Beaker Administration Guide

Release 29.1

Red Hat, Inc.

February 11, 2024

CHAPTER

ONE

SCOPE OF DOCUMENT

This document provides a general overview of a complete Beaker installation, as well as details on configuring the main Beaker server, individual lab controllers, and configuring Beaker's integration with other systems.

CHAPTER

TWO

AUDIENCE

This guide is primarily aimed at experienced Linux system administrators, tasked with maintaining a Beaker installation, including administering and configuring the main Beaker server or individual lab controllers.

CHAPTER

THREE

CONTENTS

3.1 Beaker architecture

Beaker is an automated software testing application that allows users to store, manage, run and review the results of customized tasks. A task is a script that performs a specific task (or multiple tasks) and presents these task results to the user. These tasks consist of code, data, meta data, shell scripts, hooks to other code, and additional packages (or dependencies). Beaker provides an interactive web application and shell based client to do this.

Although tasks can be written in any language that is supported by the target host, environments such as `rhts-devel` and `beakerlib` are often used. Tasks are written and (ideally) tested before being packaged and uploaded to the server. Once they are uploaded to the server, they are then available to deploy to a target host. The `stdout` of a task executing on a target host is captured and made available via the beaker web application.

3.1.1 Scheduler

The Beaker scheduler co-ordinates the target host systems that will ultimately run the tasks. It manages system selection and the co-ordination of multiple hosts if the job specifications require them. It also manages the schedule of when recipes are run. Currently it operates on a simple FIFO queue. The scheduler does allow some recipes to be prioritized over one another in a non FIFO fashion. This can happen if the recipe matches only one system (priority is bumped by one level), or if the priority is manually set (in which case it can also be set lower).

3.1.2 Server and target host relationship

The server hosts the scheduler, the task repository and the web application. Optionally, the lab controllers and database can also live on the server. The target host makes requests to the server for packages that it needs to create and setup the local test environment, then runs the tasks specified in its recipe. These are run locally on the target host.

The target host runs only one recipe at a time, and before each recipe is run it is re-provisioned with a fresh distro install. As such, the target host reports the results back to the server; It does not store them locally.

3.1.3 Database

The Beaker database is responsible for storing inventory, users and groups, jobs, historical activity, task results and more. It is configured in `/etc/beaker/server.cfg` (or an alternative configuration file if one has been specified) with the `sqlalchemy.dburi` key. Any database type that is supported by sqlalchemy can be used, however the test suite is only run performed against MySQL.

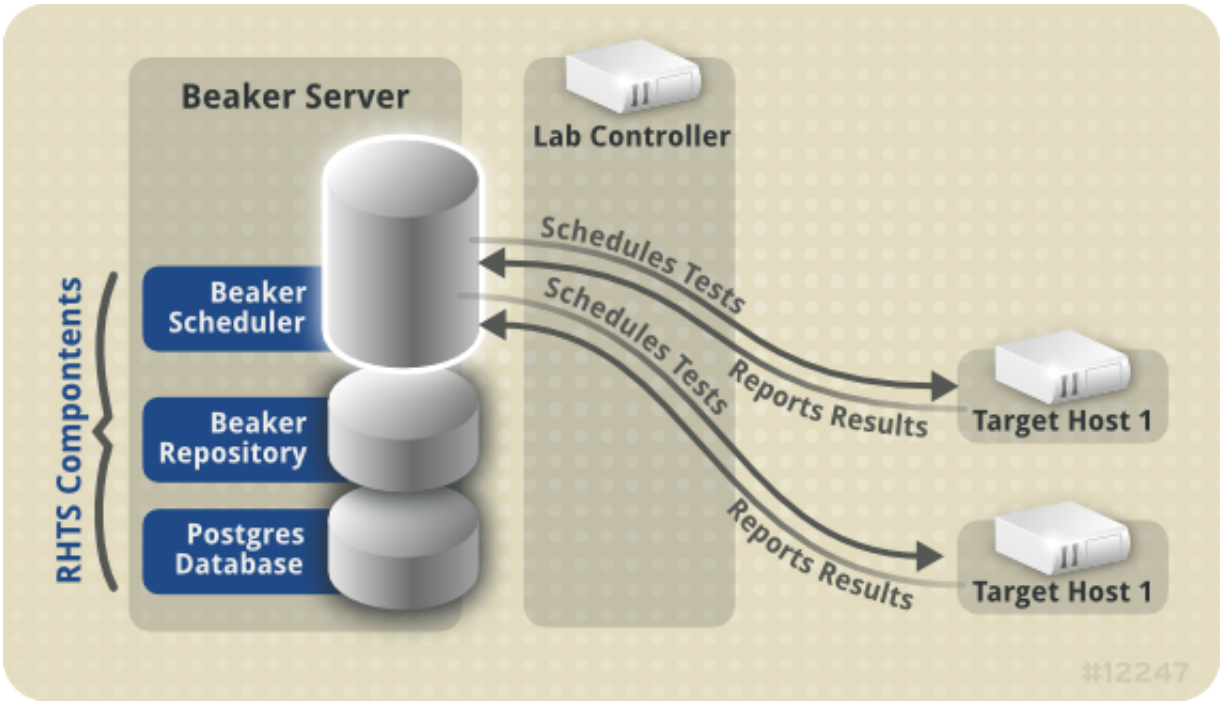

Figure 3.1: Server and target host relationship

3.2 Chronological overview

This section describes how a user creates a script and integrates it into the Beaker environment:

