresses.append("192.168.1.1")
        settingData.push()
```

Manage static routes for given setting

Static route can be added by calling *LMI_AddStaticIPRoute* method on the instance of the *LMI_IPAssignmentSettingData* class:

```
setting = ns.LMI_IPAssignmentSettingData.first_instance({ "Caption": "eth0 Static" })
result = setting.LMI_AddStaticIPRoute(
        AddressType=setting.LMI_AddStaticIPRouteValues.IPv4,
        DestinationAddress="192.168.2.1",
        DestinationMask="255.255.255.0")
route = result.rparams["Route"]
```

Additional parameters can be set by modifying the instance of *LMI_IPRouteSettingData*. The route can be deleted by using `DeleteInstance` intrinsic method (`delete()` in lmishell).

Delete setting

For setting deletion just call DeleteInstance intrinsic method (`delete()` in the lmishell) to the instance of *LMI_IPAssignmentSettingData*:

```
setting = ns.LMI_IPAssignmentSettingData.first_instance({ 'Caption': 'eth0 Static' })
setting.delete()
```

Apply setting

The setting can by applied to the network interface by calling *ApplySettingToIPNetworkConnection* of the *LMI_IPConfigurationService* class.

This method is asynchronous and returns a job, but lmishell can call it synchronously:

```
setting = ns.LMI_IPAssignmentSettingData.first_instance({ "Caption": "eth0 Static" })
port = ns.LMI_IPNetworkConnection.first_instance({ 'ElementName': 'ens8' })
service = ns.LMI_IPConfigurationService.first_instance()
service.SyncApplySettingToIPNetworkConnection(SettingData=setting, IPNetworkConnection=port, Mode=32
```

`Mode` parameter affects how is the setting applied. Most commonly used values are:

- Mode 1 – apply the setting now and make it auto-activated
- Mode 2 – just make it auto-activated, don't apply now
- Mode 4 – disconnect and disable auto-activation
- Mode 5 – don't change the setting state, only disable auto-activation
- Mode 32768 – apply the setting
- Mode 32769 – disconnect

Bridging and bonding

Warning: Bridge, bond and vlan support needs to be explicitly enabled when using 0.8 version of NetworkManager as a backend (for example on RHEL-6). Add following line to the /etc/sysconfig/network file and restart NetworkManager
NM_BOND_BRIDGE_VLAN_ENABLED=yes

Setting up Use following code to create and activate bond with eth0 and eth1 interfaces:

```
# Get the interfaces
interface1 = ns.LMI_IPNetworkConnection.first_instance({ 'ElementName': 'eth0' })
interface2 = ns.LMI_IPNetworkConnection.first_instance({ 'ElementName': 'eth1' })

# Get the capabilities
capability1 = interface1.first_associator(AssocClass="LMI_IPNetworkConnectionElementCapabilities",
        ResultClass="LMI_IPNetworkConnectionCapabilities")
capability2 = interface2.first_associator(AssocClass="LMI_IPNetworkConnectionElementCapabilities",
        ResultClass="LMI_IPNetworkConnectionCapabilities")
# Use one of the capabilities to create the bond
result = capability1.LMI_CreateIPSetting(Caption='Bond',
        Type=capability1.LMI_CreateIPSetting.TypeValues.Bonding,
        IPv4Type=capability1.LMI_CreateIPSetting.IPv4TypeValues.DHCP)
setting = result.rparams["SettingData"].to_instance()
# Get first slave setting
slave1setting = setting.first_associator_name(ResultClass="LMI_BondingSlaveSettingData",
            AssocClass="LMI_OrderedIPAssignmentComponent")
# Enslave the second interface using the second capability
result = capability2.LMI_CreateSlaveSetting(MasterSettingData=setting)
# Get second slave setting
slave2setting = result.rparams["SettingData"]
service = ns.LMI_IPConfigurationService.first_instance()
# Activate the bond
service.SyncApplySettingToIPNetworkConnection(
        SettingData=slave1setting,
        IPNetworkConnection=interface1,
        Mode=32768)
service.SyncApplySettingToIPNetworkConnection(
        SettingData=slave2setting,
        IPNetworkConnection=interface2,
        Mode=32768)
```

Displaying current state Following code displays existing bonds and bonded interfaces:

```
for linkaggregation in ns.LMI_LinkAggregator8023ad.instances():
    print "Bond: %s" % linkaggregation.Name
    for lagport in linkaggregation.associators(AssocClass="LMI_LinkAggregationBindsTo",
            ResultClass="LMI_LAGPort8023ad"):
        print "Bonded interface: %s" % lagport.Name
```

Following code displays existing bridges and bridged interfaces:

```
for switchservice in ns.LMI_SwitchService.instances():
    print "Bridge: %s" % switchservice.Name
    for switchport in switchservice.associators(AssocClass="LMI_SwitchesAmong",
```

```
            ResultClass="LMI_SwitchPort"):
        print "Bridged interface: %s" % switchport.Name
```

OpenLMI Classes:

3.2.16 CIM classes

As described *elsewhere*, WBEM just provides remote API over set of protocols. This API is object oriented and here you can find list of all classes that are remotely accessible.

