Write Once, Run Everywhere: Beaker & Puppet Labs

Alice Nodelman

alice@puppetlabs.com

Abstract

Over the past two years, Puppet Labs has developed the open source Beaker acceptance testing tool. Beaker provides 'write once, run everywhere' testing. The same test can be run locally or in the cloud, it can exercise Puppet from source, Puppet from package or Puppet from enterprise installation, and it can be executed against all of Puppet Enterprise's 14 supported platforms which includes *nix, OS X, and Windows.

Beaker is written in Ruby and uses object inheritance to create a host abstraction. Test writers interact with the abstraction, while Beaker manipulates the actual platform under test. Beaker also includes a domain specific language (DSL) which wraps common testing tasks. Combined together, the host abstraction and DSL make test writing cheap and easy.

Building a new testing system is expensive and complicated. It requires the same effort as any other software development project, including designing, developing, documenting, testing and ongoing maintenance. The significant investment that Puppet Labs has put into Beaker has resulted in the creation of over two thousand Beaker acceptance tests - with more added every day by both employees and community members. Any regression discovered is converted into a Beaker test to ensure future behavior. Puppet Labs project builds are now subjected to days of testing effort without any manual intervention.   Beaker is central to Puppet Labs' automation infrastructure and essential to the dependability of Puppet Labs software.

Biography

For over a decade, Alice Nodelman has specialized in automated testing of hard-to-test software. She believes that software should keep its promises. She has mocked out online marketplaces, generated giant databases, replicated web browsing, and otherwise pressed buttons that she shouldn't have. Currently a Senior Quality Engineer with Puppet Labs, she is lead developer of Beaker — an open source, multi-platform, multi-node, multi-cloud acceptance testing tool. Someday she will automate everything.

1 Introduction

Puppet Labs provides IT automation solutions to easily automate repetitive tasks and centrally manage configuration of systems. The software supports a wide variety of operating systems, from *nix-style to Windows and OS X. Puppet can be deployed on as little as one node, or throughout an entire data center. The combination of the number of supported platforms and the deployment methods makes Puppet an incredibly complicated product to test. In addition, Puppet supports a thriving open source module, or plugin, community that needs to test their contributions reliably. An automated testing system that could use computer cycles and free person-hours was a necessity.

Four main areas of functionality were identified for a Puppet acceptance testing tool:

1. Create understandable, platform agnostic tests to avoid code duplication
2. Flexibly provision test nodes locally, on the internal network, or in various cloud systems
3. Provide a variety of methods to install Puppet, either directly from source, as an open source package or from the enterprise installer
4. Be available both to employees and community members

With this problem space defined, the Beaker automated testing tool was created.

1.1 Terminology

Object inheritance is the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes.

Puppet is a configuration management solution that allows you to define the state of your IT infrastructure, and then automatically enforces the desired state.

Cloud Provisioning is the creation of new virtual instances on an internal or external cloud computing provider.

Software is tested on a *System Under Test* (SUT), which could be the user's own hardware, local Virtual Machines (VM), or cloud-provisioned boxes. They can also be termed *nodes*, *boxes*, *machines, instances,* or *VMs*.

The *full stack* describes all of the front and backend dependencies necessary to install and run a project – from the hardware, to libraries, databases, and external services.

Acceptance or *acceptance level* tests are those that ensure a piece of software keeps its promises or contract. These are full stack tests on 'clean' (non-development) environments that match a user's projected environment. These tests confirm that the software can be installed and interacted with, regressions are absent and functionality is verified. The software itself is treated as a black box that tests interact with in the same way a user would.

A *hypervisor* or *virtual machine monitor* (VMM) is a piece of computer software, firmware, or hardware that creates and runs virtual machines.

Provisioning is a mechanism for creating nodes or *instances* in a particular hypervisor.

An *Application Program Interface* (API) is a precise specification written by providers of a service that programmers must follow when using that service. It describes what functionality is available, how it must be used, and what formats it will accept as input or return as output.

A *domain-specific language* (DSL) is a computer language specialized to a particular application domain, in this case custom created to aid Puppet Labs automated acceptance testing.

2 Beaker Basics

Beaker is a tool that creates and communicates with a set of SUTs. Beaker provisions and configures the SUTs, installs appropriate dependencies, and then executes tests on those nodes. All communication between Beaker and its SUTs is done over SSH (a UNIX-based command interface and protocol for securely getting access to a remote computer). Test commands are processed sequentially, one per SSH channel on a dedicated SSH connection.