- On a workstation, a user writes a script and then tests the code.
- The user builds an RPM and submits it to the repository (If they are using one).
- They add their task to Beaker's task library.
- They create a job that uses the task.
- The scheduler will then provision a system for the tasks to run on.
- – The task results are sent back to the scheduler for reporting.
- The scheduler uses email to notify the users' of their task results.

3.3 System requirements

There are separate system requirements for the Beaker server and target hosts. Due to the large number of test files that users can store in the database, Beaker requires a multiple terabyte disk storage system.

Your Beaker server should have:

- Red Hat Enterprise Linux 7.8 or higher.
- 200 or more gigabytes hard disk space.
- 4 or more gigabytes of RAM.
- 4 or more CPUs running at 2.5GHz or higher.

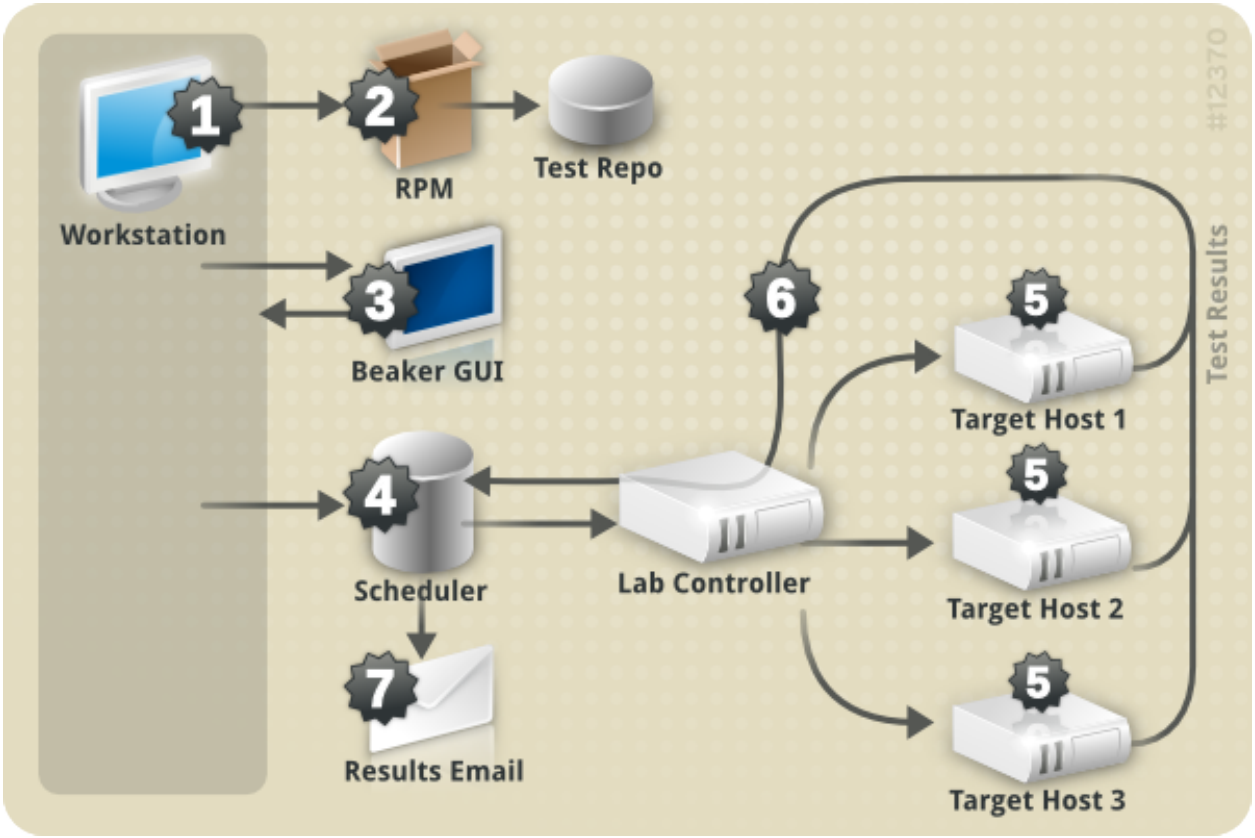

Figure 3.2: Chronological Overview

- 2 or more terabytes tree storage requirement.