The list is organized into inheritance tree(s).

CIM_AbstractComponent
– *CIM_Component*
– *CIM_OrderedComponent*
| – *LMI_OrderedIPAssignmentComponent*
– *CIM_DirectoryContainsFile*
| – *LMI_DirectoryContainsFile*
– *CIM_SettingsDefineCapabilities*
| – *LMI_SettingsDefineManagementCapabilities*
| – *LMI_SettingsDefineAccountCapabilities*
– *CIM_SystemComponent*
| – *CIM_HostedFileSystem*
| | – *LMI_HostedFileSystem*
| – *CIM_AccountOnSystem*
| | – *LMI_AccountOnSystem*
| – *CIM_SystemDevice*
| – *LMI_NetworkSystemDevice*
| – *LMI_PCIDeviceSystemDevice*
| – *LMI_SystemStorageDevice*
| – *LMI_MemorySystemDevice*
| – *LMI_BatterySystemDevice*
| – *LMI_ProcessorSystemDevice*
| – *LMI_PCIBridgeSystemDevice*
| – *LMI_DiskDriveSystemDevice*
– *CIM_AssociatedComponentExtent*
| – *LMI_VGAssociatedComponentExtent*
– *CIM_Container*
| – *LMI_SystemSlotContainer*
| – *LMI_DiskPhysicalPackageContainer*
| – *LMI_BaseboardContainer*
| – *LMI_PhysicalBatteryContainer*
| – *LMI_MemorySlotContainer*
| – *LMI_PortPhysicalConnectorContainer*
| – *LMI_ProcessorChipContainer*
| – *LMI_PhysicalMemoryContainer*
– *LMI_RootDirectory*

CIM_AbstractElementStatisticalData
– CIM_ElementStatisticalData
– LMI_StorageElementStatisticalData

CIM_AffectedJobElement
– LMI_AffectedJobElement
– LMI_AffectedSoftwareJobElement
– LMI_AffectedStorageJobElement
– LMI_AffectedNetworkJobElement
– LMI_AffectedSELinuxJobElement

CIM_AssignedIdentity
– LMI_AssignedGroupIdentity
– LMI_AssignedAccountIdentity

CIM_AssociatedBlockStatisticsManifestCollection
– LMI_AssociatedBlockStatisticsManifestCollection

CIM_AssociatedJobMethodResult
– LMI_AssociatedJobMethodResult
– LMI_AssociatedSoftwareJobMethodResult
– LMI_AssociatedSELinuxJobMethodResult
– LMI_AssociatedStorageJobMethodResult

CIM_Dependency
– CIM_RemoteAccessAvailableToElement
| *– LMI_NetworkRemoteAccessAvailableToElement*
– LMI_SELinuxServiceHasElement
– CIM_AbstractBasedOn
| *– CIM_BasedOn*
| *– LMI_MDRAIDBasedOn*
| *– LMI_PartitionBasedOn*
| *– LMI_LVBasedOn*
| *– LMI_LUKSBasedOn*
– CIM_MediaPresent
| *– LMI_MediaPresent*
– CIM_RouteUsesEndpoint
| *– LMI_RouteUsesEndpoint*
– CIM_AssociatedSensor
| *– LMI_FanAssociatedSensor*
– CIM_Realizes
| *– LMI_PhysicalMemoryRealizes*
| *– LMI_ProcessorChipRealizes*
| *– LMI_PhysicalBatteryRealizes*
| *– LMI_DiskDriveRealizes*
– CIM_DeviceSAPImplementation

| – *LMI_DiskDriveDeviceSAPImplementation*
| – *LMI_NetworkDeviceSAPImplementation*
– *LMI_MountPoint*
– *CIM_ElementInConnector*
| – *CIM_PackageInConnector*
| – *LMI_MemoryPhysicalPackageInConnector*
– *CIM_ElementSoftwareIdentity*
| – *LMI_DiskDriveElementSoftwareIdentity*
– *CIM_AbstractElementAllocatedFromPool*
| – *CIM_ElementAllocatedFromPool*
| – *CIM_AllocatedFromStoragePool*
