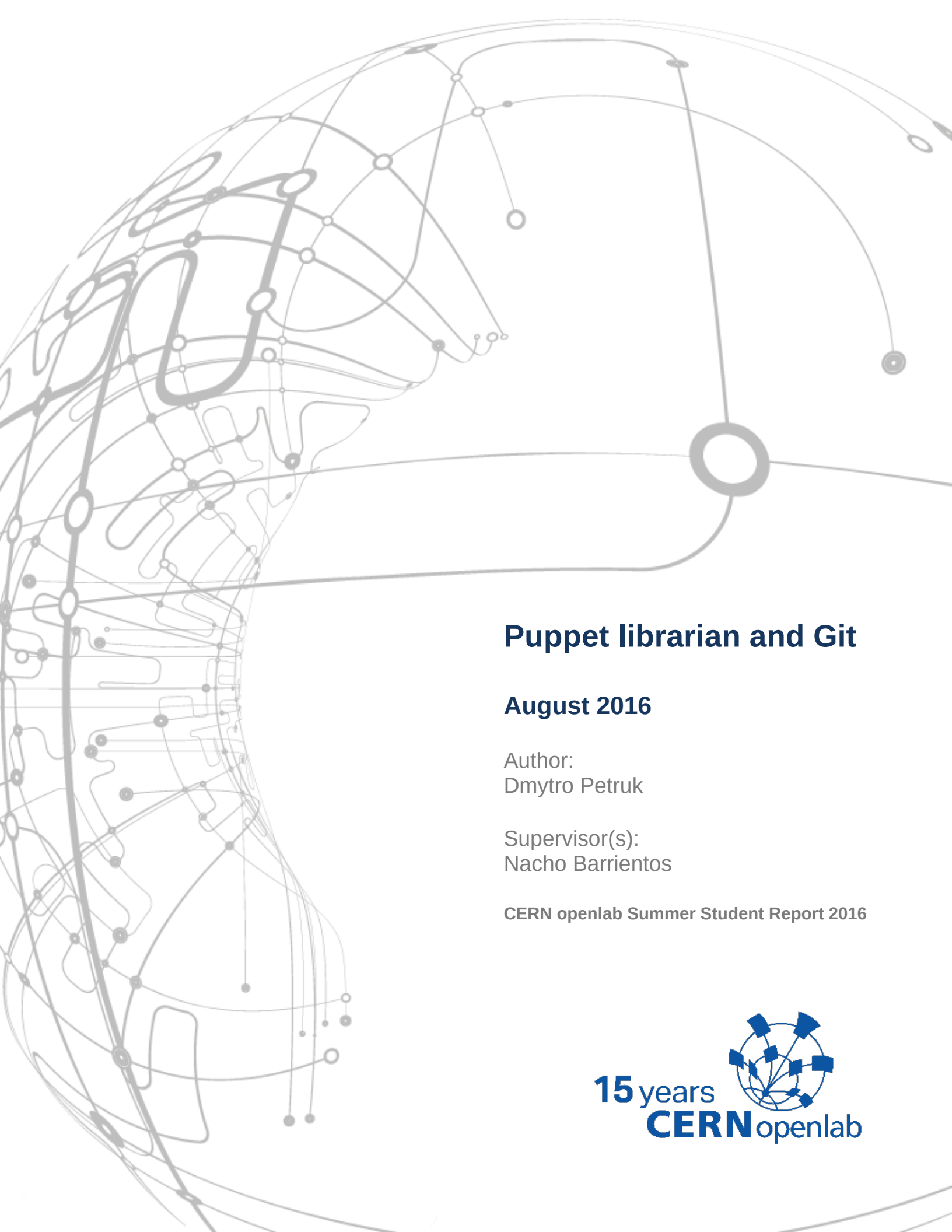


Puppet librarian and Git

August 2016

Author:
Dmytro Petruk

Supervisor(s):
Nacho Barrientos

CERN openlab Summer Student Report 2016

15 years CERN openlab

Project Specification

Puppet is used by CERN extensively in order to manage server configuration of over 23k servers in its data centers. The project is to evaluate Python libraries which interface to the Git source code management system and enhance the CERN Jens Puppet library component to improve performance and build on community contributions for better future compatibility.