Note: If your site already has an existing repository of Red Hat install trees, you do not have to meet the tree storage requirement above.

Your target hosts are required to have:

- Network connectivity to a system running a DHCP server.
- Network booting capability (like PXE or Netboot).
- Serial console logging support using a Target Host's management adapter or a terminal server such as the Avocent ACS series.
- KVM support.

Your target hosts also need to include a power controller. Here are some the most common compatible controllers that are available:

- HP iLO
- Dell DRAC
- WTI Boot bars
- IPMI 1.5 (or higher)
- APC

You may mix and match any of these controllers on the target hosts, but you must include at least one compatible controller per system. Beaker supports the cman package's fence component. Beaker supports any device that you control from the Red Hat Enterprise Linux 7 command line.

3.4 Installation

Pre-built Beaker packages are available from the Download section of Beaker's web site. There are two main repos. One containing packages needed for installing the Beaker server and required components, another for packages needed to run the Beaker client. Download the repo file that suits your requirements and copy it to `/etc/yum.repos.d`.

3.4.1 Installing the Beaker server

Start by installing the `beaker-server` package:

```
$ yum install beaker-server
```

Preparing the database

Beaker uses the SQLAlchemy database library, which supports a large number of databases (including MySQL, PostgreSQL, SQLite, Oracle, and Microsoft SQL Server) provided that a suitable client driver is installed. However Beaker is only tested against MariaDB, so it is recommended to use that.

First, make sure MariaDB server is installed, and configure the daemon to run at startup:

```
$ yum install -y mariadb-server MySQL-python
$ systemctl enable mariadb
```

For Unicode support in Beaker, it is recommended to configure MariaDB to store strings as UTF-8. This is controlled by the `character-set-server` option in `/etc/my.cnf`:

```
[mysqld]
...
character-set-server=utf8
```

Now start the MySQL server:

```
$ systemctl start mariadb
```

Create a database, and grant access to a user. You can put the database on the local machine, or on a remote machine.

```
$ echo "create database <db_name> ;" | mysql
$ echo "create user <user_name> ;" | mysql
$ echo "grant all on <db_name>.* to <user_name> IDENTIFIED BY '<password>';"| mysql
```

Update `/etc/beaker/server.cfg` with the details of the database:

```
sqlalchemy.dburi = "mysql://<user_name>:<password>@<hostname>/<db_name>?charset=utf8"
```

Now let's initialise our DB with tables. We'll also create an admin account called *admin* with password *testing*, and email *root@localhost*.

```
$ beaker-init -u admin -p testing -e root@localhost
```

Starting Beaker

We are now ready to start the Beaker service. It is strongly recommended that the Apache configuration be updated to serve Beaker over HTTPS rather than HTTP.

First make sure Apache is on and configured to run on startup:

```
$ systemctl enable httpd
$ systemctl start httpd
```

We unfortunately need to switch SELinux off on the main Beaker server.

```
$ setenforce 0
```

The appropriate port (80/443 for HTTP/HTTPS) must also be open in the server firewall.

Start the Beaker scheduling daemon and configure it to run on startup.

```
$ systemctl enable beakerd
$ systemctl start beakerd
```

To make sure Beaker is running, open the URL configured in Apache in a browser.

3.4.2 Adding a lab controller

Beaker uses lab controllers to manage the systems in its inventory. The lab controller serves files for network booting, monitors console logs, and executes fence commands to reboot systems.

In small Beaker installations, the lab controller can be the same system as the Beaker server.

External services

Beaker expects DHCP, DNS, and NTP services to be available in the lab, with the appropriate TFTP, DNS and NTP details provided to test systems by the DHCP server.

The TFTP service must run directly on the lab controller to allow Beaker to correctly provision test systems. The DHCP, DNS and NTP services *may* be run on the lab controller, but do not need to be.

A serial console server is also a useful addition to the lab configuration (as it can provide useful diagnostic information for failure, and allows Beaker to monitor the console log for kernel panics), but Beaker will operate correctly without one.

Registering the lab controller

To start with, we need to make Beaker aware of the new lab controller. Log in to Beaker using your administrator account created above, and select Admin → Lab Controllers from the menu. Click "Add (+)" to add a new lab controller.

The new lab controller form requires the following fields:

- *FQDN*: This is the fully qualified domain name of the lab controller.
- *Username*: This is the login name that the lab controller will use to login to beaker. Because this is a machine account we recommend prepending it with host/, so for example host/lab_controller.example.com
- *Password*: This is the password that goes along with the username, again we will use: *testing*
- *Lab Controller Email Address*: All user accounts require a unique email address, you can use root@FQDN of lab controller.

