Lago Documentation

Release 0.40.0

David Caro

Jul 02, 2017

Contents

CHAPTER 1

Lago Introduction

Lago is an add-hoc virtual framework which helps you build virtualized environments on your server or laptop for various use cases.

It currently utilizes 'libvirt' for creating VMs, but we are working on adding more providers such as 'containers'.

CHAPTER 2

Getting started

Installing Lago

Lago is officially supported and tested on Fedora 24+ and CentOS 7.3 distributions. However, it should be fairly easy to get it running on debian variants.

As Lago requires libvirtd installed and several group permissions, it cannot be installed solely via `pip`. For that reason the recommended method of installation is using the RPM. The easiest way, is to use the *Install script* which we test and verify regularly[2].

pip

1. Install system package dependencies, this may vary according to your distribution:
   (a) CentOS 7.3+

```
$ yum install -y epel-release centos-release-qemu-ev
$ yum install -y libvirt libvirt-devel libguestfs-tools \
    libguestfs-devel gcc libffi-devel openssl-devel \
    qemu-kvm-ev
```

   (a) Fedora 24+

```
$ dnf install -y libvirt libvirt-devel libguestfs-tools \
    libguestfs-devel gcc libffi-devel openssl-devel qemu-kvm
```

   (a) Debian / Ubuntu - *TO-DO*
   (b) ArchLinux - *TO-DO*

[2] If the installation script does not work for you on the supported distributions, please open an issue at h ttps://github.com/lago-project/lago-demo.git

3. Install libguestfs Python bindings, as they are not available on PyPI[3]:

```
$ pip install http://download.libguestfs.org/python/guestfs-1.36.4.tar.gz
```

4. Install Lago with pip:

```
$ pip install lago
```

5. Setup permissions(replacing USERNAME accordingly):

```
$ sudo usermod -a -G qemu,libvirt USERNAME
$ sudo usermod -a -G USERNAME qemu
$ chmod g+x $HOME
```

6. Create a global share for Lago to store templates:

```
$ sudo mkdir -p /var/lib/lago
$ sudo mkdir -p /var/lib/lago/{repos,store,subnets}
$ sudo chown -R USERNAME:USERNAME /var/lib/lago
```

   Note: if you don't want to share the templates between users, have a look at the Configuration section, and change `lease_dir`, `template_repos` and `template_store` accordingly.

7. Restart libvirt(if you have systemd, otherwise use your distribution specific tool):

```
$ systemctl restart libvirtd
```

8. Log out and login again

Fedora 24+ / CentOS 7.3

Install script

1. Download the installation script and make it executable:

```
$ curl https://raw.githubusercontent.com/lago-project/lago-demo/master/install_
↪scripts/install_lago.sh \
    -o install_lago.sh \
    && chmod +x install_lago.sh
```

2. Run the installation script(replacing `username` with your username):

```
$ sudo ./install_lago.sh --user [username]
```

3. Log out and login again.

That's it! Lago should be installed.

Manual installation

1. Add the following repository to a new file at `/etc/yum.repos.d/lago.repo`:

   For Fedora:

[3] libguestfs Python bindings is unfortunately not available on PyPI, see https://bugzilla.redhat.com/show_bug.cgi?id=1075594 for current status. You may also use the system-wide package, if those are available for your distribution. In that case, if using a virtualenv, ensure you are creating it with '–system-site-packages' option. For Fedora/CentOS the package is named *python2-libguestfs*, and for Debian *python-guestfs*.

```
[lago]
baseurl=http://resources.ovirt.org/repos/lago/stable/0.0/rpm/fc$releasever
name=Lago
enabled=1
gpgcheck=0
```

For CentOS:

```
[lago]
baseurl=http://resources.ovirt.org/repos/lago/stable/0.0/rpm/el$releasever
name=Lago
enabled=1
gpgcheck=0
```

For CentOS only, you need **EPEL** and **centos-release-qemu-ev** repositories, those can be installed by running:

```
$ sudo yum install -y epel-release centos-release-qemu-ev
```

2. With the Lago repository configured, run(for Fedora use `dnf` instead):

```
$ sudo yum install -y lago
```

3. Setup group permissions:

```
$ sudo usermod -a -G lago USERNAME
$ sudo usermod -a -G qemu USERNAME
$ sudo usermod -a -G USERNAME qemu
```

4. Add group execution rights to your home directory:[1]

```
$ chmod g+x $HOME
```

5. Restart libvirtd:

```
$ sudo systemctl enable libvirtd
$ sudo systemctl restart libvirtd
```

6. Log out and login again.

FAQ

- *Q*: After using the install script, how do I fix the permissions for another username?

  A: Run:

  ```
  $ ./install_lago.sh -p --user [new_user]
  ```

- *Q*: Can Lago be installed in a Python virtualenv?

  ***A*: Follow the same procedure as in the *pip* instructions, only run the** pip installation under your virtualenv. Consult[3] if you want to install libguestfs Python bindings not from pip.

[1] For more information why this step is needed see https://libvirt.org/drvqemu.html, at the bottom of "POSIX users/groups" section.

Troubleshooting

- *Problem*: QEMU throws an error it can't access files in my home directory.

  Solution: Check again that you have setup the permissions described in the *Manual Installation* section. After doing that, log out and log in again. If QEMU has the proper permissions, the following command should work(replace `some/nested/path` with a directory inside your home directory):

```
$ sudo -u qemu ls $HOME/some/nested/path
```

LagoInitFile Specification

Note: this is work under progress, if you'd like to contribute to the documentation, please feel free to open a PR. In the meanwhile, we recommend looking at LagoInitFile examples available at:

https://github.com/lago-project/lago-examples/tree/master/init-files

Each environment in Lago is created from an init file, the recommended format is YAML, although at the moment of writing JSON is still supported. By default, Lago will look for a file named `LagoInitFile` in the directory it was triggered. However you can pick a different file by running:

```
$ lago init <FILENAME>
```

Sections

The init file is composed out of two major sections: domains, and nets. Each virtual machine you wish to create needs to be under the `domains` section. `nets` will define the network topology, and when you add a nic to a domain, it must be defined in the `nets` section.

Example:

```
domains:
  vm-el73:
    memory: 2048
    service_provider: systemd
    nics:
      - net: lago
    disks:
      - template_name: el7.3-base
        type: template
        name: root
        dev: vda
        format: qcow2
    artifacts:
      - /var/log
nets:
  lago:
    type: nat
    dhcp:
      start: 100
      end: 254
    management: true
    dns_domain_name: lago.local
```

domains

`<name>`: The name of the virtual machine.

memory(int) The virtual machine memory in GBs.

vcpu(int) Number of virtual CPUs.

service_provider(string) This will instruct which service provider to use when enabling services in the VM by calling *lago.plugins.vm.VMPlugin.service()*, Possible values: *systemd, sysvinit*.

cpu_model(string) CPU Family to emulate for the virtual machine. The list of supported types depends on your hardware and the libvirtd version you use, to list them you can run locally:

```
$ virsh cpu-models x86_64
```

cpu_custom(dict) This allows more fine-grained control of the CPU type, see CPU section for details.

nics(list) Network interfaces. Each network interface must be defined in the global *nets* section. By default each nic will be assigned an IP according to the network definition. However, you may also use static IPs here, by writing:

```
nics:
    - net: net-01
      ip:  192.168.220.2
```

The same network can be declared multiple times for each domain.

disks(list)

type Disk type, possible values:

template A Lago template, this would normally the bootable device.

file A local disk image. Lago will thinly provision it during init stage, this device will not be bootable. But can obviously be used for additional storage.

template_name(string) Applies only to disks of type `template`. This should be one of the available Lago templates, see Templates section for the list.

size(string) Disk size to thinly provision in GB. This is only supported in `file` disks.

format(string) *TO-DO: no docs yet.*.

device(string) Linux device: vda, sdb, etc. Using a device named "sd*" will use virtio-scsi.

build(list) This section should describe how to build/configure VMs. The build/configure action will happen during `init`.

virt-customize(dict) Instructions to pass to virt-customize, where the key is the name of the option and the value is the arguments for that option.

This operation is only supported on disks which contains OS.

A special instruction is `ssh-inject: ''` Which will ensure Lago's generated SSH keys will be injected into the VM. This is useful when you don't want to run the bootstrap stage.

For example:

```
- template_name: el7.3-base
  build:
      - virt-customize:
```

```
            ssh-inject: ''
            touch: [/root/file1, /root/file2]
```

See build section for details.

artifacts(list) Paths on the VM that Lago should collect when using *lago collect* from the CLI, or `collect_artifacts()` from the SDK.

groups(list) Groups this VM belongs to. This is most usefull when deploying the VM with Ansible.

bootstrap(bool) Whether to run bootstrap stage on the VM's template disk, defaults to True.

ssh-user(string) SSH user to use and configure, defaults to *root*

vm-provider(string) VM Provider plugin to use, defaults to *local-libvirt*.

vm-type(string) VM Plugin to use. A custom VM Plugin can be passed here, note that it needs to be available in your Python Entry points. See lago-ost-plugin for an example.

metadata(dict) *TO-DO: no docs yet.*.

nets

`<name>`: The name of the network.

type(string) Type of the network. May be *nat* or *bridge*.

Lago SDK

The SDK goal is to automate the creation of virtual environments, by using Lago directly from Python. Currently, most CLI operations are supported from the SDK, though not all of them(specifically, snapshot and export).

Starting an environment from the SDK

Prerequirements

1. Have Lago installed, see the installation notes.
2. Create a `LagoInitFile`, check out LagoInitFile syntax for more details.

Prepare the environment

Note: This example is available as a Jupyter notebook here or converted to reST here.

Assuming the `LagoInitFile` is saved in `/tmp/el7-init.yaml` and contains:

```
domains:
  vm01:
    memory: 1024
    nics:
      - net: lago
    disks:
      - template_name: el7.3-base
        type: template
        name: root
```

```
        dev: sda
        format: qcow2
nets:
  lago:
    type: nat
    dhcp:
      start: 100
      end: 254
```

Which is a simple setup, containing one CentOS 7.3 virtual machine and one management network. Then you start the environment by running:

```
import logging
from lago import sdk

env = sdk.init(config='/tmp/el7-init.yaml',
               workdir='/tmp/my_test_env',
               logfile='/tmp/lago.log',
               loglevel=logging.DEBUG)
```

Where:

1. `config` is the path to a valid init file, in YAML format.
2. `workdir` is the place Lago will use to save the images and metadata.
3. The `logfile` and `loglevel` parameters add a FileHandler to Lago's root logger.

Note that if this is the first time you are running Lago it will first download the template(in this example `el7-base`), which might take a while[1]. You can follow up the progress by watching the log file, or alternatively if working in an interactive session, by running:

```
from lago import sdk
sdk.add_stream_logger()
```

Which will print all the Lago operations to stdout.

Starting the environment

Once `init()` method returns, the environment is ready to be started, taking up from the last example, executing:

```
env.start()
```

Would start the VMs specified in the init file, and make them available(among others) through SSH:

```
>>> vm = env.get_vms()['vm01']
>>> vm.ssh(['hostname', '-f'])
CommandStatus(code=0, out='vm01.lago.local\n', err='')
```

You can also run an interactive SSH session:

```
>>> res = vm.interactive_ssh()
[root@vm01 ~]# ls -lsah
total 20K
0 dr-xr-x---.  3 root root 103 May 28 03:11 .
```

[1] On a normal setup, where the templates are already downloaded, the `init` stage should take less than a minute(but probably at least 15 seconds).

```
0 dr-xr-xr-x. 17 root root 224 Dec 12 17:00 ..
4.0K -rw-r--r--.  1 root root  18 Dec 28  2013 .bash_logout
4.0K -rw-r--r--.  1 root root 176 Dec 28  2013 .bash_profile
4.0K -rw-r--r--.  1 root root 176 Dec 28  2013 .bashrc
4.0K -rw-r--r--.  1 root root 100 Dec 28  2013 .cshrc
0 drwx------.  2 root root  29 May 28 03:11 .ssh
4.0K -rw-r--r--.  1 root root 129 Dec 28  2013 .tcshrc
[root@vm01 ~]# exit
exit
>>> res.code
0
```

Controlling the environment

You can start or stop the environment by calling `start()` and `stop()`, finally you can destroy the environment with `lago.sdk.SDK.destroy()` method, note that it will stop all VMs, and remove the provided working directory.

```
>>> env.destroy()
>>>
```

Disk consumption for the workdir

Generally speaking, the workdir disk consumption depends on which operation you run inside the underlying VMs. Lago uses QCOW2 layered images by default, so that each environment you create, sets up its own layer on top of the original template Lago downloaded the first time `init` was ran with that specific template. So when the VM starts, it usually consumes less than 30MB. As you do more operations - the size might increase, as your current image diverges from the original template. For more information see qemu-img

Differences from the CLI

1. Creating Different `prefixes` inside the workdir is not supported. In the CLI, you can have several prefixes inside a `workdir`. The reasoning behind that is that when working from Python, you can manage the environment directly by your own(using a temporary or fixed path).
2. Logging - In the CLI, all log operations are kept in the current `prefix` under *logs/lago.log* path. The SDK keeps that convention, but allows you to add additional log files by passing log filename and level parameters to `init()` command. Additionally, you can work in debug mode, by logging all commands to stdout and stderr, calling the module-level method `add_stream_logger()`. Note that this will log everything for all environments.
3. `Prefix` class. This is more of an implementation issue: the core per-environment operations are exposed both for the CLI and SDK in that class. In order to provide consistency and ease of use for the SDK, only the methods which make sense for SDK usage are exposed in the SDK, the CLI does not require that, as the methods aren't exposed at all(only verbs in `py`.