Beaker is written entirely in Ruby. In the Puppet Labs ecosystem, Ruby is the *lingua franca* – known internally and externally. It was a natural choice to make it as easy as possible for people to become test authors – both employees and open source community members.

2.1 Test Files

Beaker tests are Ruby scripts. A single test is captured in a single file. Here is an example test that echoes 'hello' on each SUT:

```
test_name "say hello to all my hosts!"

hellos = 0
@hosts.each do |host|
  step "i'm going to say hello to #{host}"
  on host, 'echo hello'
  hellos += 1
end

#make sure that i've been polite!
assert_equal(hellos, hosts.length, \
             "#{hellos} <> #{hosts.length}")
```

From this example, you can see the use of a *hosts* object. The *hosts* represents an array of all SUTs currently under test. By looping through these one at a time we can execute code against each SUT.

Running against two test hosts, a CentOS 6 box and an Ubuntu Lucid box, results in the following console output:

```
Begin tests/confirm.rb

say hello to all my hosts!

  * i'm going to say hello to pe-ubuntu-lucid

pe-ubuntu-lucid 15:10:02$ echo hello
hello

pe-ubuntu-lucid executed in 0.71 seconds

  * i'm going to say hello to pe-centos6

pe-centos6 15:10:03$ echo hello
hello

pe-centos6 executed in 0.18 seconds
tests/hello.rb passed in 0.90 seconds
      Test Suite: tests @ 2015-06-03 15:10:02 -0700
```

```
            - Host Configuration Summary -


                  - Test Case Summary for suite 'tests' -
           Total Suite Time: 0.90 seconds
          Average Test Time: 0.90 seconds
                  Attempted: 1
                     Passed: 1
                     Failed: 0
                    Errored: 0
                    Skipped: 0
                    Pending: 0
                      Total: 1
```

Since no exceptions or assertions are generated during Beaker test execution, this test will pass. A Beaker test suite is a directory of Ruby files. Each individual Ruby file will be executed in alphanumeric order by file name.

2.2 Logging And Reporting

Beaker provides three methods for exploring and interpreting test results: terminal logging, plain text logging, and JUnit XML.

2.2.1 Terminal Logging

Beaker generates a large amount of information printed to the terminal during test execution. When the –-debug option is enabled, Beaker reports each command executed and the resulting output from each individual SUT.

Below demonstrates a command first executed on a CentOS 7 SUT and then executed on a Windows 2008 SUT.

```
 * Replace site.pp with a new manifest

host1.net (centos7-64-1) 05:54:43$ puppet agent --configprint
node_name_value
host1.net

host1 (centos7-64-1) executed in 0.48 seconds

host2.net (windows2008r2-64-2) 05:54:44$ cmd.exe /c puppet agent --
configprint node_name_value
host2.net
```

2.2.2 Plain Text Logs

After the test run, Beaker generates three log files in `./log/#{HOST_YAML_FILE}/#{TIME}`.

```
log/example_hosts_file.yml/2015-06-25_15_23_42
├── sut.log
├── tests-run.log
└── tests-summary.txt
```

The `sut.log` file contains information about the SUTs used during the last test run.

```
$ cat sut.log
2015-06-25 15:23:42    [+]    fusion    el-6-i386    pe-centos6
2015-06-25 15:23:42    [-]    fusion    el-6-i386    pe-centos6
```

The `tests-run.log` contains the same data printed to terminal during test execution. All test execution information is contained in this single, flat file.

`tests-summary.txt` has the overall pass/fail/error/skip results of the tests that were included in the run.

```
    Test Suite: tests @ 2015-06-26 05:46:50 -0700

      - Host Configuration Summary -

              - Test Case Summary for suite 'tests' -
       Total Suite Time: 545.70 seconds
      Average Test Time: 1.48 seconds
              Attempted: 369
                 Passed: 42
                 Failed: 0
                Errored: 0
                Skipped: 327
                Pending: 0
                  Total: 369

      - Specific Test Case Status -

Failed Tests Cases:
Errored Tests Cases:
Skipped Tests Cases:
  Test Case /var/lib/jenkins/workspace/qe_beaker_intn-sys_vpool-
pe372/agent/windows2008r2/master/centos7/pe_acceptance_tests/acceptance
/tests/aix/aix_package_provider.rb skip
  Test Case /var/lib/jenkins/workspace/qe_beaker_intn-sys_vpool-
pe372/agent/windows2008r2/master/centos7/pe_acceptance_tests/acceptance
/tests/puppetdb/puppetdb_cert_whitelist.rb skip
<snip> . . . . more skipped tests here . . . . . . . . . </snip>
```