Save the form and we are done with the server side for now.

Configuring the lab controller

Install the lab controller package:

```
$ yum install beaker-lab-controller
```

Settings for the lab controller daemons are in `/etc/beaker/labcontroller.conf`. At a minimum you will need to change the following settings:

- `HUB_URL`: The URL of your Beaker server *without the trailing slash*. If the lab controller and server are the same machine then the default value `https://localhost/bkr` is adequate.
- `USERNAME`, `PASSWORD`: The username and password which the lab controller will use when logging in to Beaker. This is the username and password you picked when registering the lab controller above.

Turn on Apache:

```
$ systemctl enable httpd
$ systemctl start httpd
```

By default, Beaker stores log files for jobs locally on the lab controller and publishes them through Apache. The `beaker-transfer` daemon can be configured to move log files for completed recipes to a separate archive server. The relevant settings to configure this are described in `/etc/beaker/labcontroller.conf`.

Turn on tftp:

```
$ systemctl enable xinetd
$ systemctl enable tftp
$ systemctl start xinetd
```

You can also use dnsmasq or any other TFTP server implementation. If your TFTP server is configured to use a root directory other than the default `/var/lib/tftpboot` you will need to set the `TFTP_ROOT` option in `/etc/beaker/labcontroller.conf`.

The `beaker-proxy` daemon handles XML-RPC requests from within the lab and proxies them to the server.

```
$ systemctl enable beaker-proxy
$ systemctl start beaker-proxy
```

The `beaker-watchdog` daemon monitors systems and aborts their recipes if they panic or exceed the time limit.

```
$ systemctl enable beaker-watchdog
$ systemctl start beaker-watchdog
```

The `beaker-provision` daemon writes netboot configuration files in the TFTP root directory and runs fence commands to reboot systems.

```
$ systemctl enable beaker-provision
$ systemctl start beaker-provision
```

Beaker installs a configuration file into `/etc/sudoers.d` so that beaker-proxy (running as apache) can clear the TFTP netboot files for specific servers (owned by root). To ensure that Beaker lab controllers read this directory, the following command must be enabled in `/etc/sudoers` (it is enabled by default from RHEL 6 forward):

```
#includedir /etc/sudoers.d
```

The appropriate ports (80/443 for HTTP/HTTPS access to log files through Apache, 8000 for test system access to beaker-proxy and 69 for TFTP) must also be open in the lab controller firewall.

3.4.3 Adding the core Beaker tasks

There are a number of standard tasks that are expected to be available in every Beaker installation. You should add these to your Beaker installation before attempting to run jobs.

You can build and upload most of the tasks from source by cloning the beaker-core-tasks git repository, or fetch a pre-built version of the tasks as RPMs from beaker-project.org.

The guest recipe related `/distribution/virt/*` tasks are currently only available as pre-built RPMs.

Copying the tasks from an existing Beaker installation

Alternatively, you can copy *all* the tasks from another Beaker instance using the `beaker-sync-tasks` tool (distributed as a part of the `beaker-server` package and first available with the 0.12 release). For example:

```
$ beaker-sync-tasks --remote=https://server1.com
```

The above command will copy all the tasks, including the standard tasks, from the Beaker instance at `http://server1.com` to the local instance. If there are tasks having the same name in the local Beaker instance, they will be overwritten only if the versions are different.

By default, the script asks for your approval before beginning the task upload. If that is not suitable for your purpose, you may specify a *`--force`* switch so that the script may run without any user intervention. The *`--debug`* switch turns on verbose logging messages on the standard output.

3.4.4 Next steps

You can now proceed to *adding tasks*, *importing distros*, *adding systems*, and *running jobs*.

3.5 Upgrading an existing installation

Before you start, check the `release notes` for any specific instructions beyond the general steps described here. If you are upgrading through multiple releases (for example, from 0.10 to 0.13) follow the instructions for all of the releases.

Always upgrade the Beaker server before the lab controllers, in case the new lab controller version relies on interfaces in the new server version.

3.5.1 Maintenance releases

You can upgrade to a new maintenance release within the same *x.y* series with no interruption to Beaker.

Use Yum to upgrade the relevant packages. The packages will automatically restart any running Beaker daemons, but you should signal the Apache server to reload its configuration.

On the Beaker server:

```
yum upgrade beaker-server
service httpd graceful
```

Then, on the lab controllers:

```
yum upgrade beaker-lab-controller
service httpd graceful
```

3.5.2 Feature releases

When upgrading to a new feature release of Beaker, some database changes may be required. These will be detailed in the release notes.