Abstract

To have the most up to date version of the configuration files for the servers, Jens interfaces directly to the Git binaries to synchronize GitLab repositories and Puppet environments.

The initial implementation of the Git module, which is responsible for interacting with Git repositories, was built on just creating a new Git process for each operation. The aim of this project was to evaluate all the libraries in the wild for interacting with Git, go for the best and rewrite git.py using it but also to improve error processing and increase test coverage for better future development.

Table of Contents

1 Introduction

The main tools, technologies and methodologies used in this project were Puppet, Jens, Python and unit testing. Here is a brief description of each one:

- **Puppet**[1]
  It is a tool designed to manage the configuration of Unix-like and Microsoft Windows systems declaratively. The user describes system resources and their state, either using Puppet's declarative language or a Ruby DSL (domain-specific language). This information is stored in files called "Puppet manifests". Puppet discovers the system information via a utility called Facter, and compiles the Puppet manifests into a system-specific catalog containing resources and resource dependency, which are applied against the target systems. Any actions taken by Puppet are then reported.

- **Jens**[2]
  Jens is the Puppet modules/hostgroups librarian used by the CERN IT department.

  It is basically a Python toolkit that generates Puppet environments dynamically based on some input metadata and maintains them after. So, it's Jens' responsibility to distribute the new version of the module when changes have been pushed to the repository. Jens is useful in sites where there are several administrators taking care of siloed services (mapped to what we call top-level "hostgroups") with very service-specific configuration but sharing configuration logic via modudules.

  Jens has been used as the production Puppet librarian at CERN IT since August 2013.

  This tool covers the need of several roles that might show up in a typical shared Puppet infrastructure:

- Developers writing manifests who want an environment to test new code: Jens provides dynamic environments that automatically update with overrides for the modules being developed that point to development branches.
- Administrators who don't care: Jens supports simple dynamic environments that default to the production branch of all modules and that only update automatically when there's new production code.
- Administrators looking for extra stability who are reluctant to do rolling updates: Jens implements snapshot environments that are completely static and don't update unless redefined, as all modules are pinned by commit identifier.

In Jens' realm, Puppet environments are basically a collection of modules, hostgroups, hierarchies of Hiera data and a site.pp. These environments are defined in environment definition files which are stored in a separate repository that's known to the program. Also, Jens makes use of a second metadata repository to know what modules and hostgroups are part of the library and are therefore available to generate environments.

With all this information, Jens produces a set of *environments* that can be used by Puppet masters to compile Puppet *catalogs*. Two types of environments are supported: dynamic and static. The former update automatically as new commits arrive to the concerned repositories whereas the latter remain static pointing to the specified commits to implement the concept of "configuration snapshot".

Jens is composed by several CLIs: *jens-config, jens-gc, jens-reset, jens-stats and jens-update* to perform different tasks. Manual pages are shipped for all of them.

Basically, the input data that's necessary for an execution of *jens-update* (the core tool provided by this toolset) is two Git repositories:

- The repository metadata repository (or the library)
- The environment definitions repository (or the environments)

Jens uses a single YAML file stored in a Git repository to know what are the modules and hostgroups available to generate environments. Apart from that, it's also used to define the paths to two special Git repositories containing what's called around here the common Hiera data and the site manifest.

This is all set up via two configuration keys: *repositorymetadata* (which is the directory containing a clone of the repository) and *repositorymetadatadir* (the file itself).