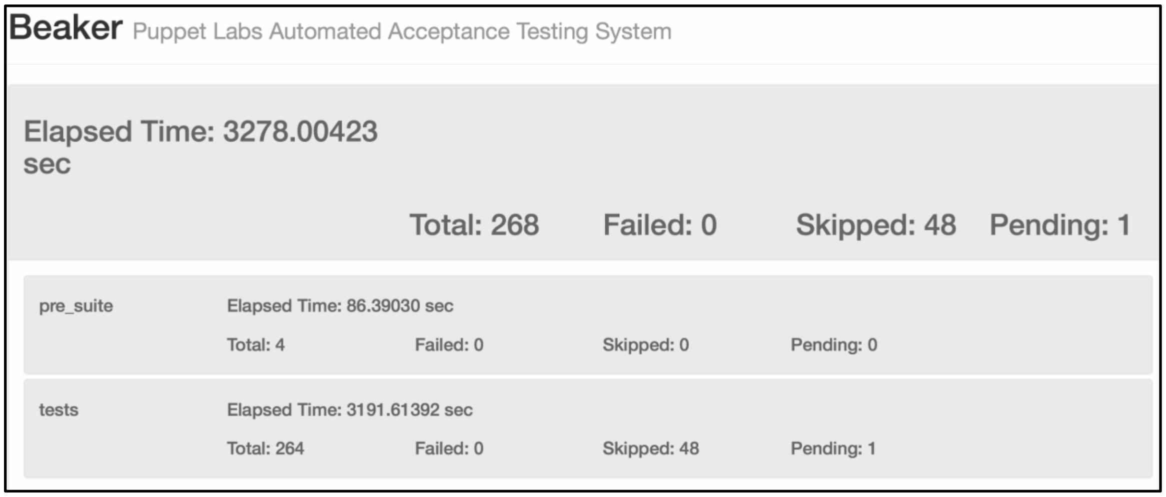
2.2.3 JUnit XML And Web Interface

Due to the difficulty of sorting and interpreting the amount of information captured in a Beaker test run all of the collected log information is converted to JUnit XML format (a popular Java unit test framework). This format can be interpreted by a number of third party tools, including a Jenkins plugin. The Beaker team has also created a web based interface using Bootstrap (a popular HTML, CSS, and JS framework for developing web projects), which is included in Beaker.

The JUnit web interface color codes test output green, orange and red for passed, skipped and failed/errored tests, respectively. The test execution log is initially collapsed, but can be expanded by clicking on a specific test.

Post testing you can view this interface by loading the file `./junit/log/#{HOST_YAML_FILE}/#{TIME}/beaker_junit.xml` in your web browser.

First look:

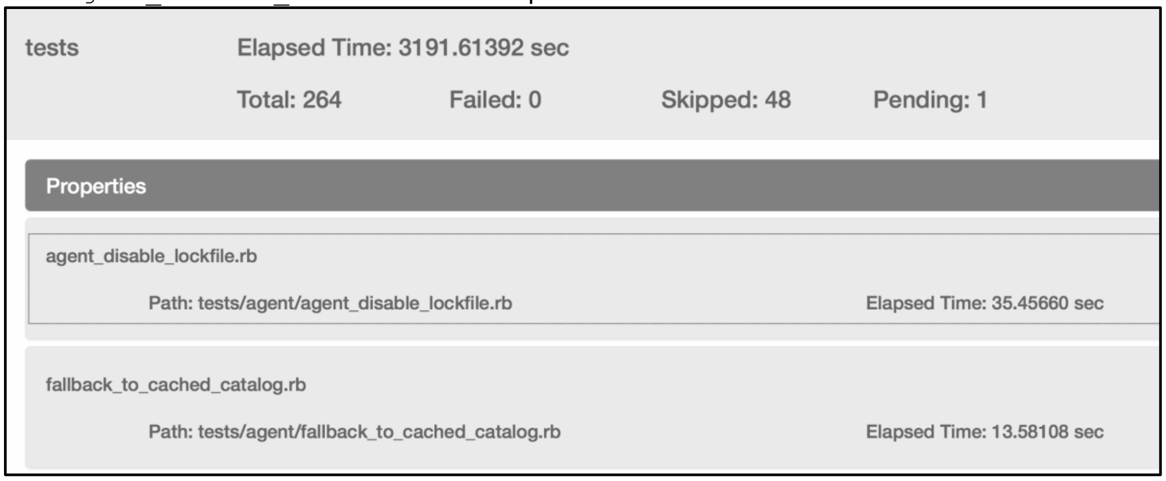

Beaker Puppet Labs Automated Acceptance Testing System

Elapsed Time: 3278.00423 sec

Total: 268 Failed: 0 Skipped: 48 Pending: 1

pre_suite	Elapsed Time: 86.39030 sec			
	Total: 4	Failed: 0	Skipped: 0	Pending: 0
tests	Elapsed Time: 3191.61392 sec			
	Total: 264	Failed: 0	Skipped: 48	Pending: 1