Database schema changes can interfere with Beaker's normal operation, so you should stop all Beaker services before beginning the upgrade. The sequence of events in this case is:

1. Stop Beaker daemons on the lab controllers.
2. Stop Apache and beakerd on the Beaker server.
3. Use Yum to upgrade all relevant packages.
4. Run **beaker-init** to apply database changes.
5. Start Apache and beakerd on the Beaker server.
6. Extend all watchdogs using **bkr watchdogs-extend**.
7. Start Beaker daemons on the lab controllers.

Note that during the outage period, running jobs will be affected. The harness will be unable to report in to Beaker, so the effects may include missing results and missing logs. Extending all watchdogs at the end of the upgrade will mitigate the problem, by allowing recipes to complete normally if their watchdog time was exceeded during the outage.

3.5.3 New harness packages

New releases of Beaker occasionally include updated versions of `beah`, `rhts`, and other packages which are installed on test systems. The latest versions are published on the Beaker web site.

To update your Beaker server's copy of the harness packages, run the **beaker-repo-update** command. See *beaker-repo-update* for more details.

3.5.4 Downgrading

The procedure for downgrading to an earlier version of Beaker is similar to the upgrade procedures described above, with the following differences:

- Use `yum downgrade` instead of `yum upgrade`, naming the specific package version you are downgrading to.
- If you are downgrading to an earlier release series, run `beaker-init --downgrade=<version>` to downgrade the database schema *before* downgrading any packages. The table below lists the database version corresponding to each Beaker release series.

Beaker release	Database version
29	`140c5eea2836`
28	`4b3a6065eba2`
27	`4cddc14ab090`
26	`348daa35773c`
25	`1ce53a2af0ed`
24	`f18df089261`
23	`2e171e6198e6`
22	`54395adc8646`
21	`171c07fb4970`
20	`19d89d5fbde6`
19	`53942581687f`
0.18	`431e4e2ccbba`
0.17	`431e4e2ccbba`
0.16	`2f38ab976d17`
0.15	`49a4a1e3779a`
0.14	`057b088bfb32`
0.13	`41aa3372239e`
0.12	`442672570b8f`

3.6 Configuration files

The following configuration files are used by Beaker. Each setting in the configuration file has an explanatory comment, and the default value for the setting is shown commented out.