Getting started with some Lago Examples!

Get Lago up & running in no time using one of the available examples

Important: make sure you followed the installation step before to have Lago installed.

Available Examples

- LagoInitFiles examples
- Simple Jenkins server + slaves: Jenkins_Example
- Advanced oVirt example (using nested virtualization): oVirt_Example
- SDK Usage example - GitHub, or in reST
- Integrating Lago with Pytest

Lago Templates

We maintain several templates which are publicly available here, and Lago will use them by default. We try to ensure each of those templates is fully functional out of the box. All templates are more or less the same as the standard cloud image for each distribution.

The templates specification and build scripts are managed in a different repository, and it should be fairly easy to create your own templates repository.

Available templates

Template name	OS
el7-base	CentOS 7.2
el7.3-base	CentOS 7.3
fc23-base	Fedora 23
fc24-base	Fedora 24
fc25-base	Fedora 25
el6-base	CentOS 6.7
debian8-base	Debian 8
ubuntu16.04-base	Ubuntu 16.04

Configuration

The recommend method to override the configuration file is by letting lago auto-generate them:

```
$ mkdir -p $HOME/.config/lago
$ lago generate-config > $HOME/.config/lago/lago.conf
```

This will dump the current configuration to `$HOME/.config/lago/lago.conf`, and you may edit it to change any parameters. Take into account you should probably comment out parameters you don't want to change when editing the file. Also, all parameters in the configuration files can be overridden by passing command line arguments or with environment variables, as described below.

lago.conf format

Lago runs without a configuration file by default, for reference-purposes, when lago is installed from the official packages(RPM or DEB), a commented-out version of lago.conf(INI format) is installed at `/etc/lago/lago.conf`.

In `lago.conf` global parameters are found under the `[lago]` section. All other sections usually map to subcommands(i.e. `lago init` command would be under `[init]` section).

Example:

```
$ lago generate-config
> [lago]
> # log level to use
> loglevel = info
> logdepth = 3
> ....
> [init]
> # location to store repos
> template_repos = /var/lib/lago/repos
> ...
```

lago.conf look-up

Lago attempts to look `lago.conf` in the following order:

1. `/etc/lago/lago.conf`
2. According to XDG standards , which are by default:
    - `/etc/xdg/lago/lago.conf`
    - `/home/$USER/.config/lago/lago.conf`
3. Any environment variables.
4. CLI passed arguments.

If more than one file exists, all files are merged, with the last occurrence of any parameter found used.

Overriding parameters with environment variables

To differentiate between the root section in the configuration file, lago uses the following format to look for environment variables:

```
'LAGO_GLOBAL_VAR' -> variable in [lago] section
'LAGO__SUBCOMMAND__PARAM_1' -> variable in [subcommand] section
```

Example: changing the `template_store` which `init` subcommand uses to store templates:

```
# check current value:
$ lago generate-config | grep -A4 "init"
> [init]
> # location to store repos
> template_repos = /var/lib/lago/repos
> # location to store temp
> template_store = /var/lib/lago/store

$ export LAGO__INIT__TEMPLATE_STORE=/var/tmp
$ lago generate-config | grep -A4 "init"
> [init]
> # location to store repos
> template_repos = /var/lib/lago/repos
```

```
> # location to store temp
> template_store = /var/tmp
```

Lago build

Lago allows to build / configure VM disks during init stage. In the init file, the key `build` should be added to each disk that needs to be configured.

`build` should map to a list of Builders, where each builder in the list is a one entry dictionary that maps to a dictionary that holds the options for that builder.

Options are key-value pairs, where the key is the name of the option (without leading dashes), and the value is the argument for that option. If the option takes no arguments, the empty string should be set as the value. If the builder allows specifying an options multiple times, the value should be a list of arguments.

Note: The build process runs "behind-the-back" of the OS (Before the VM starts), thus should be used with care.

Builders

Builders are commands that can be used to build/configure VMs. Builders are called by the order they appear in the init file.

virt-customize

A tool for customizing a virtual machine (install packages, copying files, etc...). *virt-customize* is part of the *libguestfs* tool set which is part of Lago's dependencies.

virt-customize can be called only on disks which contains an OS.

Depends on the version of *virt-customize* installed on your system (it can vary between different OS), all the valid options for *virt-customize* can be specified in the init file. For the full list of options please refer to virt-customize documentation.

There is a special case when using *virt-customize* to inject ssh keys. If the empty string is provided to `ssh-inject` option, Lago will replace it with the path to lago's self generated ssh keys.

Note: If SELinux is enabled in a VM, it's possible that `selinux-relabel` will be required after adding / modifiyng its files.

Relation to bootstrap

Configuration is taking place after Lago runs bootstrap. You can disable bootstrap to all VMs by passing `--skip-bootstrap` to `lago init`, or by adding `bootstrap: false` to the VM definition in the init file (the second allows to control bootstrap per VM).

Since bootstrap is injecting ssh keys to the VMs, If skipping it, it's recommended to inject the ssh keys using *virt-customize* builder otherwise, shell access to the VM will use password authentication (more details can be found in the Builders sections in this documents).

Example

In the following example, *virt-customize* builder will be called on the disk of vm01.

The changes will be:

- Injecting lago's self generated ssh keys.
- Copy `dummy_file` from the host to `/root` in `vm01`
- Create files `/root/file1` and `/root/file2` in `vm01`
- Finish with SELinux relabel of `vm01`.

```
domains:
  vm01:
    artifacts: [/var/log]
    bootstrap: false
    disks:
    - build:
      - virt-customize:
          ssh-inject: ''
          copy: dummy_file:/root
          touch: [/root/file1, /root/file2]
          selinux-relabel: ''
      dev: vda
      format: qcow2
      name: root
      path: $LAGO_INITFILE_PATH/lago-basic-suite-4-1-engine_root.qcow2
      template_name: el7.3-base
      template_type: qcow2
      type: template
```

Lago CPU Models in detail

There are several ways you can configure the CPU model Lago will use for each VM. This section tries to explain more in-depth how it will be mapped to libvirt XML.

- `vcpu`: Number of virtual CPUs.
- `cpu_model`: `<model>`: This defines an exact match of a CPU model. The generated Libvirt `<cpu>` XML will be:

  ```
  <cpu>
      <model>Westmere</model>
      <topology cores="1" sockets="3" threads="1"/>
      <feature name="vmx" policy="require"/>
  </cpu>
  ```

  If the vendor of the host CPU and the selected model match, it will attempt to require `vmx` on Intel CPUs and `svm` on AMD CPUs, assuming the host CPU has that feature. The topology node will be generated with sockets equals to `vcpu` parameter, by default it is set to `2`.

- `cpu_custom`: This allows to override entirely the CPU definition, by writing the domain CPU XML in YAML syntax, for example, for the following LagoInitFile:

  ```
  domains:
    vm-el73:
  ```

```
    vcpu: 2
    cpu_custom:
      '@mode': custom
      '@match': exact
      model:
        '@fallback': allow
        '#text': Westmere
      feature:
        '@policy': optional
        '@name': 'vmx'
      numa:
        cell:
          -
            '@id': 0
            '@cpus': 0
            '@memory': 2048
            '@unit': 'MiB'
          -
            '@id': 1
            '@cpus': 1
            '@memory': 2048
            '@unit': 'MiB'
  ...
```

This will be the generated <cpu> XML:

```
<cpu mode="custom" match="exact">
    <model fallback="allow">Westmere</model>
    <feature policy="optional" name="vmx"/>
    <numa>
        <cell id="0" cpus="0" memory="2048" unit="MiB"/>
        <cell id="1" cpus="1" memory="2048" unit="MiB"/>
    </numa>
    <topology cores="1" sockets="2" threads="1"/>
</cpu>
<vcpu>2</vcpu>
```

The conversion is pretty straight-forward, `@` maps to attribute, and `#text` to text fields. If `topology` section is not defined, it will be added.

- No `cpu_custom` or `cpu_model`: Then Libvirt's `host-passthrough` will be used. For more information see: Libvirt CPU model

CHAPTER 3

Developing

CI Process

Here is described the usual workflow of going through the CI process from starting a new branch to getting it merged and released in the unstable repo.

Starting a branch

First of all, when starting to work on a new feature or fix, you have to start a new branch (in your fork if you don't have push rights to the main repo). Make sure that your branch is up to date with the project's master:

```
git checkout -b my_fancy_feature
# in case that origin is already lago-project/lago
git reset --hard origin/master
```

Then, once you can just start working, doing commits to that branch, and pushing to the remote from time to time as a backup.

Once you are ready to run the ci tests, you can create a pull request to master branch, if you have hub installed you can do so from command line, if not use the ui:

```
$ hub pull-request
```

That will automatically trigger a test run on ci, you'll see the status of the run in the pull request page. At that point, you can keep working on your branch, probably just rebasing on master regularly and maybe amending/squashing commits so they are logically meaningful.

A clean commit history

An example of not good pull request history:

- Added right_now parameter to virt.VM.start function

- Merged master into my_fancy_feature
- Added tests for the new parameter case
- Renamed right_now parameter to sudo_right_now
- Merged master into my_fancy_feature
- Adapted test to the rename

This history can be greatly improved if you squashed a few commits:

- Added sudo_right_now parameter to virt.VM.start function
- Added tests for the new parameter case
- Merged master into my_fancy_feature
- Merged master into my_fancy_feature

And even more if instead of merging master, you just rebased:

- Added sudo_right_now parameter to virt.VM.start function
- Added tests for the new parameter case

That looks like a meaningful history :)

Rerunning the tests

While working on your branch, you might want to rerun the tests at some point, to do so, you just have to add a new comment to the pull request with one of the following as content:

- ci test please
- ci :+1:
- ci :thumbsup:

Asking for reviews

If at any point, you see that you are not getting reviews, please add the label 'needs review' to flag that pull request as ready for review.

Getting the pull request merged

Once the pull request has been reviewed and passes all the tests, an admin can start the merge process by adding a comment with one of the following as content:

- ci merge please
- ci :shipit:

That will trigger the merge pipeline, that will run the tests on the merge commit and deploy the artifacts to the unstable repo on success.

Environment setup

Here are some guidelines on how to set up your development of the lago project.

Requirements

You'll need some extra packages to get started with the code for lago, assuming you are runnig Fedora:

```
> sudo dnf install git mock libvirt-daemon qemu-kvm autotools
```

And you'll need also a few Python libs, which you can install from the repos or use venv or similar, for the sake of this example we will use the repos ones:

```
> sudo dnf install python-flake8 python-nose python-dulwich yapf
```

Yapf is not available on older Fedoras or CentOS, you can get it from the official yapf repo or try on copr.

Now you are ready to get the code:

```
> git clone git@github.com:lago-project/lago.git
```

From now on all the commands will be based from the root of the cloned repo:

```
> cd lago
```

Style formatting

We will accept only patches that pass pep8 and that are formatted with yapf. More specifically, only patches that pass the local tests:

```
> make check-local
```

It's recommended that you setup your editor to check automatically for pep8 issues. For the yapf formatting, if you don't want to forget about it, you can install the pre-commit git hook that comes with the project code:

```
> ln -s scripts/pre-commit.style .git/pre-commit
```

Now each time that you run *git commit* it will automatically reformat the code you changed with yapf so you don't have any issues when submitting a patch.

Testing your changes

Once you do some changes, you should make sure they pass the checks, there's no need to run on each edition but before submitting a patch for review you should do it.

You can run them on your local machine, but the tests themselves will install packages and do some changes to the os, so it's really recommmended that you use a vm, or as we do on the CI server, use mock chroots. If you don't want to setup mock, skip the next section.

Hopefully in a close future we can use lago for that ;)

Setting up mock_runner.sh with mock (fedora)

For now we are using a script developed by the *oVirt* devels to generate chroots and run tests inside them, it's not packaged yet, so we must get the code itself:

```
> cd ..
> git clone git://gerrit.ovirt.org/jenkins
```

As an alternative, you can just download the script and install them in your *$PATH*:

```
> wget https://gerrit.ovirt.org/gitweb?p=jenkins.git;a=blob_plain;f=mock_configs/mock_
↪runner.sh;hb=refs/heads/master
```

We will need some extra packages:

```
> sudo dnf install mock
```

And, if not running as root (you shouldn't!) you have to add your user to the newly created mock group, and make sure the current session is in that group:

```
> sudo usermod -a -G mock $USER
> newgrp mock
> id  # check that mock is listed
```

Running the tests inside mock

Now we have all the setup we needed, so we can go back to the lago repo and run the tests, the first time you run them, it will take a while to download all the required packages and install them in the chroot, but on consecutive runs it will reuse all the cached chroots.

The *mock_runner.sh* script allows us to test also different distributions, any that is supported by mock, for example, to run the tests for fedora 23 you can run:

```
> ../jenkins/mock_runner.sh -p fc23
```

That will run all the *check-patch.sh* (the *-p* option) tests inside a chroot, with a minimal fedora 23 installation. It will leave any logs under the *logs* directory and any generated artifacts under *exported-artifacts*.