With `agent_disable_lockfile.rb` collapsed:


tests Elapsed Time: 3191.61392 sec

Total: 264 Failed: 0 Skipped: 48 Pending: 1

Properties

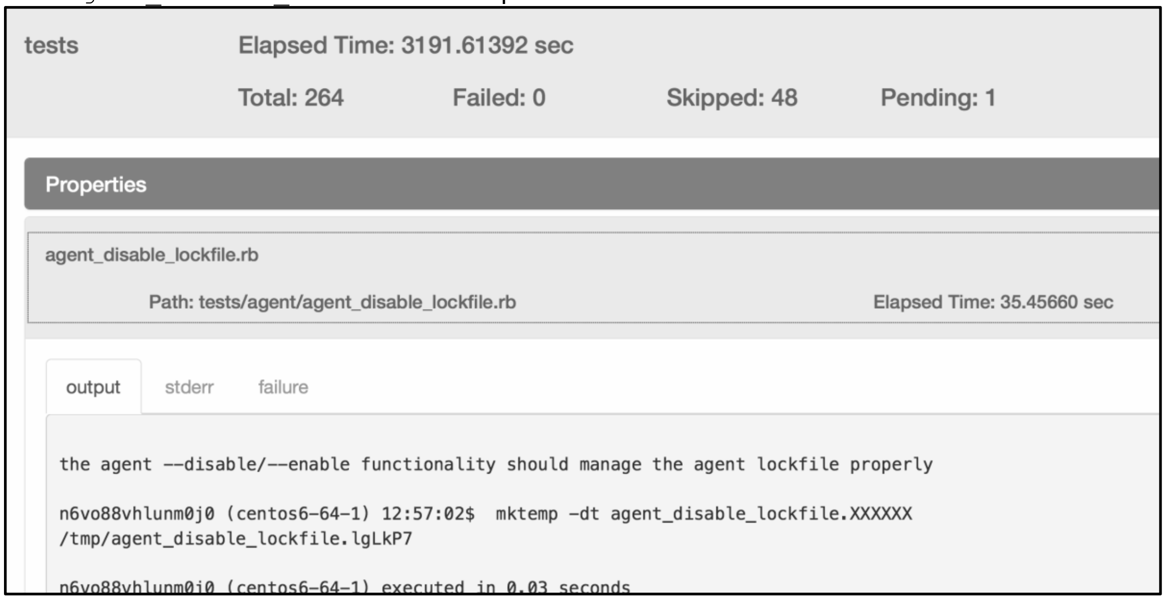
agent_disable_lockfile.rb

Path: tests/agent/agent_disable_lockfile.rb Elapsed Time: 35.45660 sec

fallback_to_cached_catalog.rb

Path: tests/agent/fallback_to_cached_catalog.rb Elapsed Time: 13.58108 sec

With `agent_disable_lockfile.rb` expanded:


tests Elapsed Time: 3191.61392 sec

Total: 264 Failed: 0 Skipped: 48 Pending: 1

Properties

agent_disable_lockfile.rb

Path: tests/agent/agent_disable_lockfile.rb Elapsed Time: 35.45660 sec

```
the agent --disable/--enable functionality should manage the agent lockfile properly

n6vo88vhlunm0j0 (centos6-64-1) 12:57:02$  mktemp -dt agent_disable_lockfile.XXXXXX
/tmp/agent_disable_lockfile.lgLkP7

n6vo88vhlunm0j0 (centos6-64-1) executed in 0.03 seconds
```

3 The Many Platforms Problem

Puppet supports a wide variety of platforms - from Red Hat, Ubuntu, and OpenBSD to Windows, AIX, and many more. Consider doing a simple task such as creating a temporary file.