| – *LMI_LVAllocatedFromStoragePool*
| – *LMI_VGAllocatedFromStoragePool*
– *LMI_AttachedFileSystem*
– *CIM_SAPSAPDependency*
| – *CIM_EndpointForIPNetworkConnection*
| | – *LMI_EndpointForIPNetworkConnection*
| – *CIM_BindsTo*
| | – *CIM_BindsToLANEndpoint*
| | | – *LMI_BindsToLANEndpoint*
| | – *LMI_LinkAggregationBindsTo*
| – *LMI_NetworkSAPSAPDependency*
– *CIM_ServiceSAPDependency*
| – *CIM_ForwardsAmong*
| – *CIM_SwitchesAmong*
| – *LMI_SwitchesAmong*
– *CIM_SystemPackaging*
| – *CIM_ComputerSystemPackage*
| – *LMI_ChassisComputerSystemPackage*
– *CIM_AssociatedMemory*
| – *CIM_AssociatedCacheMemory*
| – *LMI_AssociatedProcessorCacheMemory*
– *LMI_HostedMount*
– *CIM_ResidesOnExtent*
| – *LMI_ResidesOnExtent*
– *CIM_HostedDependency*
| – *CIM_HostedAccessPoint*
| | – *LMI_HostedSoftwareIdentityResource*
| | – *LMI_NetworkHostedAccessPoint*
| – *CIM_HostedService*
| | – *LMI_HostedIPConfigurationService*
| | – *LMI_HostedSSSDService*
| | – *LMI_HostedSystemService*
| | – *LMI_HostedStorageService*
| | – *LMI_HostedAccountManagementService*
| | – *LMI_HostedSoftwareInstallationService*
| | – *LMI_HostedSELinuxService*

| | – *LMI_HostedRealmdService*
| | – *LMI_HostedPowerManagementService*
| – *CIM_HostedCollection*
| – *LMI_HostedSoftwareCollection*
| – *LMI_HostedStorageStatisticsCollection*
– *CIM_InstalledPartitionTable*
– *LMI_InstalledPartitionTable*

CIM_ElementCapabilities
– *LMI_BlockStorageStatisticsElementCapabilities*
– *LMI_AssociatedSoftwareInstallationServiceCapabilities*
– *LMI_NetworkElementCapabilities*
– *LMI_FileSystemConfigurationElementCapabilities*
– *LMI_MDRAIDElementCapabilities*
– *LMI_LVElementCapabilities*
– *LMI_FileSystemElementCapabilities*
– *LMI_ProcessorElementCapabilities*
– *LMI_MountElementCapabilities*
– *LMI_VGElementCapabilities*
– *LMI_AccountManagementServiceCapabilities*
– *LMI_AccountCapabilities*
– *LMI_IPNetworkConnectionElementCapabilities*
– *LMI_ElementCapabilities*
– *LMI_DiskPartitionElementCapabilities*

CIM_ElementSettingData
– *LMI_FileSystemElementSettingData*
– *LMI_IPVersionElementSettingData*
– *LMI_MDRAIDElementSettingData*
– *LMI_AccountManagementServiceSettingData*
– *LMI_LVElementSettingData*
– *LMI_DiskPartitionElementSettingData*
– *LMI_IPElementSettingData*
– *LMI_VGElementSettingData*
– *LMI_MountedFileSystemElementSettingData*

CIM_Indication
– *CIM_InstIndication*
– *CIM_InstModification*
| – *LMI_NetworkInstModification*
| – *LMI_ServiceInstanceModificationIndication*
| – *LMI_SoftwareInstModification*
| – *LMI_SELinuxInstModification*
| – *LMI_StorageInstModification*
– *CIM_InstCreation*
| – *LMI_SoftwareInstCreation*
| – *LMI_JournalLogRecordInstanceCreationIndication*

| – *LMI_SELinuxInstCreation*
| – *LMI_AccountInstanceCreationIndication*