Getting started developing

Everyone is welcome to send patches to lago, but we know that not everybody knows everything, so here's a reference list of technologies and methodologies that lago uses for reference.

Python!

Lago is written in python 2.7 (for now), so you should get yourself used to basic-to-medium python constructs and technics like:

- Basic python: Built-in types, flow control, pythonisms (import this)
- Object oriented programming (OOP) in python: Magic methods, class inheritance

Some useful resources:

- Base docs: https://docs.python.org/2.7/
- Built-in types: https://docs.python.org/2.7/library/stdtypes.html
- About classes: https://docs.python.org/2.7/reference/datamodel.html#new-style-and-classic-classes
- The Zen of Python:

```
> python -c "import this"

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Bash

Even though there is not much bash code, the functional tests and some support scripts use it, so better to get some basics on it. We will try to follow the same standards for it than the oVirt project has.

Libvirt + qemu/kvm

As we are using intesively libvirt and qemu/kvm, it's a good idea to get yourself familiar with the main commands and services:

- libvirt: http://libvirt.org
- virsh client: http://libvirt.org/virshcmdref.html
- qemu (qemu-img is useful to deal with vm disk images): https://en.wikibooks.org/wiki/QEMU/Images

Also, there's a library and a set of tools from the libguestfs project that we use to prepare templates and are very useful when debugging, make sure you play at least with virt-builder, virt-customize, virt-sparsify and guestmount.

Git + Github

We use git as code version system, and we host it on Github right now, so if you are not familiar with any of those tools, you should get started with them, specially you should be able to:

- Clone a repo from github
- Fork a repo from github
- Create/delete/move to branches (git checkout)
- Move to different points in git history (git reset)
- Create/delete tags (git tag)

- See the history (git log)
- Create/amend commits (git commit)
- Retrieve changes from the upstream repository (git fetch)
- Apply your changes on top of the retrieved ones (git rebase)
- Apply your changes as a merge commit (git merge)
- Squash/reorder existing commits (git rebase –interactive)
- Send your changes to the upstream (git push)
- Create a pull request

You can always go to the git docs though there is a lot of good literature on it too.

Unit tests with py.test

Lately we decided to use py.test for the unit tests, and all the current unit tests were migrated to it. We encourage adding unit tests to any pull requests you send.

Functional tests with bats

For the functional tests, we decided to use bats framework. It's completely written in bash, and if you are modifying or adding any functionality, you should add/modify those tests accordingly. It has a couple of custom constructs, so take a look to the bats docs while reading/writing tests.

Packaging

Our preferred distribution vector is though packages. Right now we are only building for rpm-based system, so right now you can just take a peek on how to build rpms. Keep in mind also that we try to move as much of the packaging logic as posible to the python packaging system itself too, worth getting used to it too.

Where to go next

You can continue setting up your environment and try running the examples in the readme to get used to lago. Once you get familiar with it, you can pick any of the existing issues and send a pull request to fix it, so you get used to the ci process we use to get stuff developed flawlessly and quickly, welcome!

CHAPTER 4

Contents

lago package

Subpackages

lago.plugins package

exception lago.plugins.**NoSuchPluginError**
Bases: *lago.plugins.PluginError*

lago.plugins.**PLUGIN_ENTRY_POINTS = {'vm': 'lago.plugins.vm', 'vm-service': 'lago.plugins.vm_service', 'vm-provider**
Map of plugin type string -> setuptools entry point

class lago.plugins.**Plugin**
Bases: object

Base class for all the plugins

exception lago.plugins.**PluginError**
Bases: exceptions.Exception

lago.plugins.**load_plugins**(*namespace*, *instantiate=True*)
Loads all the plugins for the given namespace

Parameters

- **namespace** (*str*) – Namespace string, as in the setuptools entry_points
- **instantiate** (*bool*) – If true, will instantiate the plugins too

Returns Returns the list of loaded plugins

Return type dict of str, object

Submodules

lago.plugins.cli module

About CLIPlugins

A CLIPlugin is a subcommand of the lagocli command, it's ment to group actions together in a logical sense, for example grouping all the actions done to templates.

To create a new subcommand for testenvcli you just have to subclass the CLIPlugin abstract class and declare it in the setuptools as an entry_point, see this module's setup.py/setup.cfg for an example:

```
class NoopCLIplugin(CLIPlugin):
    init_args = {
        'help': 'dummy help string',
    }

    def populate_parser(self, parser):
        parser.addArgument('--dummy-flag', action='store_true')

    def do_run(self, args):
        if args.dummy_flag:
            print "Dummy flag passed to noop subcommand!"
        else:
            print "Dummy flag not passed to noop subcommand!"
```

You can also use decorators instead, an equivalent is:

```
@cli_plugin_add_argument('--dummy-flag', action='store_true')
@cli_plugin(help='dummy help string')
def my_fancy_plugin_func(dummy_flag, **kwargs):
    if dummy_flag:
        print "Dummy flag passed to noop subcommand!"
    else:
        print "Dummy flag not passed to noop subcommand!"
```

Or:

```
@cli_plugin_add_argument('--dummy-flag', action='store_true')
def my_fancy_plugin_func(dummy_flag, **kwargs):
    "dummy help string"
    if dummy_flag:
        print "Dummy flag passed to noop subcommand!"
    else:
        print "Dummy flag not passed to noop subcommand!"
```

Then you will need to add an entry_points section in your setup.py like:

```
setup(
    ...
    entry_points={
        'lago.plugins.cli': [
            'noop=noop_module:my_fancy_plugin_func',
        ],
    }
    ...
)
```

Or in your setup.cfg like:

```
[entry_points]
lago.plugins.cli =
    noop=noop_module:my_fancy_plugin_func
```

Any of those will add a new subcommand to the lagocli command that can be run as:

```
$ lagocli noop
Dummy flag not passed to noop subcommand!
```

TODO: Allow per-plugin namespacing to get rid of the ***kwargs* parameter

class lago.plugins.cli.**CLIPlugin**
Bases: *lago.plugins.Plugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

do_run(*args*)
Execute any actions given the arguments

Parameters **args** (*Namespace*) – with the arguments

Returns None

init_args
Dictionary with the argument to initialize the cli parser (for example, the help argument)

populate_parser(*parser*)
Add any required arguments to the parser

Parameters **parser** (*ArgumentParser*) – parser to add the arguments to

Returns None

class lago.plugins.cli.**CLIPluginFuncWrapper**(*do_run=None*, *init_args=None*)
Bases: *lago.plugins.cli.CLIPlugin*

Special class to handle decorated cli plugins, take into account that the decorated functions have some limitations on what arguments can they define actually, if you need something complicated, used the abstract class *CLIPlugin* instead.

Keep in mind that right now the decorated function must use ***kwargs* as param, as it will be passed all the members of the parser, not just whatever it defined

__call__(**args*, ***kwargs*)
Keep the original function interface, so it can be used elsewhere

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

add_argument(**argument_args*, ***argument_kwargs*)

do_run(*args*)

init_args

populate_parser(*parser*)

set_help(*help=None*)

set_init_args(*init_args*)

lago.plugins.cli.**cli_plugin**(*func=None*, ***kwargs*)

Decorator that wraps the given function in a *CLIPlugin*

Parameters

- **func** (*callable*) – function/class to decorate
- ****kwargs** – Any other arg to use when initializing the parser (like help, or prefix_chars)

Returns cli plugin that handles that method

Return type *CLIPlugin*

Notes

It can be used as a decorator or as a decorator generator, if used as a decorator generator don't pass any parameters

Examples

```
>>> @cli_plugin
... def test(**kwargs):
...     print 'test'
...
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
```

```
>>> @cli_plugin()
... def test(**kwargs):
...     print 'test'
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
```

```
>>> @cli_plugin(help='dummy help')
... def test(**kwargs):
...     print 'test'
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
>>> print test.init_args['help']
'dummy help'
```

lago.plugins.cli.**cli_plugin_add_argument**(**args*, ***kwargs*)

Decorator generator that adds an argument to the cli plugin based on the decorated function

Parameters

- ***args** – Any args to be passed to `argparse.ArgumentParser.add_argument()`
- ***kwargs** – Any keyword args to be passed to `argparse.ArgumentParser.add_argument()`

Returns

Decorator that builds or extends the cliplugin for the decorated function, adding the given argument definition

Return type function

Examples

```
>>> @cli_plugin_add_argument('-m', '--mogambo', action='store_true')
... def test(**kwargs):
...     print 'test'
...
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
>>> print test._parser_args
[(('-m', '--mogambo'), {'action': 'store_true'})]
```

```
>>> @cli_plugin_add_argument('-m', '--mogambo', action='store_true')
... @cli_plugin_add_argument('-b', '--bogabmo', action='store_false')
... @cli_plugin
... def test(**kwargs):
...     print 'test'
...
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
>>> print test._parser_args
[(('-b', '--bogabmo'), {'action': 'store_false'}),
 (('-m', '--mogambo'), {'action': 'store_true'})]
```

lago.plugins.cli.**cli_plugin_add_help**(*help*)

Decorator generator that adds the cli help to the cli plugin based on the decorated function

Parameters **help** (*str*) – help string for the cli plugin

Returns

Decorator that builds or extends the cliplugin for the decorated function, setting the given help

Return type function

Examples

```
>>> @cli_plugin_add_help('my help string')
... def test(**kwargs):
...     print 'test'
...
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
>>> print test.help
my help string
```

```
>>> @cli_plugin_add_help('my help string')
... @cli_plugin()
... def test(**kwargs):
```