3.6.1 `/etc/beaker/server.cfg`

This is the main configuration file for beakerd and the web application, including database connection settings.

```
[global]
# This defines the URL prefix under which the Beaker web application will be
# served. This must match the prefix used in the Alias and WSGIScriptAlias
# directives in /etc/httpd/conf.d/beaker-server.conf.
```

```
# The default configuration places the application at: http://example.com/bkr/
server.webpath = "/bkr/"

# Database connection URI for Beaker's database, in the form:
#   <driver>://<user>:<password>@<hostname>:<port>/<database>?<options>
# The charset=utf8 option is required for proper Unicode support.
# The pool_recycle setting is required for MySQL, which will (by default)
# terminate idle client connections after 10 hours.
sqlalchemy.dburi = "mysql://beaker:beaker@localhost/beaker?charset=utf8"
sqlalchemy.pool_recycle = 3600

# If you want to send read-only report queries to a separate backup
# database, configure it here. If not configured, report queries will
# fall back to using the main Beaker database (above).
#reports_engine.dburi = "mysql://beaker_ro:beaker_ro@dbbackup/beaker?charset=utf8"
#reports_engine.pool_recycle = 3600

# Set to True to enable sending emails.
#mail.on = False

# TurboMail transport to use. The default 'smtp' sends mails over SMTP to the
# server configured below. Other transports may be available as TurboMail
# extension packages.
#mail.transport = "smtp"
# SMTP server where mails should be sent. By default we assume there is an
# SMTP-capable MTA running on the local host.
#mail.smtp.server = "127.0.0.1"

# The address which will appear as the From: address in emails sent by Beaker.
#beaker_email = "root@localhost.localdomain"

# If this is set to a value greater than zero, Beaker will enforce a limit on
# the number of concurrently running power/provision commands in each lab. Set
# this option if you have a lab with many machines and are concerned about
# a flood of commands overwhelming your lab controller.
#beaker.max_running_commands = 10

# Timeout for authentication tokens. After this many minutes of inactivity
# users will be required to re-authenticate.
#visit.timeout = 360

# Secret key for encrypting authentication tokens. Set this to a very long
# random string and DO NOT disclose it. Changing this value will invalidate all
# existing tokens and force users to re-authenticate.
# If not set, a secret key will be generated and stored in /var/lib/beaker,
# however this configuration impacts performance therefore you should supply
# a secret key here.
#visit.token_secret_key = ""

# Enable LDAP for user account lookup and password authentication.
#identity.ldap.enabled = False
# URI of LDAP directory.
#identity.soldapprovider.uri = "ldaps://ldap.domain.com"
# Base DN for looking up user accounts.
#identity.soldapprovider.basedn = "dc=domain,dc=com"
# If set to True, Beaker user acounts will be automatically created on demand
# if they exist in LDAP. Account attributes are populated from LDAP.
#identity.soldapprovider.autocreate = False
```

```
# Timeout (seconds) for LDAP lookups.
#identity.soldapprovider.timeout = 20
# Server principal and keytab for Kerberos authentication. If using Kerberos
# authentication, this must match the mod_auth_kerb configuration in
# /etc/httpd/conf.d/beaker-server.conf.
#identity.krb_auth_principal = "HTTP/hostname@EXAMPLE.COM"
#identity.krb_auth_keytab = "/etc/krb5.keytab"

# Automatically create user accounts if the user successfully authenticates
# via Apache but there is no matching account in Beaker. The automatic creation
# will only happen if REMOTE_USER_FULLNAME and REMOTE_USER_EMAIL variables are
# also populated in the WSGI environment.
# mod_lookup_identity and mod_auth_mellon can be configured to do this.
#identity.autocreate = True

# These are used when generating absolute URLs (e.g. in e-mails sent by Beaker)
# You should only have to set this if socket.gethostname() returns the wrong
# name, for example if you are using CNAMEs.
#tg.url_domain = "beaker.example.com"
#tg.url_scheme = "http"
# If your scheduler is multi-homed and has a different hostname for your test
# machines you can use the tg.lab_domain variable here to specify it.
# If tg.lab_domain is not set it will fall back to tg.url_domain, and if that's
# not set it will fall back to socket.gethostname().
#tg.lab_domain = "this.hostname.from.lab.domain"

# Tag for distros which are considered "reliable".
# Broken system detection logic will be activated for distros with this tag
# (see the bkr.server.model:System.suspicious_abort method). Leave this unset
# to deactivate broken system detection.
#beaker.reliable_distro_tag = "RELEASED"

# The contents of this file will be displayed to users on every page in Beaker.
# If it exists, it must contain a valid HTML fragment (e.g. <span>...</span>).
#beaker.motd = "/etc/beaker/motd.xml"

# The URL of a page describing your organisation's policies for reserving
# Beaker machines. If configured, a message will appear on the reserve workflow
# page, warning users to adhere to the policy with a hyperlink to this URL. By
# default no message is shown.
#beaker.reservation_policy_url = "http://example.com/reservation-policy"

# These install options are used as global defaults for every provision. They
# can be overriden by options on the distro tree, the system, or the recipe.
#beaker.ks_meta = ""
#beaker.kernel_options = ""
#beaker.kernel_options_post = ""

# When generating MAC addresses for virtual systems, Beaker will always pick
# the lowest free address starting from this base address.
#beaker.base_mac_addr = "52:54:00:00:00:00"

# Beaker increases the priority of recipes when it detects that they match only
# one candidate system. You can disable this behaviour here.
#beaker.priority_bumping_enabled = True

# When generating RPM repos, we can configure what utility to use. The
# createrepo_c implementation is chosen by default: it is faster and more
```

```
# memory-efficient. The original createrepo command can also be used.
#beaker.createrepo_command = "createrepo_c"

# If you have set up a log archive server (with beaker-transfer) and it
# requires HTTP digest authentication for deleting old logs, set the username
# and password here.
#beaker.log_delete_user = "log-delete"
#beaker.log_delete_password = "examplepassword"

# If carbon.address is set, Beaker will send various metrics to carbon
# (collection daemon for Graphite) at the given address. The address must be
# a tuple of (hostname, port).
# The value of carbon.prefix is prepended to all names used by Beaker.
#carbon.address = ('graphite.example.invalid', 2023)
#carbon.prefix = 'beaker.'

# Use OpenStack for running recipes on dynamically created guests.
# Beaker uses the credentials given here to authenticate on OpenStack,
# when creating OpenStack instances on behalf of users.
#openstack.identity_api_url = 'https://openstack.example.com:13000/v3'
#openstack.dashboard_url = 'https://openstack.example.com/dashboard/'
#openstack.username = ""
#openstack.password = ""

# The user domain name when authenticating on OpenStack. If not provided, Beaker
# will not provide a domain name when connecting to OpenStack. This option is
# required if the OpenStack instance has been configured to require a domain name.
#openstack.user_domain_name = ""

# OpenStack external network name for the instance. If not provided, Beaker
# will search for an external network and use the first one it finds.
#openstack.external_network_name = ""

# Beaker will attempt to set up a floating IP address for a newly created
# instance by default. You can disable this behavior here. If set to False,
# the Beaker code will use the IP address assigned when the instance is
# created as the public IP address of the instance.
#openstack.create_floating_ip = True

# Set this to limit the Beaker web application's address space to the given
# size (in bytes). This may be helpful to catch excessive memory consumption by
# Beaker. On large deployments 1500000000 is a reasonable value.
# By default no address space limit is enforced.
#rlimit_as=

# These limits are applied to all running recipes. They are intended as
# a last-resort sanity check, to prevent a runaway task from accidentally
# producing so many results that it can cause problems elsewhere in Beaker (for
# example, excessive memory usage when rendering the results).
# Setting a limit to 0 means the limit will not be enforced.
#beaker.max_results_per_recipe = 7500
#beaker.max_logs_per_recipe = 7500

# OS major names to try (in order of preference) for running inventory jobs on systems.
# The default list includes RHEL, CentOS, and Fedora is suitable in most cases.
# If you have special systems which do not support any of RHEL, CentOS, or Fedora
# then you may need to extend the default list.
#beaker.inventory_osmajors = ['RedHatEnterpriseLinux7', ...]
```

```
# AMQ messaging
# If amq attributes are set, Beaker will send updates via AMQ messages
#amq.url = amqps://broker01.example.com
#amq.cert = /etc/beaker/cert.pem
#amq.key = /etc/beaker/key.pem
#amq.cacerts = /etc/pki/tls/certs/ca-bundle.crt
#amq.topic_prefix = VirtualTopic.eng.beaker
```