On a Unix or Linux system it would look something like:

```
$ mktemp -t tmpname.XXXXXX
```

On AIX it would be:

```
$ rndnum=${RANDOM} && touch /tmp/tmpname.${rndnum} && echo
/tmp/tmpname.${rndnum}
```

On Windows:

```
> echo C:\Windows\Temp\tmpname.%RANDOM%
```

An automated test requiring the creation of a temporary file on a SUT would be:

```
if platform =~ /unix/
  # unix only code
elsif platform =~ /aix/
  # aix only code
elsif platform =~ /windows/
  # windows only code
else
  # some fallthrough here
end
```

The code quickly becomes difficult to read and maintain. Any new platform will require the rewrite of all tests to ensure that functionality is preserved.

3.1 The Beaker Host Abstraction

Beaker wraps all of this complexity in a host abstraction. Test authors do not have to consider what platform a test is to be executed on, they only have to consider what part of the software they are attempting to exercise.

For temporary file creation, Beaker reduces the test code to:

```
host.tmpfile('tmpname')
```

Behind the scenes Beaker does the right thing per-platform. If more platforms are added Beaker can be updated without having to revise individual tests.

3.2 The Beaker Host Inheritance Architecture

To manage the many platforms problem, Beaker depends upon a host object inheritance architecture. Each Beaker host is described in a YAML-formatted host configuration file provided by the tester. A host definition is required to contain a platform string that describes a supported platform type. The platform string determines the host object that is created to handle interaction with that node. The individual host types (windows, pswindows, unix, freebsd, aix, mac) inherit the basic host operations (ssh connection, scp, parameter storage) and then provide their own appropriate methods for platform dependent methods (temporary file creation, package installation, environment variable creation, etc).

An example `hosts.yaml` file:

```
HOSTS:
  node1:
    roles:
      - master
        - agent
        - dashboard
        - database
    platform: el-6-i386
    snapshot : clean-w-keys
    hypervisor : fusion
  node2:
    roles:
      - agent
    platform: windows-2008r2-x86_64
    snapshot : clean-w-keys
    hypervisor : fusion
```

The platform for `node1` is `el-6-i386` and the platform for `node2` is `windows-2008r2-x86_64`. Individual, platform-specific host objects are created by comparing the provided platform string to known, supported operating system types.

Host creation based upon provided platform string:

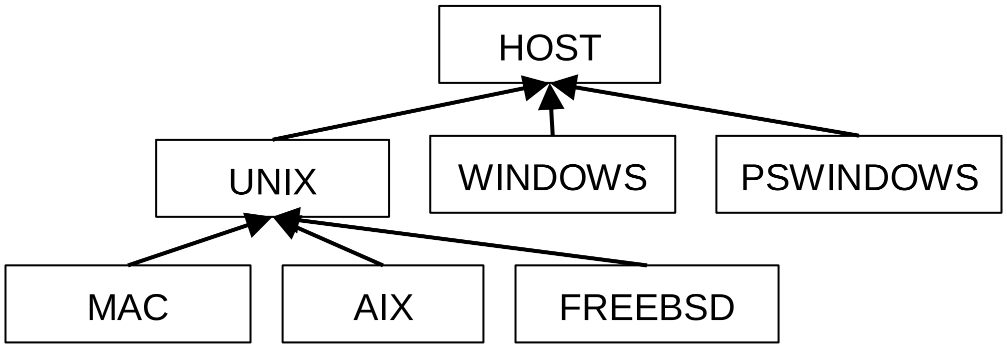
```
def self.create name, host_hash, options
  case host_hash['platform']
    when /windows/
      cygwin = host_hash['is_cygwin']
      if cygwin.nil? or cygwin == true
        Windows::Host.new name, host_hash, options
      else
        PSWindows::Host.new name, host_hash, options
      end
    when /aix/
      Aix::Host.new name, host_hash, options
    when /osx/
      Mac::Host.new name, host_hash, options
    when /freebsd/
      FreeBSD::Host.new name, host_hash, options
    else
      Unix::Host.new name, host_hash, options
  end
end
```

The node defined as `el-6-i386` will become a `Host::Unix` object, while the node defined as `windows-2008-x86_64` will become a `Host::Windows` object.

There are currently two Windows style objects: `Host::Windows` and `Host::PSWindows.` `Host::Windows` objects are assumed to run on Windows nodes that have Cygwin (a Unix-like environment and command-line interface for Microsoft Windows) installed, while `Host::PSWindows` nodes rely on native PowerShell commands.

The Beaker inheritance structure makes it simple to add new platform types. Simply inherit from the appropriate location – either from the base `Host` object or one of its children – and then override any platform-dependent code.