The following is how a skeleton of the file looks like:

```
---
repositories:
  common:
      hieradata: http://git.example.org/pub/it-puppet-common-hieradata
      site: http://git.example.org/pub/it-puppet-site
  hostgroups:
      ...
      aimon: http://git.example.org/pub/it-puppet-hostgroup-aimon
      cloud: http://git.example.org/pub/it-puppet-hostgroup-cloud
      ...
  modules:
      ...
      apache: http://git.example.org/pub/it-puppet-module-apache
```

```
bcache: http://git.example.org/pub/it-puppet-module-bcache
...
```

The idea is that when a new top-level hostgroup is added or a new module is needed this file gets populated with the corresponding clone URLs of the repositories. Jens will add new elements to all the environments that are entitled to get them during the next run of *jens-update*.

What's a Jens run?

It's an execution of *jens-update*, which is normally trigged by a cronjob. It will determine what's new, what branches have to be updated and what environments have to be created/modified/deleted. The following is an example of what's typically found in the log files after a run where there was not much to do (a hostgroup got new code in the QA branch and a new environment was created):

```
INFO Obtaining lock 'aijens' (attempt: 1)...
INFO Refreshing metadata...
INFO Refreshing repositories...
INFO Fetching repositories inventory...
INFO Refreshing bare repositories (modules)
INFO New repositories: []
INFO Deleted repositories: []
INFO Cloning and expanding NEW bare repositories...
INFO Expanding EXISTING bare repositories...
INFO Purging REMOVED bare repositories...
INFO Refreshing bare repositories (hostgroups)
INFO New repositories: []
INFO Deleted repositories: []
INFO Cloning and expanding NEW bare repositories...
INFO Expanding EXISTING bare repositories...
INFO Updating ref '/mnt/puppet/aijens-
3afegt67.cern.ch/clone/hostgroups/vocms/qa'
INFO Purging REMOVED bare repositories...
INFO Refreshing bare repositories (common)
INFO New repositories: []
INFO Deleted repositories: []
INFO Cloning and expanding NEW bare repositories...
INFO Expanding EXISTING bare repositories...
INFO Purging REMOVED bare repositories...
INFO Persisting repositories inventory...
INFO Executed 'refresh_repositories' in 6287.78 ms
INFO Refreshing environments...
INFO New environments: ['am1286']
INFO Existing and changed environments: []
INFO Deleted environments: []
INFO Creating new environments...
INFO Creating new environment 'am1286'
INFO Processing modules...
INFO Processing hostgroups...
INFO hostgroups 'aimon' overridden to use treeish 'am1286'
INFO Processing site...
INFO Processing common Hiera data...
INFO Purging deleted environments...
INFO Recreating changed environments...
INFO Refreshing not changed environments...
INFO Executed 'refresh_environments' in 1395.03 ms
INFO Releasing lock 'aijens'...
```

```
INFO Done
```

One of the modules of the tool is named "*git.py*" and it is responsible for performing any kind of interaction with Git repositories. The available operations are: *"gc", "hash-object", "clone", "fetch", "reset" and "show-refs"*. Versions up to 0.14-1 were formatting commands, setting environmental variables if needed, creating new *nix subprocesses, waiting for output from the process and analyzing the output and the return code.

- **Python[3]**
  Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

- **Unit testing[4]**
  It is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing.
  Nosetests and Python's unittest module were used in this project.

2 Previous implementation

To be able to perform basic operations on Git repositories Jens used to start subprocesses which communicated with Git binaries on the system to *fetch/clone/reset/etc* repositories. Working on such low level of abstraction implied rudimentary error handling, low level calls and high cost of further modifications and extensions due to redundant code.

Thus, the goal of the project was to stop using this custom made solution and to use a popular and maintained solution (see the original issue for more details: issue#8)

3 Development

The first step of the project was to evaluate popular open source libraries for Python to work with Git repositories.

After comparing capabilities of a number of existing libraries (*pygit2, GitPython, python-git, dulwich, StGit, Gittle*) the choice has been made towards *GitPython's* direction, since it is actively maintained, highly flexible, it allows to use a large number of wrapped functions that can be customized and it possesses an extensive documentation.