```
...     print 'test'
>>> print test.__class__
<class 'cli.CLIPluginFuncWrapper'>
>>> print test.help
my help string
```

lago.plugins.output module

About OutFormatPlugins

An OutFormatPlugin is used to format the output of the commands that extract information from the perfixes, like status.

class lago.plugins.output.**DefaultOutFormatPlugin**
Bases: *lago.plugins.output.OutFormatPlugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

format(*info_obj*, *indent=''*)

indent_unit = ' '

class lago.plugins.output.**FlatOutFormatPlugin**
Bases: *lago.plugins.output.OutFormatPlugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

format(*info_dict*, *delimiter='/'*)
This formatter will take a data structure that represent a tree and will print all the paths from the root to the leaves

in our case it will print each value and the keys that needed to get to it, for example:

vm0: net: lago memory: 1024

will be output as:

vm0/net/lago vm0/memory/1024

Args: info_dict (dict): information to reformat delimiter (str): a delimiter for the path components

Returns: str: String representing the formatted info

class lago.plugins.output.**JSONOutFormatPlugin**
Bases: *lago.plugins.output.OutFormatPlugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

format(*info_dict*)

class lago.plugins.output.**OutFormatPlugin**
Bases: *lago.plugins.Plugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

format(*info_dict*)
Execute any actions given the arguments

Parameters **info_dict** (*dict*) – information to reformat

Returns String representing the formatted info

Return type str

class lago.plugins.output.**YAMLOutFormatPlugin**
Bases: *lago.plugins.output.OutFormatPlugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

format(*info_dict*)

lago.plugins.service module

Service Plugin

This plugins are used in order to manage services in the vms

class lago.plugins.service.**ServicePlugin**(*vm*, *name*)
Bases: *lago.plugins.Plugin*

BIN_PATH
Path to the binary used to manage services in the vm, will be checked for existence when trying to decide if the serviece is supported on the VM (see *func:is_supported*).

Returns Full path to the binary insithe the domain

Return type str

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

_request_start()
Low level implementation of the service start request, used by the *func:start* method

Returns True if the service succeeded to start, False otherwise

Return type bool

_request_stop()
Low level implementation of the service stop request, used by the *func:stop* method

Returns True if the service succeeded to stop, False otherwise

Return type bool

alive()

exists()

classmethod is_supported(*vm*)

start()

state()
Check the current status of the service

Returns Which state the service is at right now

Return type *ServiceState*

stop()

class lago.plugins.service.**ServiceState**
Bases: enum.Enum

ACTIVE = 2

INACTIVE = 1

MISSING = 0
This state corresponds to a service that is not available in the domain

_member_map_ = OrderedDict([('MISSING', <ServiceState.MISSING: 0>), ('INACTIVE', <ServiceState.INACTIVE: 1

_member_names_ = ['MISSING', 'INACTIVE', 'ACTIVE']

_member_type_
alias of object

_value2member_map_ = {0: <ServiceState.MISSING: 0>, 1: <ServiceState.INACTIVE: 1>, 2: <ServiceState.ACTIVE:

lago.plugins.vm module

VM Plugins

There are two VM-related plugin extension points, there's the VM Type Plugin, that allows you to modify at a higher level the inner workings of the VM class (domain concept in the initfile). The other plugin extension point, the [VM Provider Plugin], that allows you to create an alternative implementation of the provisioning details for the VM, for example, using a remote libvirt instance or similar.

exception lago.plugins.vm.**ExtractPathError**
Bases: *lago.plugins.vm.VMError*

exception lago.plugins.vm.**ExtractPathNoPathError**
Bases: *lago.plugins.vm.VMError*

exception lago.plugins.vm.**VMError**
Bases: exceptions.Exception

class lago.plugins.vm.**VMPlugin**(*env*, *spec*)

Bases: *lago.plugins.Plugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

_artifact_paths()

_detect_service_provider()

_get_service_provider()

NOTE: Can be reduced to just one get call once we remove support for the service_class spec entry

Returns class for the loaded provider for that vm_spec None: if no provider was specified in the vm_spec

Return type class

_get_vm_provider()

classmethod _normalize_spec(*spec*)

_scp(**args*, ***kwds*)

_template_metadata()

alive()

all_ips()

bootstrap(**args*, ***kwargs*)

Thin method that just uses the provider

collect_artifacts(*host_path*, *ignore_nopath*)

copy_from(*remote_path*, *local_path*, *recursive=True*, *propagate_fail=True*)

copy_to(*local_path*, *remote_path*, *recursive=True*)

create_snapshot(*name*, **args*, ***kwargs*)

Thin method that just uses the provider

defined(**args*, ***kwargs*)

Thin method that just uses the provider

disks

distro()

export_disks(*standalone=True*, *dst_dir=None*, *compress=False*, *collect_only=False*, *with_threads=True*, **args*, ***kwargs*)

Thin method that just uses the provider

extract_paths(*paths*, **args*, ***kwargs*)

Thin method that just uses the provider

extract_paths_dead(*paths*, **args*, ***kwargs*)

Thin method that just uses the provider

groups

Returns –

list of str: The names of the groups to which this vm belongs (as specified in the init file)

guest_agent()

has_guest_agent()

interactive_console(**args*, ***kwargs*)
Thin method that just uses the provider

interactive_ssh(**args*, ***kwargs*)

ip()

iscsi_name()

metadata

mgmt_name

mgmt_net

name()

nets()

nics()

reboot(**args*, ***kwargs*)
Thin method that just uses the provider

revert_snapshot(*name*, **args*, ***kwargs*)
Thin method that just uses the provider

root_password()

save(*path=None*)

service(**args*, ***kwargs*)

shutdown(**args*, ***kwargs*)
Thin method that just uses the provider

spec

ssh(*command*, *data=None*, *show_output=True*, *propagate_fail=True*, *tries=None*)

ssh_reachable(**args*, ***kwargs*)
Check if the VM is reachable with ssh

Parameters

- **tries** (*int*) – Number of tries to try connecting to the host
- **propagate_fail** (*bool*) – If set to true, this event will appear
- **the log and fail the outter stage. Otherwise, it will be** (*in*) –
- **discarded.** –

Returns True if the VM is reachable.

Return type bool

ssh_script(*path*, *show_output=True*)

start(**args*, ***kwargs*)
Thin method that just uses the provider

state(**args*, ***kwargs*)
Thin method that just uses the provider

stop(**args*, ***kwargs*)
Thin method that just uses the provider

vm_type

wait_for_ssh()

class lago.plugins.vm.**VMProviderPlugin**(*vm*)
Bases: *lago.plugins.Plugin*

If you want to use a custom provider for you VMs (say, ovirt for example), you have to inherit from this class, and then define the 'default_vm_provider' in your config to be your plugin, or explicitly specify it on each domain definition in the initfile with 'vm-provider' key

You will have to override at least all the abstractmethods in order to write a provider plugin, even if they are just runnig *pass*.

_extract_paths_scp(*paths*, *ignore_nopath*)

bootstrap(**args*, ***kwargs*)
Does any actions needed to get the domain ready to be used, ran on prefix init.

Returns None

create_snapshot(*name*, **args*, ***kwargs*)
Take any actions needed to create a snapshot

Parameters **name** (*str*) – Name for the snapshot, will be used as key to retrieve it later

Returns None

defined(**args*, ***kwargs*)
Return if the domain is defined (libvirt concept), currently used only by the libvirt provider, put here to allow backwards compatibility.

Returns True if the domain is already defined (libvirt concept)

Return type bool

export_disks(*standalone*, *dst_dir*, *compress*, **args*, ***kwargs*)
Export 'disks' as a standalone image or a layered image.

Parameters

- **disks** (*list*) – The names of the disks to export (None means all the disks)
- **standalone** (*bool*) – If true create a copy of the layered image else create a new disk which is a combination of the current layer and the base disk.
- **dst_dir** (*str*) – dir to place the exported images
- **compress** (*bool*) – if true, compress the exported image.

extract_paths(*paths*, *ignore_nopath*)
Extract the given paths from the domain

Parameters

- **paths** (*list of str*) – paths to extract
- **ignore_nopath** (*boolean*) – if True will ignore none existing paths.

Returns None

Raises

- *ExtractPathNoPathError* – if a none existing path was found on the VM, and `ignore_nopath` is True.
- *ExtractPathError* – on all other failures.

extract_paths_dead(*paths*, *ignore_nopath*)
Extract the given paths from the domain, without the underlying OS awareness

interactive_console()
Run an interactive console

Returns resulf of the interactive execution

Return type *lago.utils.CommandStatus*

reboot(**args*, ***kwargs*)
Reboot a domain

Returns None

revert_snapshot(*name*, **args*, ***kwargs*)
Take any actions needed to revert/restore a snapshot

Parameters **name** (*str*) – Name for the snapshot, same that was set on creation

Returns None

shutdown(**args*, ***kwargs*)
Shutdown a domain

Returns None

start(**args*, ***kwargs*)
Start a domain

Returns None

state(**args*, ***kwargs*)
Return the current state of the domain

Returns Small description of the current domain state

Return type str

stop(**args*, ***kwargs*)
Stop a domain

Returns None

lago.plugins.vm.**_resolve_service_class**(*class_name*, *service_providers*)
NOTE: This must be remved once the service_class spec entry is fully deprecated

Retrieves a service plugin class from the class name instead of the provider name

Parameters

- **class_name** (*str*) – Class name of the service plugin to retrieve
- **service_providers** (*dict*) – provider_name->provider_class of the loaded service providers

Returns Class of the plugin that matches that name

Return type class

Raises *lago.plugins.NoSuchPluginError* – if there was no service plugin that matched the search

lago.plugins.vm.**check_alive**(*func*)

lago.plugins.vm.**check_defined**(*func*)

lago.providers package

Subpackages

lago.providers.libvirt package

Submodules

lago.providers.libvirt.cpu module

class lago.providers.libvirt.cpu.**CPU**(*spec*, *host_cpu*)
Bases: object

cpu_xml

generate_cpu_xml()
Get CPU XML

Returns cpu node

Return type lxml.etree.Element

generate_custom(*cpu*, *vcpu_num*, *fill_topology*)
Generate custom CPU model. This method attempts to convert the dict to XML, as defined by xmltodict.unparse method.

Parameters

- **cpu** (*dict*) – CPU spec
- **vcpu_num** (*int*) – number of virtual cpus
- **fill_topology** (*bool*) – if topology is not defined in cpu and vcpu was not set, will add CPU topology to the generated CPU.

Returns CPU XML node

Return type lxml.etree.Element

Raises LagoInitException – when failed to convert dict to XML

generate_exact(*model*, *vcpu_num*, *host_cpu*)
Generate exact CPU model with nested virtualization CPU feature.

Parameters

- **model** (*str*) – libvirt supported CPU model
- **vcpu_num** (*int*) – number of virtual cpus
- **host_cpu** (*lxml.etree.Element*) – the host CPU model

Returns CPU XML node

Return type lxml.etree.Element

generate_feature(*name*, *policy='require'*)
Generate CPU feature element

Parameters

- **name** (*str*) – feature name
- **policy** (*str*) – libvirt feature policy

Returns feature XML element

Return type lxml.etree.Element

generate_host_passthrough(*vcpu_num*)
Generate host-passthrough XML cpu node

Parameters **vcpu_num** (*int*) – number of virtual CPUs

Returns CPU XML node

Return type lxml.etree.Element

generate_topology(*vcpu_num*, *cores=1*, *threads=1*)
Generate CPU <topology> XML child

Parameters

- **vcpu_num** (*int*) – number of virtual CPUs
- **cores** (*int*) – number of cores
- **threads** (*int*) – number of threads

Returns topology XML element

Return type lxml.etree.Element

generate_vcpu(*vcpu_num*)
Generate <vcpu> domain XML child

Parameters **vcpu_num** (*int*) – number of virtual cpus

Returns vcpu XML element

Return type lxml.etree.Element

generate_vcpu_xml(*vcpu_num*)

Parameters **vcpu_num** (*int*) – number of virtual cpus

Returns vcpu XML node

Return type lxml.etree.Element

model

validate()
Validate CPU-related VM spec are compatible

Raises LagoInitException – if both 'cpu_model' and 'cpu' are defined.

vcpu_xml

vendor

class lago.providers.libvirt.cpu.**LibvirtCPU**
Bases: object

Query data from /usr/share/libvirt/cpu_map.xml

classmethod get_cpu_props(*family*, *arch='x86'*)
Get CPU info XML

Parameters

- **family** (*str*) – CPU family
- **arch** (*str*) – CPU arch

Returns CPU xml

Return type lxml.etree.Element

Raises LagoException – If no such CPU family exists

classmethod get_cpu_vendor(*family*, *arch='x86'*)
Get CPU vendor, if vendor is not available will return 'generic'

Parameters

- **family** (*str*) – CPU family
- **arch** (*str*) – CPU arch

Returns CPU vendor if found otherwise 'generic'

Return type str

classmethod get_cpus_by_arch(*arch*)
Get all CPUs info by arch

Parameters **arch** (*str*) – CPU architecture

Returns CPUs by arch XML

Return type lxml.etree.element

Raises LagoException – If no such ARCH is found

lago.providers.libvirt.network module

class lago.providers.libvirt.network.**BridgeNetwork**(*env*, *spec*, *compat*)
Bases: *lago.providers.libvirt.network.Network*

_libvirt_xml()

start()

stop()

class lago.providers.libvirt.network.**NATNetwork**(*env*, *spec*, *compat*)
Bases: *lago.providers.libvirt.network.Network*

_generate_dns_disable()

_generate_dns_forward(*forward_ip*)

_generate_main_dns(*records*, *subnet*, *forward_plain='no'*)

_ipv6_prefix(*subnet*, *const='fd8f:1391:3a82:'*)

_libvirt_xml()

class lago.providers.libvirt.network.**Network**(*env*, *spec*, *compat*)
Bases: object

_libvirt_name()

_libvirt_xml()

add_mapping(*name*, *ip*, *save=True*)

add_mappings(*mappings*)

alive()

gw()

is_management()

mapping()

name()

resolve(*name*)

save()

spec

start(*attempts=5*, *timeout=2*)
Start the network, will check if the network is active `attempts` times, waiting `timeout` between each attempt.

Parameters

- **attempts** (*int*) – number of attempts to check the network is active
- **timeout** (*int*) – timeout for each attempt

Returns None

Raises

- `RuntimeError` – if network creation failed, or failed to verify it is
- `active`.

stop()

lago.providers.libvirt.utils module

Utilities to help deal with the libvirt python bindings

`lago.providers.libvirt.utils.`**DOMAIN_STATES = {<class 'sphinx.ext.autodoc.VIR_DOMAIN_SHUTDOWN'>: 'bei**
Mapping of domain statuses values to human readable strings

class `lago.providers.libvirt.utils.`**Domain**
Bases: `object`

Class to namespace libvirt domain related helpers

static resolve_state(*state_number*)
Get a nice description from a domain state number

Parameters **state_number** (*list of int*) – State number as returned by `libvirt.virDomain.state()`

Returns

small human readable description of the domain state, unknown if the state is not in the known list

Return type str

lago.providers.libvirt.utils.**LIBVIRT_CONNECTIONS = {}**
Singleton with the cached opened libvirt connections

lago.providers.libvirt.utils.**auth_callback**(*credentials*, *user_data*)

lago.providers.libvirt.utils.**dict_to_xml**(*spec*, *full_document=False*)
Convert dict to XML

Parameters

- **spec** (*dict*) – dict to convert
- **full_document** (*bool*) – whether to add XML headers

Returns XML tree

Return type lxml.etree.Element

lago.providers.libvirt.utils.**get_domain_template**(*distro*, *libvirt_ver*, ***kwargs*)
Get a rendered Jinja2 domain template

Parameters

- **distro** (*str*) – domain distro
- **libvirt_ver** (*int*) – libvirt version
- **kwargs** (*dict*) – args for template render

Returns rendered template

Return type str

lago.providers.libvirt.utils.**get_libvirt_connection**(*name*, *libvirt_url='qemu:///system'*)

lago.providers.libvirt.utils.**get_template**(*basename*)
Load a file as a string from the templates directory

Parameters **basename** (*str*) – filename

Returns string representation of the file

Return type str

lago.providers.libvirt.vm module

class lago.providers.libvirt.vm.**LocalLibvirtVMProvider**(*vm*)
Bases: *lago.plugins.vm.VMProviderPlugin*

_create_dead_snapshot(*name*)

_create_live_snapshot(*name*)

_get_qemu_kvm_path()

_libvirt_name()

_libvirt_xml()

_load_xml()

_reclaim_disk(*path*)

_reclaim_disks()

_shutdown(*libvirt_cmd*, *ssh_cmd*, *msg*)
Choose the invoking method (using libvirt or ssh) to shutdown / poweroff the domain.

If acpi is defined in the domain use libvirt, otherwise use ssh.

Parameters

- **libvirt_cmd** (*function*) – Libvirt function the invoke
- **ssh_cmd** (*list of str*) – Shell command to invoke on the domain
- **msg** (*str*) – Name of the command that should be inserted to the log message.

Returns None

Raises RuntimeError – If acpi is not configured an ssh isn't available

bootstrap()

cpu_model
VM CPU model

Returns CPU model

Return type str

cpu_vendor
VM CPU Vendor

Returns CPU vendor

Return type str

create_snapshot(*name*)

defined()

export_disks(*standalone*, *dst_dir*, *compress*, *collect_only=False*, *with_threads=True*, **args*, ***kwargs*)
Export all the disks of self.

Parameters

- **standalone** (*bool*) – if true, merge the base images and the layered image into a new file (Supported only in qcow2 format)
- **dst_dir** (*str*) – dir to place the exported disks
- **compress** (*bool*) – if true, compress each disk.
- **collect_only** (*bool*) – If true, return only a dict which maps between the name of the vm to the paths of the disks that will be exported (don't export anything).
- **with_threads** (*bool*) – If True, export disks in parallel

Returns which maps between the name of the vm to the paths of the disks that will be exported

Return type (dict)

extract_paths(*paths*, *ignore_nopath*)
Extract the given paths from the domain

Attempt to extract all files defined in paths with the method defined in *extract_paths()*, if it fails, and *guestfs* is available it will try extracting the files with guestfs.

Parameters

- **paths** (*list of tuples*) – files to extract in *[(src1, dst1), (src2, dst2)...]* format.
- **ignore_nopath** (*boolean*) – if True will ignore none existing paths.

Returns None

Raises

- *ExtractPathNoPathError* – if a none existing path was found on the VM, and *ignore_nopath* is False.
- *ExtractPathError* – on all other failures.

extract_paths_dead(*paths*, *ignore_nopath*)

Extract the given paths from the domain using guestfs. Using guestfs can have side-effects and should be used as a second option, mainly when SSH is not available.

Parameters

- **paths** (*list of str*) – paths to extract
- **ignore_nopath** (*boolean*) – if True will ignore none existing paths.

Returns None

Raises

- *LagoException* – if `guestfs` is not importable.
- *ExtractPathNoPathError* – if a none existing path was found on the VM, and *ignore_nopath* is True.
- *ExtractPathError* – on failure extracting the files.

interactive_console(**args*, ***kwargs*)

Opens an interactive console

Returns result of the virsh command execution

Return type *lago.utils.CommandStatus*

reboot(**args*, ***kwargs*)

revert_snapshot(*name*)

shutdown(**args*, ***kwargs*)

start()

state()

Return a small description of the current status of the domain

Returns small description of the domain status, 'down' if it's not defined at all.

Return type str

stop()

Submodules

lago.brctl module

`lago.brctl.`**_brctl**(*command*, **args*)

`lago.brctl.`**_set_link**(*name*, *state*)

lago.brctl.**create**(*name*, *stp=True*)

lago.brctl.**destroy**(*name*)

lago.brctl.**exists**(*name*)

lago.build module

class lago.build.**Build**(*name*, *disk_path*, *paths*)
Bases: object

A Build object represents a build section in the init file. Each build section (which in turn belongs to a specific disk) should get his own Build object

In order to add support for a new build command, a new function with the name of the command should be implemented in this class. this function should accept a list of options and arguments and return a named tuple 'Command', where 'Command.name' is the name of the command and 'Command.cmd' is the a list containing the command and its args, for example: Command.name = 'virt-customize' Command.cmd = ['virt-customize', '-a', PATH_TO_DISK, SOME_CMDS...]

name
str – The name of the vm this builder belongs

disk_path
str – The path to the disk that needs to be customized

paths
lago.paths.Paths – The paths of the current prefix

build_cmds
list of str – A list of commands that should be invoked on the disk located in disk_path

build()
Run all the commands in self.build_cmds

Raises *lago.build.BuildException* – If a command returned a non-zero code

get_cmd_handler(*cmd*)
Return an handler for cmd. The handler and the command should have the same name. See class description for more info about handlers.

Parameters **cmd** (*str*) – The name of the command

Returns which handles cmd

Return type callable

Raises *lago.build.BuildException* – If an handler for cmd doesn't exist

classmethod get_instance_from_build_spec(*name*, *disk_path*, *build_spec*, *paths*)

Parameters

- **name** (*str*) – The name of the vm this builder belongs
- **disk_path** (*str*) – The path to the disk that needs to be customized
- **paths** (lago.paths.Paths) – The paths of the current prefix
- **build_spec** (*dict*) – The build spec part, associated with the disk located at disk_path, from the init file.

Returns

An instance of Build with a normalized build spec i.e ready to be invoked.

normalize_build_spec(*build_spec*)
Convert a build spec into a list of Command tuples. After running this command, self.build_cmds should hold all the commands that should be run on the disk in self.disk_path.

Parameters **build_spec** (*dict*) – The buildspec part from the init file

static normalize_options(*options*)
Turns a mapping of 'option: arg' to a list and prefix the options. arg can be a list of arguments.

for example:

dict = { o1: a1, o2: , o3: [a31, a32] o4: []

}

will be transformed to:

[prefix_option(o1), a1, prefix_option(o2), prefix_option(o3), a31, prefix_option(o3), a32 prefix_option(o4)

]

note that empty arguments are omitted

Parameters **options** (*dict*) – A mapping between options and arguments

Returns A normalized version of 'options' as mentioned above

Return type lst

static prefix_option(*option*)
Depends on the option's length, prefix it with '-' or '–' :param option: The option to prefix :type option: str

Returns prefixed option

Return type str

virt_customize(*options*)
Handler for 'virt-customize' note: if 'ssh-inject' option was specified without a path to a key, the prefix' key will be copied to the vm.

Parameters **options** (*lst of str*) – Options and arguments for 'virt-customize'

Returns which handles cmd

Return type callable

Raises *lago.build.BuildException* – If an handler for cmd doesn't exist

exception lago.build.**BuildException**
Bases: *lago.utils.LagoException*

class lago.build.**Command**(*name*, *cmd*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values

_fields = ('name', 'cmd')

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x906d60>*, *len=<built-in function len>*)
Make a new Command object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new Command object replacing specified fields with new values

cmd
Alias for field number 1

name
Alias for field number 0

lago.cmd module

lago.config module

class lago.config.**ConfigLoad**(*root_section='lago'*, *defaults={}*)
Bases: object

Merges configuration parameters from 3 different sources: 1. Enviornment vairables 2. config files in .INI format 3. argparse.ArgumentParser

The assumed order(but not necessary) order of calls is: load() - load from config files and environment variables update_parser(parser) - update from the declared argparse parser update_args(args) - update from passed arguments to the parser

__getitem__(*key*)
Get a variable from the default section, good for fail-fast if key does not exists.

Parameters **key** (*str*) – key

Returns config variable

Return type str

get(**args*)
Get a variable from the default section :param *args: dict.get() args :type *args: args

Returns config variable

Return type str

get_ini(*incl_unset=False*)
Return the config dictionary in INI format :param incl_unset: include variables with no defaults. :type incl_unset: bool

Returns string of the config file in INI format

Return type str

get_section(**args*)
get a section dictionary Args:

Returns section config dictionary

Return type dict

load()
Load all configurations from available resources, skip if empty:

1.default` dict passed to ConfigLoad.__init__().

2. **Custom paths as defined in `CONFS_PATH` in** *constants*.

3. XDG standard paths.

4. Environment variables.

Returns dict of dicts.

Return type dict

`update_args`(*args*)
Update config dictionary with parsed args, as resolved by argparse. Only root positional arguments that already exist will overridden.

Parameters **`args`** (*namespace*) – args parsed by argparse

`update_parser`(*parser*)
Update config dictionary with declared arguments in an argparse.parser New variables will be created, and existing ones overridden.

Parameters **`parser`** (*argparse.ArgumentParser*) – parser to read variables from

lago.config.**`_get_configs_path`**()
Get a list of possible configuration files, from the following sources: 1. All files that exists in constants.CONFS_PATH. 2. All XDG standard config files for "lago.conf", in reversed order of importance.

Returns list of files

Return type list(str)

lago.config.**`get_env_dict`**(*root_section*)
Read all Lago variables from the environment. The lookup format is: LAGO_VARNAME - will land into 'lago' section LAGO__SECTION1__VARNAME - will land into 'section1' section, notice the double '__'. LAGO__LONG_SECTION_NAME__VARNAME - will land into 'long_section_name'

Returns dict of section configuration dicts

Return type dict

Examples

