Lago Documentation

Release 0.41.0

David Caro

Jul 31, 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 install it on any Linux distribution that can run libvirt and qemu-kvm using *pip*, here we provide instructions also for Ubuntu 16.04 which we test from time to time. Feel free to open PR if you got it running on a distribution which is not listed here so it could be added.

pip

1. Install system package dependencies:

   (a) CentOS 7.3+

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

   (a) Fedora 24+

```
$ dnf install -y python2-devel libvirt libvirt-devel \
    libguestfs-tools libguestfs-devel gcc libffi-devel \
    openssl-devel qemu-kvm
```

   (a) Ubuntu 16.04+

```
$ apt-get install -y python-dev build-essential libssl-dev \
    libffi-dev qemu-kvm libvirt-bin libvirt-dev pkg-config \
    libguestfs-tools libguestfs-dev
```

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):

   - Fedora / CentOS:

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

   - Ubuntu 16.04+ :

```
$ sudo usermod -a -G libvirtd,kvm USERNAME
$ chmod 0644 /boot/vmlinuz*
```

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'd like to store the templates in a different location look at the Configuration section, and change `lease_dir`, `template_repos` and `template_store` accordingly. This can be done after the installation is completed.

7. Restart libvirt:

```
$ systemctl restart libvirtd
```

8. Log out and login again

Thats it! Lago should be working now. You can jump to Lago Examples.

RPM Based - 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.

[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 Ubuntu *python-guestfs*.

Manual RPM installation

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

   For Fedora:

```
[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 RPM 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

Repository metadata

A templates repository should contain a *repo.metadata* file describing all available templates. The repository build script creates this file automatically. The file contains a serialized JSON object with the members detailed below. For an example, see the above repository's metadata file.

`name`: The name of the repository.

`sources`:

`<name>`: Name of a source.

`type`: Source type. May be either `http` or `file`.

`args`: Varies depending on the source type.

For an `http` source, should contain a `baseurl` member pointing to the root of the repository on the web.

For a `file` source, should contain a `root` member pointing to the root of the repository on the filesystem.

`templates`:

`<name>`: Unique template name.

`versions`:

`<version>`: Unique version string.

source: Name of the source from which this template version was created.

timestamp: Creation time of the template version.

handle: Either a base file name of the template located in the root directory of the repository, or a root-relative path to the template file.

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

lago.providers package

Subpackages

lago.providers.libvirt package

Submodules

lago.providers.libvirt.cpu module

lago.providers.libvirt.network module

lago.providers.libvirt.utils module

lago.providers.libvirt.vm module

Submodules

lago.brctl module

lago.build module

lago.cmd module

lago.config module

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

lago.guestfs_tools module

lago.lago_ansible module

lago.log_utils module

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

lago.service module

lago.ssh module

lago.subnet_lease module

lago.sysprep module

lago.templates module

lago.utils module

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

lago.vm module

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

U

V

Y