| – *LMI_StorageInstCreation*
| – *LMI_NetworkInstCreation*
– *CIM_InstMethodCall*
– *CIM_InstDeletion*
– *LMI_SELinuxInstDeletion*
– *LMI_SoftwareInstDeletion*
– *LMI_NetworkInstDeletion*
– *LMI_AccountInstanceDeletionIndication*

CIM_InstalledSoftwareIdentity
– *LMI_InstalledSoftwareIdentity*

CIM_LogicalIdentity
– *CIM_EndpointIdentity*
| – *LMI_EndpointIdentity*
– *CIM_ConcreteIdentity*
| – *LMI_LinkAggregationConcreteIdentity*
– *CIM_FileIdentity*
– *LMI_FileIdentity*

CIM_ManagedElement
– *LMI_SSSDDomain*
– *CIM_Identity*
| – *LMI_Identity*
– *CIM_SettingData*
| – *CIM_IPAssignmentSettingData*
| | – *LMI_IPAssignmentSettingData*
| | | – *LMI_BondingMasterSettingData*
| | | – *LMI_BridgingSlaveSettingData*
| | | – *LMI_BridgingMasterSettingData*
| | | – *LMI_BondingSlaveSettingData*
| | | – *LMI_IPRouteSettingData*
| | – *CIM_ExtendedStaticIPAssignmentSettingData*
| | | – *LMI_ExtendedStaticIPAssignmentSettingData*
| | – *CIM_DHCPSettingData*
| | | – *LMI_DHCPSettingData*
| | – *CIM_DNSSettingData*
| | – *LMI_DNSSettingData*
| – *CIM_IPVersionSettingData*
| | – *LMI_IPVersionSettingData*
| – *LMI_MountedFileSystemSetting*
| – *CIM_FileSystemSetting*
| | – *LMI_FileSystemSetting*
| – *CIM_StorageSetting*
| | – *LMI_StorageSetting*

– *CIM_RecordForLog*
| – *CIM_LogRecord*
| – *LMI_JournalLogRecord*
– *CIM_ManagedSystemElement*
| – *CIM_LogicalElement*
| | – *CIM_LogicalFile*
| | | – *CIM_FIFOPipeFile*
| | | | – *LMI_FIFOPipeFile*
| | | – *CIM_DeviceFile*
| | | | – *CIM_UnixDeviceFile*
| | | | – *LMI_UnixDeviceFile*
| | | – *CIM_Directory*
| | | | – *CIM_UnixDirectory*
| | | | – *LMI_UnixDirectory*
| | | – *CIM_DataFile*
| | | | – *LMI_UnixSocket*
| | | | – *LMI_DataFile*
| | | – *CIM_SymbolicLink*
| | | – *LMI_SymbolicLink*
| | – *CIM_EnabledLogicalElement*
| | | – *CIM_Account*
| | | | – *LMI_Account*
| | | – *CIM_Log*
| | | | – *CIM_MessageLog*
| | | | – *LMI_JournalMessageLog*
| | | – *LMI_DataFormat*
| | | | – *LMI_PVFormat*
| | | | – *LMI_EncryptionFormat*
| | | | | – *LMI_LUKSFormat*
| | | | – *LMI_MDRAIDFormat*
| | | – *CIM_Service*
| | | | – *LMI_Service*
| | | | – *CIM_SoftwareInstallationService*
| | | | | – *LMI_SoftwareInstallationService*
| | | | – *CIM_SecurityService*
| | | | | – *LMI_AccountManagementService*
| | | | – *LMI_RealmdService*
| | | | – *CIM_DiskPartitionConfigurationService*
| | | | | – *LMI_DiskPartitionConfigurationService*
| | | | – *CIM_StatisticsService*
| | | | | – *CIM_BlockStatisticsService*
| | | | | – *LMI_BlockStatisticsService*
| | | | – *LMI_SSSDService*
| | | | – *CIM_StorageConfigurationService*
| | | | | – *LMI_StorageConfigurationService*
| | | | – *CIM_FileSystemConfigurationService*
| | | | | – *LMI_FileSystemConfigurationService*

| | | | | | | – *LMI_PCIBridge*