```
>>> os.environ['LAGO_GLOBAL_VAR'] = 'global'
>>> os.environ['LAGO__INIT__REPO_PATH'] = '/tmp/store'
>>>
>>> config.get_env_dict()
{'init': {'repo_path': '/tmp/store'}, 'lago': {'global_var': 'global'}}
```

lago.constants module

lago.constants.**`CONFS_PATH`** = **['/etc/lago/lago.conf']**
CONFS_PATH - default path to first look for configuration files.

lago.constants.**`LIBEXEC_DIR`** = **'/usr/libexec/lago/'**
LIBEXEC_DIR -

lago.export module

class lago.export.**DiskExportManager**(*dst*, *disk_type*, *disk*, *do_compress*)
Bases: object

DiskExportManager object is responsible on the export process of an image from the current Lago prefix.

DiskExportManger is the base class of specific DiskExportManger. Each specific DiskExportManger is responsible on the export process of an image with a specific type (e.g template, file...)

src
str – Path to the image that should be exported

name
str – The name of the exported disk

dst
str – The absolute path of the exported disk

disk_type
str – The type of the image e.g template, file, empty...

disk
dict – Disk attributes (of the disk that should be exported) as found in workdir/current/virt/VM-NAME

exported_metadata
dict – A copy of the source disk metadata, this dict should be updated with new values during the export process.

do_compress
bool – If true, apply compression to the exported disk.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

calc_sha(*checksum*)
Calculate the checksum of the new exported disk, write it to a file, and update this managers 'exported_metadata'.

Parameters **checksum** (*str*) – The type of the checksum

compress()
Compress the new exported image, Block size was taken from virt-builder page

copy()
Copy the disk using cp in order to preserves the 'sparse' structure of the file

export()
This method will handle the export process and should implemented in each subclass.

static get_instance_by_type(*dst*, *disk*, *do_compress*, **args*, ***kwargs*)

Parameters

- **dst** (*str*) – The path of the new exported disk. can contain env variables.
- **disk** (*dict*) – Disk attributes (of the disk that should be exported) as found in workdir/current/virt/VM-NAME
- **do_compress** (*bool*) – If true, apply compression to the exported disk.

Returns An instance of a subclass of DiskExportManager which matches the disk type.

sparse()
Make the exported images more compact by removing unused space. Please refer to 'virt-sparsify' for more info.

update_lago_metadata()

write_lago_metadata()

class lago.export.**FileExportManager**(*dst*, *disk_type*, *disk*, *do_compress*, **args*, ***kwargs*)
Bases: *lago.export.DiskExportManager*

FileExportManager is responsible exporting images of type file and empty.

standalone
bool – If true, create a new image which is the result of merging all the layers of src (the image that we want to export).

src_qemu_info
dict – Metadata on src which was generated by qemu-img.

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

export()
See DiskExportManager.export

class lago.export.**TemplateExportManager**(*dst*, *disk_type*, *disk*, *do_compress*, **args*, ***kwargs*)
Bases: *lago.export.DiskExportManager*

TemplateExportManager is responsible exporting images of type template.

See superclass

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

export()
See DiskExportManager.export

rebase()
Change the backing-file entry of the exported disk. Please refer to 'qemu-img rebase' manual for more info.

update_lago_metadata()

class lago.export.**VMExportManager**(*disks*, *dst*, *compress*, *with_threads=True*, **args*, ***kwargs*)
Bases: object

VMExportManager object is responsible on the export process of a list of disks.

disks
list of dicts – Disks to export.

dst
str – Where to place the exported disks.

compress
bool – If True compress each exported disk.

with_threads
bool – If True, run the export in parallel

***args**
list – Extra args, will be passed to each DiskExportManager

****kwargs**
dict – Extra args, will be passed to each DiskExportManager

_collect()

Returns The disks that needed to be exported

Return type (generator of dicts)

_get_export_mgr()

Returns Handler for each disk

Return type (*DiskExportManager*)

collect_paths()

Returns The path of the disks that will be exported.

Return type (list of str)

export()
Run the export process :returns: The path of the exported disks. :rtype: (list of str)

exported_disks_paths()

Returns The path of the exported disks.

Return type (list of str)

lago.guestfs_tools module

lago.guestfs_tools.**_copy_path**(*guestfs_conn*, *guest_path*, *host_path*)

lago.guestfs_tools.**extract_paths**(*disk_path*, *disk_root*, *paths*, *ignore_nopath*)
Extract paths from a disk using guestfs

Parameters

- **disk_path** (*str*) – path to the disk
- **disk_root** (*str*) – root partition
- **paths** (*list of tuples*) – files to extract in *[(src1, dst1), (src2, dst2)...]* format.
- **ignore_nopath** (*bool*) – If set to True, ignore source paths that do
- **exist** (*not*) –

Returns None

Raises

- *ExtractPathNoPathError* – if a none existing path was found on the VM, and *ignore_nopath* is False.

- *ExtractPathError* – on all other failures.

lago.lago_ansible module

class lago.lago_ansible.**LagoAnsible**(*prefix*)
Bases: object

A class for handling Ansible related tasks

prefix
lago.prefix.Prefix – The prefix that this object wraps

_generate_entry(*vm*)
Generate host entry for the given VM :param vm: The VM for which the entry

should be created for.

Returns An entry for vm

Return type str

get_inventory(*keys=None*)
Create an Ansible inventory based on python dicts and lists. The returned value is a dict in which every key represents a group and every value is a list of entries for that group.

Parameters **keys** (*list of str*) – Path to the keys that will be used to create groups.

Returns dict based Ansible inventory

Return type dict

get_inventory_str(*keys=None*)
Convert a dict generated by ansible.LagoAnsible.get_inventory to an INI-like file.

Parameters **keys** (*list of str*) – Path to the keys that will be used to create groups.

Returns INI-like Ansible inventory

Return type str

get_inventory_temp_file(**args*, ***kwds*)
Context manager which returns the inventory written on a tempfile. The tempfile will be deleted as soon as this context manger ends.

Parameters **keys** (*list of str*) – Path to the keys that will be used to create groups.

Yields *tempfile.NamedTemporaryFile* – Temp file containing the inventory

static get_key(*key*, *data_structure*)
Helper method for extracting values from a nested data structure.

Parameters

- **key** (*str*) – The path to the vales (a series of keys and indexes separated by '/')
- **data_structure** (*dict or list*) – The data structure from which the value will be extracted.

Returns The values associated with key

Return type str

lago.log_utils module

This module defines the special logging tools that lago uses

lago.log_utils.**ALWAYS_SHOW_REG = <_sre.SRE_Pattern object>**
Regexp that will match the above template

lago.log_utils.**ALWAYS_SHOW_TRIGGER_MSG = 'force-show:%s'**
Message template that will always shoud the messago

class lago.log_utils.**ColorFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

Formatter to add colors to log records

CRITICAL = '\x1b[31m'

CYAN = '\x1b[36m'

DEBUG = ''

DEFAULT = '\x1b[0m'

ERROR = '\x1b[31m'

GREEN = '\x1b[32m'

INFO = '\x1b[36m'

NONE = ''

RED = '\x1b[31m'

WARNING = '\x1b[33m'

WHITE = '\x1b[37m'

YELLOW = '\x1b[33m'

classmethod colored(*color*, *message*)
Small function to wrap a string around a color

Parameters

- **color** (*str*) – name of the color to wrap the string with, must be one of the class properties
- **message** (*str*) – String to wrap with the color

Returns the colored string

Return type str

format(*record*)
Adds colors to a log record and formats it with the default

Parameters **record** (*logging.LogRecord*) – log record to format

Returns The colored and formatted record string

Return type str

class lago.log_utils.**ContextLock**
Bases: object

Context manager to thread lock a block of code

lago.log_utils.**END_TASK_MSG = 'Success'**
Message to be shown when a task is ended

lago.log_utils.**END_TASK_REG = <_sre.SRE_Pattern object>**
Regexp that will match the above template

lago.log_utils.**END_TASK_TRIGGER_MSG = 'end task%s'**
Message template that will trigger a task end

class lago.log_utils.**LogTask**(*task*, *logger=<module 'logging' from '/usr/lib/python2.7/logging/__init__.pyc'>*, *level='info'*, *propagate_fail=True*, *uuid=None*)
Bases: object

Context manager for a log task

Example