The Beaker Host object inheritance architecture:



3.3 Not Everything Can Be Platform Agnostic

Some tests are simply too platform specific to be shared. Maybe you are testing adding registry entries on Windows, which has no analog on non-Windows systems. For these situations, the Beaker DSL provides a custom-built method: *confine*. When included at the beginning of a test, *confine* informs Beaker in which situations an individual test can be executed. It is the exception to the 'write once, run everywhere' Beaker motto.

Confine to only Windows SUTs:

```
confine(:to, :platform => 'windows')
```

Confine to everything except Windows SUTs:

```
confine(:except, :platform => 'windows')
```

4 The Many Hypervisors Problem

Acceptance tests are executed for a variety of reasons:

- Locally, on a developer's box to ensure that their new code does not cause a regression
- Within a continuous integration pipeline
- As part of scale/performance testing
- At a customer site to diagnose local failures

Each of these scenarios has different requirements for the SUTs. Should they be local for offline testing? Should they be in an internal network for speed and control? Should they be in a cloud provider for mass provisioning? Should there be a combination of local, internal, and cloud SUTs?

Beaker solves this issue by supporting a variety of provisioning techniques from local (VMware Fusion, Vagrant), to internal (VMware vSphere, OpenStack) to cloud (Google Compute Engine, Amazon EC2). The tests are independent of the location that they are executed; the only difference is in the configuration information provided to Beaker at execution time.

4.1 The Beaker Hypervisor Abstraction

The YAML formatted host configuration file provided by the tester also contains important information about where SUTs are located, how they can be accessed, and what configuration steps are necessary for them. In the previous example the hypervisor string is *fusion* for both SUTs.

At runtime Beaker sorts each host into the correct hypervisor bucket based upon the provided hypervisor string. The Hypervisor objects contain all the necessary API calls to provision new SUTs, get them running, configured, and ready for testing. A Beaker user does not need to know the complexities of the APIs of each individual hypervisor - they are responsible for providing some basic hypervisor-specific parameters and Beaker handles the rest.

Beaker code creating Hypervisor objects based upon user provided hypervisor string:

```
            def self.create(type, hosts_to_provision, options)
              hyper_class = case type
                    when /^aix$/
                      Beaker::Aixer
              when /^solaris$/
                      Beaker::Solaris
      <snip> . . . . more buckets here . . . . . . . . . . . </snip>
              when /^openstack$/
                      Beaker::OpenStack
               when /^none$/
                      Beaker::Hypervisor
               else
                      # Custom hypervisor
                      begin
                  require "beaker/hypervisor/#{type}"
                rescue LoadError
                  raise "Invalid hypervisor: #{type}"
                      end
                      Beaker.const_get(type.capitalize)
            end
```

4.2 Build Your Own Hypervisor

Beaker includes a generic Hypervisor object. To add a new hypervisor method simply inherit from the Hypervisor object and override a handful of methods.

A new, barebones hypervisor:

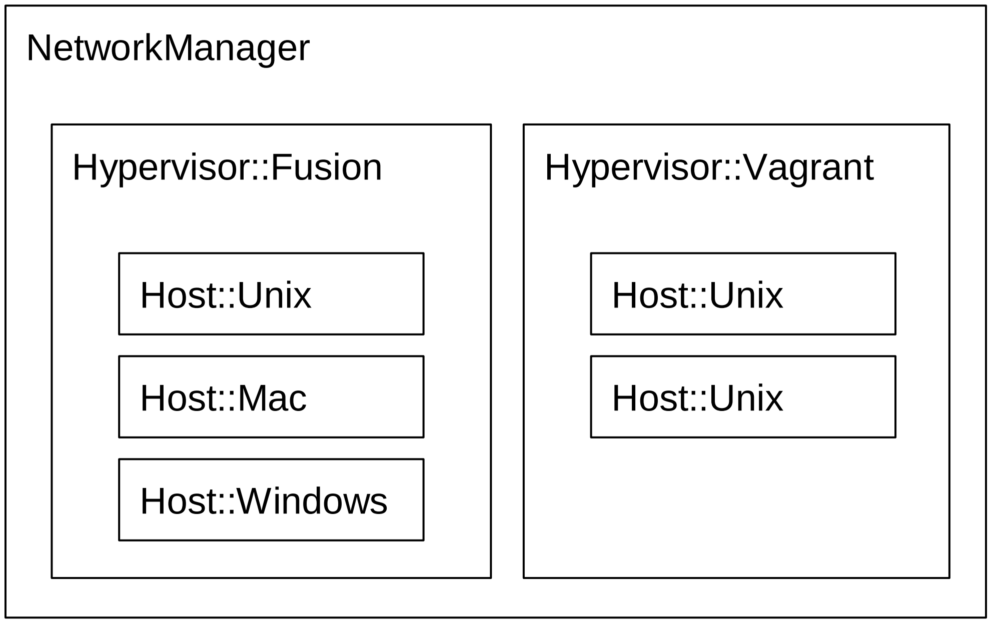
```
      module Beaker
        class NewHypervisor < Beaker::Hypervisor

            def initialize(new_hosts, options)
              @options = options
              @logger = options[:logger]
              @hosts = new_hosts
            end

            def provision
              # create your hosts here, once complete should be
              # able to log in by hostname or vmname or ip
            end

            def cleanup
              # cleanup hosts here
            end
        end
      end
```