| | | | | | – *LMI_PCIDevice*
| | | | | – *CIM_Sensor*
| | | | | | – *CIM_NumericSensor*
| | | | | | – *LMI_FanSensor*
| | | | | – *CIM_LogicalPort*
| | | | | – *CIM_NetworkPort*
| | | | | | – *CIM_EthernetPort*
| | | | | | – *LMI_EthernetPort*
| | | | | – *CIM_ATAPort*
| | | | | – *LMI_DiskDriveATAPort*
| | | | – *CIM_ServiceAccessPoint*
| | | | | – *CIM_IPNetworkConnection*
| | | | | | – *LMI_IPNetworkConnection*
| | | | | – *CIM_ProtocolEndpoint*
| | | | | | – *CIM_LinkAggregator8023ad*
| | | | | | | – *LMI_LinkAggregator8023ad*
| | | | | | – *CIM_SwitchPort*
| | | | | | | – *LMI_SwitchPort*
| | | | | | – *CIM_DNSProtocolEndpoint*
| | | | | | | – *LMI_DNSProtocolEndpoint*
| | | | | | – *CIM_IPProtocolEndpoint*
| | | | | | | – *LMI_IPProtocolEndpoint*
| | | | | | – *CIM_ATAProtocolEndpoint*
| | | | | | | – *LMI_DiskDriveATAProtocolEndpoint*
| | | | | | – *CIM_LAGPort8023ad*
| | | | | | | – *LMI_LAGPort8023ad*
| | | | | | – *CIM_LANEndpoint*
| | | | | | – *LMI_LANEndpoint*
| | | | | – *CIM_RemoteServiceAccessPoint*
| | | | | – *CIM_SoftwareIdentityResource*
| | | | | | – *LMI_SoftwareIdentityResource*
| | | | | – *LMI_NetworkRemoteServiceAccessPoint*
| | | | – *CIM_FileSystem*
| | | | – *CIM_LocalFileSystem*
| | | | – *LMI_LocalFileSystem*
| | | | – *LMI_TransientFileSystem*
| | | – *CIM_SoftwareIdentity*
| | | | – *LMI_SoftwareIdentity*
| | | | – *LMI_DiskDriveSoftwareIdentity*
| | | – *CIM_ResourcePool*
| | | | – *CIM_StoragePool*
| | | | – *LMI_VGStoragePool*
| | | – *CIM_UnixFile*
| | | | – *LMI_UnixFile*
| | | – *CIM_Job*
| | | – *CIM_ConcreteJob*

– *CIM_Setting*
| – *CIM_SystemSetting*
| – *LMI_Locale*
– *CIM_NextHopRoute*
| – *CIM_NextHopIPRoute*
| – *LMI_NextHopIPRoute*
– *CIM_BlockStatisticsManifest*
– *LMI_BlockStatisticsManifest*

CIM_MemberOfCollection
– *LMI_MemberOfGroup*
– *LMI_MemberOfStorageStatisticsCollection*
– *LMI_MemberOfBlockStatisticsManifestCollection*
– *LMI_MemberOfSoftwareCollection*

CIM_OwningCollectionElement
– *LMI_OwningGroup*

CIM_OwningJobElement
– *LMI_OwningJobElement*
– *LMI_OwningSoftwareJobElement*
– *LMI_OwningStorageJobElement*
– *LMI_OwningNetworkJobElement*

CIM_RecordInLog
– *LMI_JournalRecordInLog*

CIM_SAPAvailableForElement
– *LMI_ResourceForSoftwareIdentity*
– *LMI_DiskDriveSAPAvailableForElement*

CIM_ServiceAffectsElement
– *LMI_ServiceAffectsIdentity*
– *LMI_SoftwareInstallationServiceAffectsElement*
– *LMI_IPConfigurationServiceAffectsElement*

CIM_ServiceAvailableToElement
– *CIM_AssociatedPowerManagementService*
– *LMI_AssociatedPowerManagementService*

LMI_SSSDAvailableComponent

LMI_SSSDAvailableDomain

LMI_SSSDBackendDomain

LMI_SSSDBackendProvider

LMI_SSSDDomainSubdomain

LMI_SoftwareIdentityChecks

l

A

B

C

D

E

F

G

H

M

N

O

P

R

S

T

U

V

W