```
>>> with LogTask('mytask'):
...     pass
```

lago.log_utils.**START_TASK_MSG = ''**
Message to be shown when a task is started

lago.log_utils.**START_TASK_REG = <_sre.SRE_Pattern object>**
Regexp that will match the above template

lago.log_utils.**START_TASK_TRIGGER_MSG = 'start task%s'**
Message template that will trigger a task

class lago.log_utils.**Task**(*name*, **args*, ***kwargs*)
Bases: collections.deque

Small wrapper around deque to add the failed status and name to a task

name
str – name for this task

failed
bool – If this task has failed or not (if there was any error log shown during it's execution)

force_show
bool – If set, will show any log records generated inside this task even if it's out of nested depth limit

elapsed_time()

class lago.log_utils.**TaskHandler**(*initial_depth=0*, *task_tree_depth=-1*, *buffer_size=2000*, *dump_level=40*, *level=0*, *formatter=<class 'lago.log_utils.ColorFormatter'>*)
Bases: logging.StreamHandler

This log handler will use the concept of tasks, to hide logs, and will show all the logs for the current task if there's a logged error while running that task.

It will hide any logs that belong to nested tasks that have more than `task_tree_depth` parent levels, and for the ones that are above that level, it will show only the logs that have a loglevel above `level`.

You can force showing a log record immediately if you use the *log_always()* function bypassing all the filters.

If there's a log record with log level higher than `dump_level` it will be considered a failure, and all the logs for the current task that have a log level above `level` will be shown no matter at which depth the task belongs to. Also, all the parent tasks will be tagged as error.

formatter
logging.LogFormatter – formatter to use

initial_depth
int – Initial depth to account for, in case this handler was created in a subtask

tasks_by_thread (dict of str
OrderedDict of str: Task): List of thread names, and their currently open tasks with their latest log records

dump_level
int – log level from which to consider a log record as error

buffer_size
int – Size of the log record deque for each task, the bigger, the more records it can show in case of error but the more memory it will use

task_tree_depth
int – number of the nested level to show start/end task logs for, if -1 will show always

level
int – Log level to show logs from if the depth limit is not reached

main_failed
bool – used to flag from a child thread that the main should fail any current task

_tasks_lock
ContextLock – Lock for the tasks_by_thread dict

_main_thread_lock
ContextLock – Lock for the main_failed bool

TASK_INDICATORS = ['@', '#', '*', '-', '~']
List of chars to show as task prefix, to ease distinguishing them

am_i_main_thread
**Returns* – bool* – if the current thread is the main thread

close_children_tasks(*parent_task_name*)
Closes all the children tasks that were open

Parameters **parent_task_name** (*str*) – Name of the parent task

Returns None

cur_depth_level
**Returns* – int* – depth level for the current task

cur_task
**Returns* – str* – the current active task

cur_thread
**Returns* – str* – Name of the current thread

emit(*record*)
Handle the given record, this is the entry point from the python logging facility

Params: record (logging.LogRecord): log record to handle

Returns None

get_task_indicator(*task_level=None*)

Parameters **task_level** (*int or None*) – task depth level to get the indicator for, if None, will use the current tasks depth

Returns char to prepend to the task logs to indicate it's level

Return type str

get_tasks(*thread_name*)

Parameters **thread_name** (*str*) – name of the thread to get the tasks for

Returns

list of task names and log records for each for the given thread

Return type OrderedDict of str, *Task*

handle_closed_task(*task_name*, *record*)
Do everything needed when a task is closed

Params: task_name (str): name of the task that is finishing record (logging.LogRecord): log record with all the info

Returns None

handle_error()
Handles an error log record that should be shown

Returns None

handle_new_task(*task_name*, *record*)
Do everything needed when a task is starting

Params: task_name (str): name of the task that is starting record (logging.LogRecord): log record with all the info

Returns None

mark_main_tasks_as_failed()
Flags to the main thread that all it's tasks sholud fail

Returns None

mark_parent_tasks_as_failed(*task_name*, *flush_logs=False*)
Marks all the parent tasks as failed

Parameters

- **task_name** (*str*) – Name of the child task
- **flush_logs** (*bool*) – If `True` will discard all the logs form parent tasks

Returns None

pretty_emit(*record*, *is_header=False*, *task_level=None*)
Wrapper around the `logging.StreamHandler` emit method to add some decoration stuff to the message

Parameters

- **record** (*logging.LogRecord*) – log record to emit
- **is_header** (*bool*) – if this record is a header, usually, a start or end task message

- **task_level** (*int*) – If passed, will take that as the current nested task level instead of calculating it from the current tasks

Returns None

should_show_by_depth(*cur_level=None*)

Parameters **cur_level** (*int*) – depth level to take into account

Returns

True if the given depth level should show messages (not taking into account the log level)

Return type bool

should_show_by_level(*record_level*, *base_level=None*)

Parameters

- **record_level** (*int*) – log level of the record to check
- **base_level** (*int or None*) – log level to check against, will use the object's *dump_level* if None is passed

Returns

True if the given log record should be shown according to the log level

Return type bool

tasks

Returns –

OrderedDict of str, Task: list of task names and log records for each for the current thread

lago.log_utils.**end_log_task**(*task*, *logger=<module 'logging' from '/usr/lib/python2.7/logging/__init__.pyc'>*, *level='info'*)

Ends a log task

Parameters

- **task** (*str*) – name of the log task to end
- **logger** (*logging.Logger*) – logger to use
- **level** (*str*) – log level to use

Returns None

lago.log_utils.**get_default_log_formatter**()

lago.log_utils.**hide_paramiko_logs**()

lago.log_utils.**hide_stevedore_logs**()

Hides the logs of stevedore, this function was added in order to support older versions of stevedore

We are using the NullHandler in order to get rid from 'No handlers could be found for logger...' msg

Returns None

lago.log_utils.**log_always**(*message*)

Wraps the given message with a tag that will make it be always logged by the task logger

Parameters **message** (*str*) – message to wrap with the tag

Returns

tagged message that will get it shown immediately by the task logger

Return type str

lago.log_utils.**log_task**(*task*, *logger=<module 'logging' from '/usr/lib/python2.7/logging/__init__.pyc'>*, *level='info'*, *propagate_fail=True*, *uuid=None*)
Parameterized decorator to wrap a function in a log task

Example