3.6.2 /etc/beaker/labcontroller.conf

The main configuration file for the lab controller daemons.

```
# Hub xml-rpc address.
#HUB_URL = "https://localhost:8080"
HUB_URL = "http://localhost/bkr"

# Hub authentication method. Example: krbv, password
AUTH_METHOD = "password"
#AUTH_METHOD = "krbv"

# Username and password
USERNAME = "host/localhost.localdomain"
PASSWORD = "password"

# Kerberos service prefix. Example: host, HTTP
KRB_SERVICE = "HTTP"

# Kerberos realm. If commented, last two parts of domain name are used. Example: MYDOMAIN.COM.
KRB_REALM = "DOMAIN.COM"

# By default, job logs are stored locally on the lab controller.
# If you have set up an archive server to store job logs, uncomment and
# configure the following settings. You will also need to enable the
# beaker-transfer daemon to move logs to the archive server.
#ARCHIVE_SERVER = "http://archive-example.domain.com/beaker"
#ARCHIVE_BASEPATH = "/var/www/html/beaker"
#ARCHIVE_RSYNC = "rsync://USER@HOST/var/www/html/beaker"
#RSYNC_FLAGS = "-ar --password-file /root/rsync-secret.txt"

# How often to renew our session on the server
#RENEW_SESSION_INTERVAL = 300

# Root directory served by the TFTP server. Netboot images and configs will be
# placed here.
TFTP_ROOT = "/var/lib/tftpboot"

# URL scheme used to generate absolute URLs for this lab controller.
# It is used for job logs served by Apache. Set it to 'https' if you have
# configured Apache for SSL and you want logs to be served over SSL.
#URL_SCHEME = "http"

# Fully qualified domain name of *this* system (not the Beaker server).
# Defaults to socket.gethostname(). Ordinarily that is sufficient, unless you
# have registered this lab controller with Beaker under a CNAME.
#URL_DOMAIN = "localhost.invalid"
```

```
# Location where the console logs are stored. Beaker will look in that
# directory for files that start with the Fully Qualified Domain Name (FQDN) of
# the system.
#   For example:
#      If CONSOLE_LOGS=/var/consoles/ and the FQDN=test.example.com
#
#      Then the following files will be logged as console files:
#        /var/consoles/test.example.com -> console.log
#        /var/consoles/test.example.com-bmc -> console-bmc.log
#        /var/consoles/test.example.com-serial2 -> console-serial2.log
#
#CONSOLE_LOGS = "/var/consoles"
```

3.6.3 Other configuration files installed by Beaker

The following configuration files are also installed by Beaker. The defaults provided by these files are suitable for most deployments, but you can tweak these settings if desired.

/etc/httpd/conf.d/beaker-server.conf Apache configuration for serving the web application. You can modify this if you need to adjust authentication or mod_wsgi settings, or if you want to serve the web application at a path other than the default (`/bkr/`).

/etc/httpd/conf.d/beaker-lab-controller.conf Apache configuration for serving job logs cached on the lab controller.

/etc/cron.d/beaker, /etc/cron.hourly/beaker_expire_distros Scheduled jobs which are required for Beaker's operation.

/etc/rsyslog.d/beaker-server.conf, /etc/rsyslog.d/beaker-lab-controller.conf Configuration for rsyslog to send Beaker log messages to the relevant files in `/var/log/beaker`.

/etc/logrotate.d/beaker Configuration for logrotate to rotate log files in `/var/log/beaker`.

/etc/sudoers.d/beaker_proxy_clear_netboot Configuration for sudo to grant beaker-proxy heightened privileges to clear netboot configuration in `/var/lib/tftpboot`.