The process is simple enough that from an initial set of 7 hypervisors (Vagrant, VMware Fusion, VMware vSphere, Amazon EC2, vCloud, and custom support for managing Solaris and AIX boxes) two years ago, Beaker now supports 16 – including Docker, OpenStack, and Google Compute Engine.

4.3 The NetworkManager Container

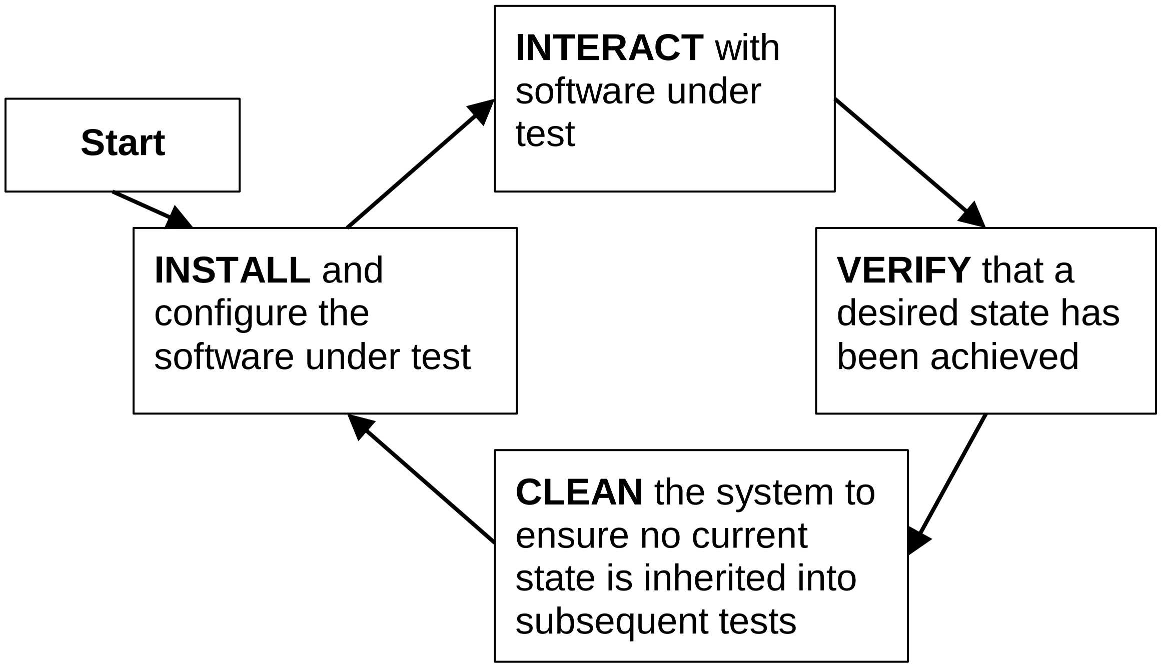
To manage sets of hosts provisioned by different hypervisors, Beaker uses a NetworkManager object which holds the sets of currently active Hypervisors. The Hypervisor objects in turn hold groups of Host Objects. The NetworkManager is responsible for triggering provisioning, configuration, validation, and cleanup on it's hypervisors - which in turn performs those tasks through the appropriate API calls to individual virtualization systems.

An example of the Beaker network management layer:



5 DSL

Acceptance testing follows a consistent pattern:


Start
INSTALL and configure the software under test
INTERACT with software under test
VERIFY that a desired state has been achieved
CLEAN the system to ensure no current state is inherited into subsequent tests

Each stage of testing contains steps and groups of related steps that can be bundled together into shared methods to accomplish useful tasks. The bundles of steps are tested and versioned in Beaker to make up the Beaker DSL (domain specific language). Instead of having to update tests upon change in an installation procedure, command line interface or API, Beaker can be patched, tested, and released with confidence – and existing tests can continue to run and produce useful information.

