Lago Documentation

Release 0.3

David Caro

February 05, 2016

Contents

CHAPTER 1

Getting started

Check out the awesome README!

CHAPTER 2

Releases

2.1 Release process

2.1.1 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
```

2.1.2 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)

2.1.3 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
        -- ...
```

2.1.4 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`)

2.1.5 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.

2.1.6 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 3

Developing

3.1 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.

3.1.1 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.

3.1.2 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 :)

3.1.3 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:

3.1.4 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.

3.1.5 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.

CHAPTER 4

Contents

4.1 lago package

4.1.1 Subpackages

lago.plugins package

Submodules

lago.plugins.cli module

4.1.2 Submodules

4.1.3 lago.brctl module

4.1.4 lago.cmd module

4.1.5 lago.config module

4.1.6 lago.constants module

4.1.7 lago.dirlock module

4.1.8 lago.log_utils module

4.1.9 lago.paths module

4.1.10 lago.subnet_lease module

4.1.11 lago.sysprep module

4.1.12 lago.templates module

4.1.13 lago.utils module

4.1.14 lago.virt module

4.2 lago_template_repo package

4.3 ovirtlago package

CHAPTER 5

Indices and tables

- genindex
- modindex
- search