The next step was to re-implement the same functionality using *GitPython*. For achieving this step, existing unit tests made a huge impact on the development speed since it was easy to see whether new partial implementations seemed to behave in a correct way just by running the test suite.

In the meanwhile, a few bugs were fixed and code was refactored to achieve higher readability and maintainability (e.g. reducing number of arguments for a lot of methods by reimplementing the settings class as a singleton object; see the merge request: MR#15)

A new feature to abort Git operations after a certain time limit has also been introduced.

Now if some operations exceed a specified time, the connection is cut instead of waiting forever until the server replies (see the merge request: MR#14)

Moreover, some tests have been added to validate code more extensively (see the MR: MR#9)

4 Deployment

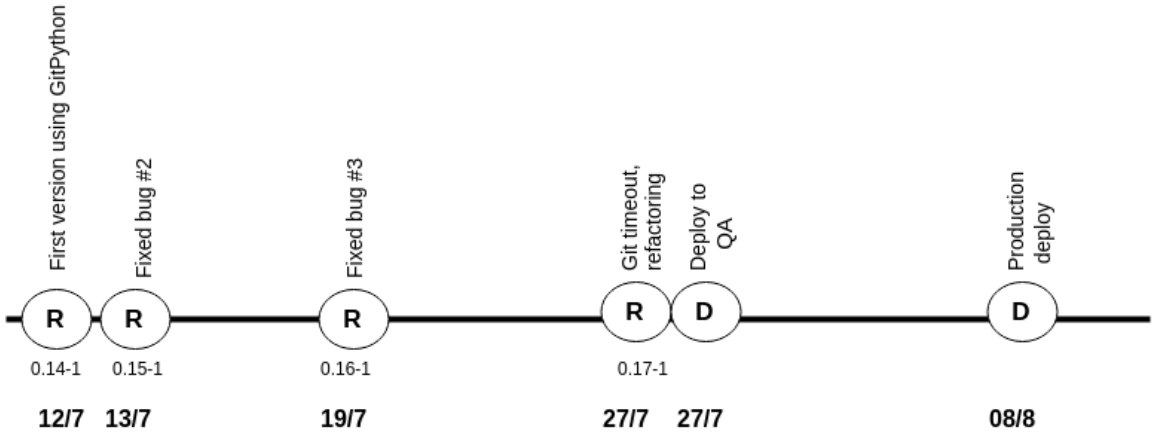
The next step was to deploy a new version of Jens and test it in a real environment.

An RPM package has been built and deployed through CERN's central package repository to the Development instance, where it was consuming data from production, though results were not used anywhere further, besides the log files that were monitored to try to spot issues. During this phase a few bugs and problems were discovered and fixed in the meanwhile, some examples:

1. Absence of the needed version of the library installed on the *target OS for deployment (CERN CentOS7)*. At first, the newest version of the library has been used for development (*2.0.8*), but it turned out that only version *1.0.1 was available*. So, it was necessary to rewrite some of the already written code.

2. Leakage of file descriptors. The available version of the library doesn't close files properly, leading to an exhaustion of file descriptors available for the process. The problem was fixed after following a suggestion of the library's author found on GitHub.
3. Failure to perform hard resets on repositories. It turned out that there was a bug in the library and an another way of doing this operation needed to be used. This bug wasn't spotted by the tests, but by analyzing log files. So, tests to validate this functionality were added.

Then, after testing on the Development instance, the latest Jens was deployed on QA instances, where it was running in parallel with production nodes. After this deployment we decided to freeze and not to release more versions for a week. And given that no more bugs were encountered during this phase, it was safe to deploy to the production servers.



Timeline of the development and deployment process

5 Summary

Thus, after going through the entire process of the software development, a new release of Jens has been rolled out and it is running in production today. As a result, all project goals have been achieved and version *0.17-1* of Jens has improved error handling and increased test coverage.