Consider installing Puppet Enterprise on a series of hosts. Depending on the version of PE, there are significant differences in the installation steps. Within Beaker this is captured in the Beaker DSL method `install_pe`. Many suites of Beaker tests start with a simple pre-suite:

```
test_name "Installing Puppet Enterprise" do
  install_pe
end
```

The `install_pe` method itself is over 600 lines of code and includes logic for

- determining the version of PE to install
- downloading the correct, platform-specific version of PE to each node
- running the installation scripts
- performing setup and configuration of the installed software

The Beaker DSL is made up of over 130 methods that wrap useful tasks. These include general purpose methods to copy files to and from hosts, run scripts on hosts, install packages on hosts, or run arbitrary commands on hosts. Puppet specific DSL methods include installation (FOSS or Enterprise), service restart and module installation.

The Beaker DSL is a powerful tool to make tests easier to write, easier to read, and easier to maintain.

6 Open Source

The Puppet Forge is a repository of modules that are contributed by members of the Puppet community, Puppet Labs, and other software companies. It currently hosts over 3,300 modules for download and use by Puppet users, both commercial and open source. Modules are self-contained bundles of code and data that wrap functionality – such as the puppetlabs-apache module that installs, configures, and manages Apache virtual hosts, web services, and modules.

For modules to be useful for Puppet users they must be dependable, which means they must be tested. Module testing has the same set of challenges as Puppet testing itself.

From the beginning of the Beaker project it was clear that any automated testing solution chosen by Puppet Labs would need to be available for all to use. Beaker's first 0.0.0 release was on August 20, 2013 under the Apache License, Version 2.

7 Adoption

Significant effort was involved in developing a Puppet Labs built acceptance testing tool. To make the choice to build-your-own is always risky – it takes time to design, build, test, and maintain. But, by taking on the challenge, we've been able to build a system that makes test authoring simple. People with minimal scripting experience can write meaningful, cross-platform tests. Those tests can be applied to both the current software and software in the future.

Beaker has been adopted both internally and as a community automated acceptance testing tool. Beaker ensures that Puppet Labs software is dependable and reliable – testing is integrated into the development process from initial planning to successful release. It is used for Puppet testing, both FOSS and Enterprise, module testing, and acceptance testing of additional Puppet Labs projects.

7.1 Beaker Tests, By the Numbers

A simple metric of success is the raw number of tests - it shows that people working on diverse projects are able to write tests and that test writing is simple enough that lots of individual tests have been written. Beaker tests by the numbers:

Puppet Labs Project	**Current # of Beaker Tests**
FOSS Puppet (master branch)	278
Puppet Enterprise (4.0)	441
PuppetDB	28
Hiera	6
Facter	23
25 Puppet Labs Supported Modules (includes concat/inifile/stdlib/etc)	351
56 Puppet Approved Community Modules (includes wget/python/etc)	31
Puppet Labs QA team internal test suites	917
	2075

7.2 Beaker Code, By the Numbers

Beaker has a growing community of developers. The Beaker GitHub repository is a busy place. Beaker currently has 3,535 commits by 92 contributors over 143 forks. A new minor version of Beaker is released every two weeks and can see 3,500 to 5,000 downloads.

8 Conclusions

Acceptance testing at Puppet Labs presents a challenging set of requirements. Any testing system had to provide tests that are:

- Shared across a continually expanding set of operating systems.
- Capable of being executed locally, internally or in the cloud.
- Maximize code reuse through creation of a centralized, versioned set of shared methods.
- Contributable by both employees and community members.

Such a system was not available as an out-of-the-box solution and we undertook the creation of the Beaker acceptance testing tool. With Beaker, a single test, without alteration, can be executed on all supported platforms, on SUTs hosted on a variety of hypervisors, against many versions of Puppet and be run internally by employees or externally by community members.

We've learned that the effort placed in tooling is inversely related to the effort involved in creating a single automated test. The better the tooling, the easier it is to test. Tooling reduces the cost of test writing and lead time for test inclusion in automated suites. Having a single, shared automation tool increases engineering efficiency and lets QA spend time on what they are good at – writing high quality, meaningful tests.

Creating a new tool is a difficult task – getting people to use it is even harder. The success of Beaker is in the simplicity of each individual Beaker test. The easier it is to test, the more likely you are to test and the more reliable your software will be. Make testing easy: write once, run everywhere.