```
>>> @log_task('mytask')
... def do_something():
...     pass
```

lago.log_utils.**setup_prefix_logging**(*logdir*)
Sets up a file logger that will create a log in the given logdir (usually a lago prefix)

Parameters **logdir** (*str*) – path to create the log into, will be created if it does not exist

Returns None

lago.log_utils.**start_log_task**(*task*, *logger=<module 'logging' from '/usr/lib/python2.7/logging/__init__.pyc'>*, *level='info'*)
Starts a log task

Parameters

- **task** (*str*) – name of the log task to start
- **logger** (*logging.Logger*) – logger to use
- **level** (*str*) – log level to use

Returns None

lago.paths module

class lago.paths.**Paths**(*prefix*)
Bases: object

images(**path*)

logs()

metadata()

prefix_lagofile()
This file represents a prefix that's initialized

prefixed(**args*)

scripts(**args*)

ssh_id_rsa()

ssh_id_rsa_pub()

uuid()

virt(**path*)

lago.prefix module

lago.sdk module

lago.sdk_utils module

class lago.sdk_utils.**SDKMethod**(*name*)
Bases: object

Metadata to store inside the decorated function

class lago.sdk_utils.**SDKWrapper**
Bases: sphinx.ext.autodoc.ObjectProxy

A proxy object that exposes only methods which were decorated with *expose()* decorator.

lago.sdk_utils.**expose**(*func*)
Decorator to be used with *SDKWrapper*. This decorator indicates that the wrapped method or class should be exposed in the proxied object.

Parameters **func** (*types.FunctionType/types.MethodType*) – function to decorate

Returns None

lago.sdk_utils.**getattr_sdk**(*attr*, *name*)
Filter SDK attributes

Parameters

- **attr** (*attribute*) – Attribute as returned by getattr().
- **name** (*str*) – Attribute name.

Returns *attr* if passed.

lago.sdk_utils.**setup_sdk_logging**(*logfile=None*, *loglevel=20*)
Setup a NullHandler to the root logger. If logfile is passed, additionally add a FileHandler in loglevel level.

Parameters

- **logfile** (*str*) – A path to setup a log file.
- **loglevel** (*int*) – logging log level.

Returns None

lago.service module

class lago.service.**SysVInitService**(*vm*, *name*)
Bases: *lago.plugins.service.ServicePlugin*

BIN_PATH = '/sbin/service'

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

_request_start()

_request_stop()

state()

class lago.service.**SystemdContainerService**(*vm*, *name*)
Bases: *lago.plugins.service.ServicePlugin*

BIN_PATH = '/usr/bin/docker'

HOST_BIN_PATH = '/usr/bin/systemctl'

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

_request_start()

_request_stop()

state()

class lago.service.**SystemdService**(*vm*, *name*)
Bases: *lago.plugins.service.ServicePlugin*

BIN_PATH = '/usr/bin/systemctl'

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

_request_start()

_request_stop()

state()

lago.ssh module

lago.ssh.**_gen_ssh_command_id**()

lago.ssh.**drain_ssh_channel**(*chan*, *stdin=None*, *stdout=<open file '<stdout>', mode 'w'>*, *stderr=<open file '<stderr>', mode 'w'>*)

lago.ssh.**get_ssh_client**(*ip_addr*, *ssh_key=None*, *host_name=None*, *ssh_tries=None*, *propagate_fail=True*, *username='root'*, *password='123456'*)

lago.ssh.**interactive_ssh**(*ip_addr*, *command=None*, *host_name=None*, *ssh_key=None*, *username='root'*, *password='123456'*)

lago.ssh.**interactive_ssh_channel**(*chan*, *command=None*, *stdin=<open file '<stdin>', mode 'r'>*)

lago.ssh.**ssh**(*ip_addr*, *command*, *host_name=None*, *data=None*, *show_output=True*, *propagate_fail=True*, *tries=None*, *ssh_key=None*, *username='root'*, *password='123456'*)

lago.ssh.**ssh_script**(*ip_addr*, *path*, *host_name=None*, *show_output=True*, *ssh_key=None*, *username='root'*, *password='123456'*)

lago.ssh.**wait_for_ssh**(*ip_addr*, *host_name=None*, *connect_timeout=600*, *ssh_key=None*, *username='root'*, *password='123456'*)

lago.subnet_lease module

lago.sysprep module

lago.sysprep.**_guestfs_version**(*default={'major': 1L, 'minor': 20L}*)

lago.sysprep.**_render_template**(*distro*, *loader*, ***kwargs*)

lago.sysprep.**sysprep**(*disk*, *distro*, *loader=None*, *backend='direct'*, ***kwargs*)

Run virt-sysprep on the `disk`, commands are built from the distro specific template and arguments passed in `kwargs`. If no template is available it will default to `sysprep-base.j2`.

Parameters

- **disk** (*str*) – path to disk
- **distro** (*str*) – distro to render template for
- **loader** (*jinja2.BaseLoader*) – Jinja2 template loader, if not passed, will search Lago's package.
- **backend** (*str*) – libguestfs backend to use
- ****kwargs** (*dict*) – environment variables for Jinja2 template

Returns None

Raises `RuntimeError` – On virt-sysprep none 0 exit code.

lago.templates module

This module contains any disk template related classes and functions, including the repository store manager classes and template providers, some useful definitions:

- **Template repositories:** Repository where to fetch templates from, as an http server
- **Template store:** Local store to cache templates
- **Template:** Unititialized disk image to use as base for other disk images
- **Template version:** Specific version of a template, to allow getting updates without having to change the template name everywhere

class lago.templates.**FileSystemTemplateProvider**(*root*)

Handles file type templates, that is, getting a disk template from the host's filesystem

_prefixed(**path*)

Join all the given paths prefixed with this provider's base root path

Parameters ***path** (*str*) – sections of the path to join, passed as positional arguments

Returns Joined paths prepended with the provider root path

Return type str

download_image(*handle*, *dest*)

Copies over the handl to the destination

Parameters

- **handle** (*str*) – path to copy over
- **dest** (*str*) – path to copy to

Returns None

get_hash (*handle*)
Returns the associated hash for the given handle, the hash file must exist (`handle + '.hash'`).

Parameters **handle** (*str*) – Path to the template to get the hash from

Returns Hash for the given handle

Return type str

get_metadata (*handle*)
Returns the associated metadata info for the given handle, the metadata file must exist (`handle + '.metadata'`).

Parameters **handle** (*str*) – Path to the template to get the metadata from

Returns Metadata for the given handle

Return type dict

class lago.templates.**HttpTemplateProvider** (*baseurl*)
This provider allows the usage of http urls for templates

download_image (*handle*, *dest*)
Downloads the image from the http server

Parameters

- **handle** (*str*) – url from the *self.baseurl* to the remote template
- **dest** (*str*) – Path to store the downloaded url to, must be a file path

Returns None

static extract_image_xz (*path*)

get_hash (*handle*)
Get the associated hash for the given handle, the hash file must exist (`handle + '.hash'`).

Parameters **handle** (*str*) – Path to the template to get the hash from

Returns Hash for the given handle

Return type str

get_metadata (*handle*)
Returns the associated metadata info for the given handle, the metadata file must exist (`handle + '.metadata'`). If the given handle has an `.xz` extension, it will get removed when calculating the handle metadata path

Parameters **handle** (*str*) – Path to the template to get the metadata from

Returns Metadata for the given handle

Return type dict

open_url (*url*, *suffix=''*, *dest=None*)
Opens the given url, trying the compressed version first. The compressed version url is generated adding the `.xz` extension to the `url` and adding the given suffix **after** that `.xz` extension. If dest passed, it will download the data to that path if able

Parameters

- **url** (*str*) – relative url from the `self.baseurl` to retrieve
- **suffix** (*str*) – optional suffix to append to the url after adding the compressed extension to the path

- **dest** (*str or None*) – Path to save the data to

Returns

response object to read from (lazy read), closed if no dest passed

Return type urllib.addinfourl

Raises RuntimeError – if the url gave http error when retrieving it

class lago.templates.**Template**(*name*, *versions*)
Disk image template class

name
str – Name of this template

_versions (dict(str
TemplateVersion)): versions for this template

get_latest_version()
Retrieves the latest version for this template, the latest being the one with the newest timestamp

Returns TemplateVersion

get_version(*ver_name=None*)
Get the given version for this template, or the latest

Parameters **ver_name** (*str or None*) – Version to retieve, None for the latest

Returns

The version matching the given name or the latest one

Return type *TemplateVersion*

class lago.templates.**TemplateRepository**(*dom*)
A template repository is a single source for templates, that uses different providers to actually retrieve them. That means for example that the 'ovirt' template repository, could support the 'http' and a theoretical 'gluster' template providers.

_dom
dict – Specification of the template

_providers
dict – Providers instances for any source in the spec

_get_provider(*spec*)
Get the provider for the given template spec

Parameters **spec** (*dict*) – Template spec

Returns A provider instance for that spec

Return type *HttpTemplateProvider* or *FileSystemTemplateProvider*

classmethod from_url(*path*)
Instantiate a *TemplateRepository* instance from the data in a file or url

Parameters **path** (*str*) – Path or url to the json file to load

Returns A new instance

Return type *TemplateRepository*

get_by_name(*name*)
Retrieve a template by it's name

Parameters **name** (*str*) – Name of the template to retrieve

Raises KeyError – if no template is found

name
Getter for the template repo name

Returns the name of this template repo

Return type str

class lago.templates.**TemplateStore**(*path*)
Local cache to store templates

The store uses various files to keep track of the templates cached, access and versions. An example template store looks like:

```
$ tree /var/lib/lago/store/
/var/lib/lago/store/
- in_office_repo:centos6_engine:v2.tmp
- in_office_repo:centos7_engine:v5.tmp
- in_office_repo:fedora22_host:v2.tmp
- phx_repo:centos6_engine:v2
- phx_repo:centos6_engine:v2.hash
- phx_repo:centos6_engine:v2.metadata
- phx_repo:centos6_engine:v2.users
- phx_repo:centos7_engine:v4.tmp
- phx_repo:centos7_host:v4.tmp
- phx_repo:storage-nfs:v1.tmp
```

There you can see the files:

- ***.tmp** Temporary file created while downloading the template from the repository (depends on the provider)
- **${repo_name}:${template_name}:${template_version}** This file is the actual disk image template
- ***.hash** Cached hash for the template disk image
- ***.metadata** Metadata for this template image in json format, usually this includes the *distro* and *root-password*

__contains__(*temp_ver*)
Checks if a given version is in this store

Parameters **temp_ver** (TemplateVersion) – Version to look for

Returns True if the version is in this store

Return type bool

_prefixed(**path*)
Join the given paths and prepend this stores path

Parameters ***path** (*str*) – list of paths to join, as positional arguments

Returns all the paths joined and prepended with the store path

Return type str

download(*temp_ver*, *store_metadata=True*)
Retrieve the given template version

Parameters

- **temp_ver** (TemplateVersion) – template version to retrieve

- **store_metadata** (*bool*) – If set to `False`, will not refresh the local metadata with the retrieved one

Returns None

get_path(*temp_ver*)
Get the path of the given version in this store

Parameters **TemplateVersion** (*temp_ver*) – version to look for

Returns The path to the template version inside the store

Return type str

Raises `RuntimeError` – if the template is not in the store

get_stored_hash(*temp_ver*)
Retrieves the hash for the given template version from the store

Parameters **temp_ver** (`TemplateVersion`) – template version to retrieve the hash for

Returns hash of the given template version

Return type str

get_stored_metadata(*temp_ver*)
Retrieves the metadata for the given template version from the store

Parameters **temp_ver** (`TemplateVersion`) – template version to retrieve the metadata for

Returns the metadata of the given template version

Return type dict

class `lago.templates.`**TemplateVersion**(*name*, *source*, *handle*, *timestamp*)
Each template can have multiple versions, each of those is actually a different disk template file representation, under the same base name.

download(*destination*)
Retrieves this template to the destination file

Parameters **destination** (*str*) – file path to write this template to

Returns None

get_hash()
Returns the associated hash for this template version

Returns Hash for this version

Return type str

get_metadata()
Returns the associated metadata info for this template version

Returns Metadata for this version

Return type dict

timestamp()
Getter for the timestamp

`lago.templates.`**_locked**(*func*)
Decorator that ensures that the decorated function has the lock of the repo while running, meant to decorate only bound functions for classes that have *lock_path* method.

lago.templates.**find_repo_by_name**(*name*, *repo_dir=None*)
Searches the given repo name inside the repo_dir (will use the config value 'template_repos' if no repo dir passed), will rise an exception if not found

Parameters

- **name** (*str*) – Name of the repo to search
- **repo_dir** (*str*) – Directory where to search the repo

Returns path to the repo

Return type str

Raises RuntimeError – if not found

lago.utils module

class lago.utils.**CommandStatus**
Bases: *lago.utils.CommandStatus*

class lago.utils.**EggTimer**(*timeout*)

elapsed()

class lago.utils.**ExceptionTimer**(*timeout*)
Bases: object

start()

stop()

exception lago.utils.**LagoException**
Bases: exceptions.Exception

exception lago.utils.**LagoInitException**
Bases: *lago.utils.LagoException*

exception lago.utils.**LagoUserException**
Bases: *lago.utils.LagoException*

class lago.utils.**LockFile**(*path*, *timeout=None*, ***kwargs*)
Bases: object

Context manager that creates a lock around a directory, with optional timeout in the acquire operation

Parameters

- **path** (*str*) – path to the dir to lock
- **timeout** (*int*) – timeout in seconds to wait while acquiring the lock
- ****kwargs** (*dict*) – Any other param to pass to *lockfile.LockFile*

__enter__()
Start the lock with timeout if needed in the acquire operation

Raises *TimerException* – if the timeout is reached before acquiring the lock

class lago.utils.**RollbackContext**(**args*)
Bases: object

A context manager for recording and playing rollback. The first exception will be remembered and re-raised after rollback

Sample usage: > with RollbackContext() as rollback: > step1() > rollback.prependDefer(lambda: undo step1) > def undoStep2(arg): pass > step2() > rollback.prependDefer(undoStep2, arg)

More examples see tests/utilsTests.py @ vdsm code

__exit__(*exc_type*, *exc_value*, *traceback*)
If this function doesn't return True (or raises a different exception), python re-raises the original exception once this function is finished.

clear()

defer(*func*, **args*, ***kwargs*)

prependDefer(*func*, **args*, ***kwargs*)

exception lago.utils.**TimerException**
Bases: exceptions.Exception

Exception to throw when a timeout is reached

class lago.utils.**VectorThread**(*targets*)

join_all(*raise_exceptions=True*)

start_all()

lago.utils.**_CommandStatus**
alias of *CommandStatus*

lago.utils.**_add_subparser_to_cp**(*cp*, *section*, *actions*, *incl_unset*)

lago.utils.**_ret_via_queue**(*func*, *queue*)

lago.utils.**_run_command**(*command*, *input_data=None*, *stdin=None*, *out_pipe=-1*, *err_pipe=-1*, *env=None*, *uuid=None*, ***kwargs*)
Runs a command

Parameters

- **command** (*list of str*) – args of the command to execute, including the command itself as command[0] as *['ls', '-l']*
- **input_data** (*str*) – If passed, will feed that data to the subprocess through stdin
- **out_pipe** (*int or file*) – File descriptor as passed to :ref:subprocess.Popen to use as stdout
- **stdin** (*int or file*) – File descriptor as passed to :ref:subprocess.Popen to use as stdin
- **err_pipe** (*int or file*) – File descriptor as passed to :ref:subprocess.Popen to use as stderr
- **of str** (*env (dict*) – str): If set, will use the given dict as env for the subprocess
- **uuid** (uuid) – If set the command will be logged with the given uuid converted to string, otherwise, a uuid v4 will be generated.
- ****kwargs** – Any other keyword args passed will be passed to the :ref:subprocess.Popen call

Returns result of the interactive execution

Return type *lago.utils.CommandStatus*

lago.utils.**add_timestamp_suffix**(*base_string*)

lago.utils.**argparse_to_ini**(*parser*, *root_section='lago'*, *incl_unset=False*)

lago.utils.**compress**(*input_file*, *block_size*, *fail_on_error=True*)

lago.utils.**cp**(*input_file*, *output_file*, *fail_on_error=True*)

lago.utils.**deepcopy**(*original_obj*)
Creates a deep copy of an object with no crossed referenced lists or dicts, useful when loading from yaml as anchors generate those cross-referenced dicts and lists

Parameters **original_obj** (*object*) – Object to deep copy

Returns deep copy of the object

Return type object

lago.utils.**filter_spec**(*spec*, *paths*, *wildcard='*'*, *separator='/'*)
Remove keys from a spec file. For example, with the following path: domains//*disks*//metadata all the metadata dicts from all domains disks will be removed.

Parameters

- **spec** (*dict*) – spec to remove keys from
- **paths** (*list*) – list of paths to the keys that should be removed
- **wildcard** (*str*) – wildcard character
- **separator** (*str*) – path separator

Returns None

Raises utils.LagoUserException – If a malformed path was detected

lago.utils.**func_vector**(*target*, *args_sequence*)

lago.utils.**get_hash**(*file_path*, *checksum='sha1'*)
Generate a hash for the given file

Parameters

- **file_path** (*str*) – Path to the file to generate the hash for
- **checksum** (*str*) – hash to apply, one of the supported by hashlib, for example sha1 or sha512

Returns hash for that file

Return type str

lago.utils.**get_qemu_info**(*path*, *backing_chain=False*, *fail_on_error=True*)
Get info on a given qemu disk

Parameters

- **path** (*str*) – Path to the required disk
- **backing_chain** (*boo*) – if true, include also info about
- **image predecessors.** (*the*) –

Returns if backing_chain == True then a list of dicts else a dict

Return type object

lago.utils.**in_prefix**(*prefix_class*, *workdir_class*)

lago.utils.**invoke_different_funcs_in_parallel**(**funcs*)

lago.utils.**invoke_in_parallel**(*func*, **args_sequences*)

lago.utils.**ipv4_to_mac**(*ip*)

lago.utils.**json_dump**(*obj*, *f*)

lago.utils.**load_virt_stream**(*virt_fd*)
Loads the given conf stream into a dict, trying different formats if needed

Parameters **virt_fd** (*str*) – file like objcect with the virt config to load

Returns Loaded virt config

Return type dict

lago.utils.**qemu_rebase**(*target*, *backing_file*, *safe=True*, *fail_on_error=True*)
changes the backing file of 'source' to 'backing_file' If backing_file is specified as "" (the empty string), then the image is rebased onto no backing file (i.e. it will exist independently of any backing file). (Taken from qemu-img man page)

Parameters

- **target** (*str*) – Path to the source disk
- **backing_file** (*str*) – path to the base disk
- **safe** (*bool*) – if false, allow unsafe rebase (check qemu-img docs for more info)

lago.utils.**read_nonblocking**(*file_descriptor*)

lago.utils.**rotate_dir**(*base_dir*)

lago.utils.**run_command**(*command*, *input_data=None*, *out_pipe=-1*, *err_pipe=-1*, *env=None*, ***kwargs*)
Runs a command non-interactively

Parameters

- **command** (*list of str*) – args of the command to execute, including the command itself as command[0] as *['ls', '-l']*
- **input_data** (*str*) – If passed, will feed that data to the subprocess through stdin
- **out_pipe** (*int or file*) – File descriptor as passed to :ref:subprocess.Popen to use as stdout
- **err_pipe** (*int or file*) – File descriptor as passed to :ref:subprocess.Popen to use as stderr
- **of str** (*env (dict*) – str): If set, will use the given dict as env for the subprocess
- ****kwargs** – Any other keyword args passed will be passed to the :ref:subprocess.Popen call

Returns result of the interactive execution

Return type *lago.utils.CommandStatus*

lago.utils.**run_command_with_validation**(*cmd*, *fail_on_error=True*, *msg='An error has occurred'*)

lago.utils.**run_interactive_command**(*command*, *env=None*, ***kwargs*)
Runs a command interactively, reusing the current stdin, stdout and stderr

Parameters

- **command** (*list of str*) – args of the command to execute, including the command itself as command[0] as *['ls', '-l']*

- **of str** (*env (dict)* – str): If set, will use the given dict as env for the subprocess
- ****kwargs** – Any other keyword args passed will be passed to the :ref:subprocess.Popen call

Returns result of the interactive execution

Return type *lago.utils.CommandStatus*

lago.utils.**service_is_enabled**(*name*)

lago.utils.**sparse**(*input_file*, *input_format*, *fail_on_error=True*)

lago.utils.**ver_cmp**(*ver1*, *ver2*)
Compare lago versions

Parameters

- **ver1** (*str*) – version string
- **ver2** (*str*) – version string

Returns Return negative if ver1<ver2, zero if ver1==ver2, positive if ver1>ver2.

lago.utils.**with_logging**(*func*)

lago.validation module

lago.validation.**check_import**(*module_name*)
Search if a module exists, and it is possible to try importing it

Parameters **module_name** (*str*) – module to import

Returns True if the package is found

Return type bool

lago.virt module

class lago.virt.**VirtEnv**(*prefix*, *vm_specs*, *net_specs*)
Bases: object

Env properties: * prefix * vms * net

•libvirt_con

_create_net(*net_spec*)

_create_vm(*vm_spec*)

_get_stop_shutdown_common_args(*vm_names*)
Get the common arguments for stop and shutdown commands

Parameters **vm_names** (*list of str*) – The names of the requested vms

Returns

list of plugins.vm.VMProviderPlugin: vms objects that should be stopped

list of virt.Network: net objects that should be stopped str: log message

Raises utils.LagoUserException – If a vm name doesn't exist

_get_unused_nets(*vms_to_stop*)
Return a list of nets that used only by the vms in vms_to_stop

Parameters **vms_to_stop** (*list of str*) – The names of the requested vms

Returns

list of virt.Network: net objects that used only by vms in vms_to_stop

Raises utils.LagoUserException – If a vm name doesn't exist

bootstrap()

create_snapshots(**args*, ***kwargs*)

export_vms(*vms_names*, *standalone*, *dst_dir*, *compress*, *init_file_name*, *out_format*, *collect_only=False*, *with_threads=True*)

classmethod from_prefix(*prefix*)

generate_init(*dst*, *out_format*, *vms_to_include*, *filters=None*)
Generate an init file which represents this env and can be used with the images created by self.export_vms

Parameters

- **dst** (*str*) – path and name of the new init file
- **out_format** (plugins.output.OutFormatPlugin) – formatter for the output (the default is yaml)
- **filters** (*list*) – list of paths to keys that should be removed from the init file
- **(list of** (*vms_to_include*) – class:lago.plugins.vm.VMPlugin): list of vms to include in the init file

Returns None

get_compat()
Get compatibility level for this environment - which is the Lago version used to create this environment

get_env_spec(*filters=None*)
Get the spec of the current env. The spec will hold the info about all the domains and networks associated with this env.

Parameters **filters** (*list*) – list of paths to keys that should be removed from the init file

Returns the spec of the current env

Return type dict

get_net(*name=None*)

get_nets()

get_snapshots(*domains=None*)
Get the list of snapshots for each domain

Parameters

- **domanins** (*list of str*) – list of the domains to get the snapshots
- **all will be returned if noneor empty list passed**(*for,*) –

Returns with the domain names and the list of snapshots for each

Return type dict of str -> list(str)

get_vm(*name*)

get_vms(*vm_names=None*)
Returns the vm objects associated with vm_names if vm_names is None, return all the vms in the prefix

Parameters **vm_names** (*list of str*) – The names of the requested vms

Returns dict: Which contains the requested vm objects indexed by name

Raises utils.LagoUserException – If a vm name doesn't exist

prefixed_name(*unprefixed_name*, *max_length=0*)
Returns a uuid pefixed identifier

Parameters

- **unprefixed_name** (*str*) – Name to add a prefix to
- **max_length** (*int*) – maximum length of the resultant prefixed name, will adapt the given name and the length of the uuid ot fit it

Returns prefixed identifier for the given unprefixed name

Return type str

revert_snapshots(**args*, ***kwargs*)

save(**args*, ***kwargs*)

shutdown(*vm_names*, *reboot=False*)

start(*vm_names=None*)

stop(*vm_names=None*)

virt_path(**args*)

lago.virt.**_gen_ssh_command_id**()

lago.virt.**_guestfs_copy_path**(*g*, *guest_path*, *host_path*)

lago.virt.**_path_to_xml**(*basename*)

lago.vm module

class lago.vm.**DefaultVM**(*env*, *spec*)
Bases: *lago.plugins.vm.VMPlugin*

_abc_cache = <_weakrefset.WeakSet object>

_abc_negative_cache = <_weakrefset.WeakSet object>

_abc_negative_cache_version = 33

_abc_registry = <_weakrefset.WeakSet object>

class lago.vm.**SSHVMProvider**(*vm*)
Bases: *lago.plugins.vm.VMProviderPlugin*

bootstrap(**args*, ***kwargs*)

create_snapshot(*name*, **args*, ***kwargs*)

defined(**args*, ***kwargs*)

revert_snapshot(*name*, **args*, ***kwargs*)

start(**args*, ***kwargs*)

state(**args*, ***kwargs*)

stop(**args*, ***kwargs*)

lago.workdir module

CHAPTER 5

Releases

Release process

Versioning

For lago we use a similar approach to semantic versioning, that is:

```
X.Y.Z
```

For example:

```
0.1.0
1.2.123
2.0.0
2.0.1
```

Where:

- `Z` changes for each patch (number of patches since `X.Y` tag)
- `Y` changes from time to time, with milestones (arbitrary bump), only for backwards compatible changes
- `X` changes if it's a non-backwards compatible change or arbitrarily (we don't like `Y` getting too high, or big milestone reached, ...)

The source tree has tags with the `X.Y` versions, that's where the packaging process gets them from.

On each `X` or `Y` change a new tag is created.

For now we have only one branch (master) and we will try to keep it that way as long as possible, if at some point we have to support old versions, then we will create a branch for each `X` version in the form:

```
vX
```

For example:

```
v0
v1
```

There's a helper script to resolve the current version, based on the last tag and the compatibility breaking commits since then, to get the version for the current repo run:

```
$ scripts/version_manager.py . version
```

RPM Versioning

The rpm versions differ from the generic version in that they have the `-1` suffix, where the `-1` is the release for that rpm (usually will never change, only when repackaging without any code change, something that is not so easy for us but if there's any external packagers is helpful for them)

Repository layout

Tree schema of the repository:

```
lago
- stable <-- subdirs for each major version to avoid accidental
|   |          non-backwards compatible ugrade
|   |
|   - 0.0  <-- Contains any 0.* release for lago
|   |   - ChangeLog_0.0.txt
|   |   - rpm
|   |   |   - el6
|   |   |   - el7
|   |   |   - fc22
|   |   |   - fc23
|   |   - sources
|   - 1.0
|   |   - ChangeLog_1.0.txt
|   |   - rpm
|   |   |   - el6
|   |   |   - el7
|   |   |   - fc22
|   |   |   - fc23
|   |   - sources
|   - 2.0
|       - ChangeLog_2.0.txt
|       - rpm
|       |   - el6
|       |   - el7
|       |   - fc22
|       |   - fc23
|       - sources
- unstable <-- Multiple subdirs are needed only if branching
    - 0.0  <-- Contains 0.* builds that might or might not have
    |   |        been released
    |   - latest  <--- keeps the latest build from merged, static
    |   |                url
    |   - snapshot-lago_0.0_pipeline_1
    |   - snapshot-lago_0.0_pipeline_2
    |   |         ^ contains the rpms created on the pipeline build
    |   |           number 2 for the 0.0 version, this is needed to
```

```
|   |           ease the automated testing of the rpms
|   |
|   - ... <-- this is cleaned up from time to time to avoid
|               using too much space
- 1.0
|   - latest
|   - snapshot-lago_1.0_pipeline_1
|   - snapshot-lago_pipeline_2
|   - ...
- 2.0
    - latest
    - snapshot-lago_2.0_pipeline_1
    - snapshot-lago_2.0_pipeline_2
    - ...
```

Promotion of artifacts to stable, aka. releasing

The goal is to have an automated set of tests, that check in depth lago, and run them in a timely fashion, and if passed, deploy to stable. As right now that's not yet possible, so for now the tests and deploy is done manually.

The promotion of the artifacts is done in these cases:

- New major version bump (`X+1.0`, for example `1.0 -> 2.0`)
- New minor version bump (`X.Y+1`, for exampyre `1.1 -> 1.2`)
- If it passed certain time since the last `X` or `Y` version bumps (`X.Y.Z+n`, for example `1.0.1 -> 1.0.2`)
- If there are blocking/important bugfixes (`X.Y.Z+n`)
- If there are important new features (`X.Y+1` or `X.Y.Z+n`)

How to mark a major version

Whenever there's a commit that breaks the backwards compatibility, you should add to it the pseudo-header:

```
Sem-Ver: api-breaking
```

And that will force a major version bump for any package built from it, that is done so in the moment when you submit the commit in gerrit, the packages that are build from it have the correct version.

After that, make sure that you tag that commit too, so it will be easy to look for it in the future.

The release procedure on the maintainer side

1. Select the snapshot repo you want to release
2. **Test the rpms, for now we only have the tests from projects that use it:**
    - Run all the ovirt tests on it, make sure it does not break anything, if there are issues -> open bug
    - **Run vdsm functional tests, make sure it does not break anything, if** there are issues -> open bug
3. **On non-major version bump `X.Y+1` or `X.Y.Z+n`**
    - Create a changelog since the base of the tag and keep it aside
4. **On Major version bump `X+1.0`**

- **Create a changelog since the previous `.0` tag (`X.0`) and keep** it aside

5. Deploy the rpms from snapshot to dest repo and copy the `ChangeLog` from the tarball to `ChangeLog_X.0.txt` in the base of the `stable/X.0/` dir
6. Send email to lago-devel with the announcement and the changelog since the previous tag that you kept aside, feel free to change the body to your liking:

```
Subject: [day-month-year] New lago release - X.Y.Z

Hi everyone! There's a new lago release with version X.Y.Z ready for you to
upgrade!

Here are the changes:
    <CHANGELOG HERE>

Enjoy!
```

CHAPTER 6

Changelog

Here you can find the full changelog for this version

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y