During the implementation of this project some lessons have been learnt:

- Tests save time in the future, due to the smaller probability of having code regressions.
- It's not always possible to have the desired environment (e.g. libraries on the servers).

6 Extra work

Due to the fact that the main project was finished way before the deadline, I received another task to work on. It was to write an extension to the CLI tool named *ai-foreman*, which is a wrapper for the Foreman API used by CERN's IT department. My task was to design and implement functionality to create and remove hostgroups.

Foreman groups hosts using hostgroups which before could only be created and deleted using the UI as there was no CLI interacting to the API. This is an easy task when there's only a single hostgroup to create or remove, however it gets more cumbersome when an entire new hostgroup tree has to be added or removed. To alleviate this situation, the task was to write a new subcommands for ai-foreman to add and remove hostgroups, recursively if needed.

1. Addhostgroup

   The aim of this command was to be able to add the specified hostgroup. In the situation when the complete tree of the parent hostgroups does not exist, there has to be the possibility to enforce the creation of all necessary hostgroups recursively.

   Options:

   - [HOSTGROUP] ...

     A list of hostgroups to create. Example: "foo/bar".

   - -p, --parents PARENTS

     Create parent hostgroups if they don't exist yet.

   Implementation was written in a form of a simple recursive function, which goes up through the hostgroup tree until finding an existing hostgroup from the hierarchy and then creating hostgroups one-by-one.

   If the *parents* option is not enabled and the parent tree does not exist an exception is raised.

   Examples :

   - *$ ai-foreman addhostgroup playground/ibarrien/bar/baz*

     Could not create hostgroup 'baz' in Foreman because some parts of the hostgroup hierarchy (playground/ibarrien/bar) do not exist (use –p to create hostgroups recursively)

- *$ ai-foreman addhostgroup playground/ibarrien/bar/baz –p*

  Creating hostgroup ‘playground/ibarrien/bar’...

  Hostgroup ‘playground/ibarrien/bar’ created in Foreman

  Creating hostgroup ‘playground/ibarrien/bar/baz’...

  Hostgroup ‘playground/ibarrien/bar/baz’ created in Foreman

2. Delhostgroup

The delhostgroup command was needed in order to remove a specific hostgroup. If the hostgroup possesses children hostgroups – there should be possibility to enforce removal of them. If the hostgroup (or its’ children) has any hosts – no hostgroups should be deleted.

Options:

- [HOSTGROUP] ...

  A list of hostgroups to remove. This subcommand will never delete hostgroups if it finds any host when traversing the hostgroup tree, regardless of the presence of -r flag. Example: "foo/bar".

- -r, -R, --recursive RECURSIVE

  Remove the hostgroup and its children recursively.

At first, a list of all potential hostgroups for removal is prepared, in DFS order. This step is needed to be done separately, to be able to verify the absence of hosts for every potential hostgroup to be removed, before actually removing them. If none of those hostgroups has hosts – hostgroups are deleted from the last visited.

Examples :

- *$ ai-foreman delhostgroup playground/ibarrien/bar*

  Hostgroup playground/ibarrien/bar can not be deleted since it contains children hostgroups. Use --recursive option

- *$ ai-foreman delhostgroup playground/ibarrien/bar –r*

  Removing hostgroup ‘playground/ibarrien/bar/baz’...

  Hostgroup ‘playground/ibarrien/bar/baz’ removed

  Removing hostgroup ‘playground/ibarrien/bar’...

  Hostgroup ‘playground/ibarrien/bar’ removed

During the development of these functionalities, a number of both unit and functional tests were added

7 Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Nacho for his continuous support duing my internship project, for his patience and motivation. His guidance helped me all the time.

Also, I would like to thank Akos, who helped me to overcome problems during my second project.

8 Bibliography

1. Puppet wikipedia
2. Jens GitHub
3. Python wikipedia
4. Unit testing wikipedia