3.7 TFTP files and directories

As part of the *provisioning process*, test systems fetch boot loader images and configuration files over TFTP from the Beaker lab controller. This section describes all the files under the TFTP root directory that the **beaker-provision** daemon either creates, or relies on indirectly, during the provisioning process.

3.7.1 Boot loader images

These images must be supplied by the Beaker administrator and copied into the TFTP root directory manually (with the exception of `pxelinux.0`). The Cobbler project provides pre-compiled binaries of common boot loaders. Many Linux distributions also package these boot loaders.

When Beaker provisions a system it creates a symlink `bootloader/`*`fqdn`*`/image` pointing to one of these images, depending on the value of the `netbootloader=` kernel option (see *kernel-options*). Alternatively, the DHCP boot filename option can be hard-coded to point at one of these images (see *adding-systems*).

pxelinux.0 Recommended location of the PXELINUX image, used for x86-based systems with BIOS firmware. PXELINUX is a network boot loader developed as part of the Syslinux project.

If this file does not exist, Beaker copies the PXELINUX image from the Syslinux package to this location so that x86 BIOS systems can be provisioned out of the box.

grub/grub.efi Recommended location of the EFI GRUB image, used for x86-based systems with UEFI firmware.

yaboot Location of the Yaboot image, used for PowerPC systems.

elilo-ia64.efi Location of the ELILO image, used for IA64 systems.

aarch64/bootaa64.efi Location of the GRUB2 boot loader image for 64-bit ARM systems.

boot/grub2/powerpc-ieee1275 Location of the GRUB2 boot loader and supporting files for PowerPC (PPC64) systems.

3.7.2 Boot loader configuration directory

New in version 20.

When Beaker provisions a system, it creates a subdirectory `bootloader/`*`fqdn`* under the TFTP root directory containing the following files.

bootloader/*fqdn*/image Symlink to the desired netboot loader image, as specified in the `netbootloader=` kernel option.

bootloader/*fqdn*/etc/*0a010203* Configuration for Yaboot.

bootloader/*fqdn*/grub.cfg-*0A010203* Configuration for GRUB2 (used by 64-bit ARM and PowerPC systems).

bootloader/*fqdn*/grub.cfg Default configuration for GRUB2 (used by 64-bit ARM systems).

bootloader/*fqdn*/petitboot.cfg Configuration for Petitboot.

bootloader/*fqdn*/pxelinux.cfg/*0A010203* Configuration for PXELINUX.

bootloader/*fqdn*/pxelinux.cfg/default Default configuration for PXELINUX.

bootloader/*fqdn*/ipxe/*0a010203* Configuration for iPXE.

bootloader/*fqdn*/ipxe/default Default configuration for iPXE.

3.7.3 Legacy boot loader configuration files

Beaker also creates the following boot loader configuration files for compatibility reasons. These locations will be used when a system's DHCP configuration specifies a hard-coded boot filename instead of using Beaker's configurable netboot loader support.

pxelinux.cfg/*0A010203* Configuration for PXELINUX. The filename is the IPv4 address of the test system, represented as 8 hexadecimal digits (using uppercase letters).

ipxe/*0a010203* Configuration for iPXE. The filename is the IPv4 address of the test system, represented as 8 hexadecimal digits (using lowercase letters).

grub/images Symlink to the `images` directory.

grub/*0A010203* Configuration for EFI GRUB. The filename follows the PXELINUX naming convention.

ppc/*0a010203* Symbolic link to the Yaboot image. The filename is the IPv4 address of the test system, represented as 8 hexadecimal digits (using lowercase letters).

etc/*0a010203* Configuration for Yaboot. The filename matches the boot loader symlink filename.

bootloader/*fqdn*/petitboot.cfg Configuration for petitboot.

ppc/*0a010203*-grub2 Symbolic link to the GRUB2 boot loader. The filename is prefixed with the IPv4 address of the test system, represented as 8 hexadecimal digits (using lowercase letters).

ppc/grub.cfg-*0A1043DE*; boot/grub2/grub.cfg-*0A1043DE*; grub.cfg-*0A1043DE* Configuration for GRUB2 for PowerPC (PPC64) systems. The filename is suffixed with the IPv4 address of the test system, represented as 8 hexadecimal digits (using uppercase letters).

***0A010203*.conf** Configuration for ELILO. The filename follows the PXELINUX naming convention.

arm/empty An empty file.

arm/pxelinux.cfg/*0A010203* Configuration for 32-bit ARM systems. The filename follows the PXELINUX naming convention.

aarch64/grub.cfg-*0A010203* Configuration for 64-bit ARM systems.

s390x/s_*fqdn*; s390x/s_*fqdn*_parm; s390x/s_*fqdn*_conf Configuration files for System/390 virtual machines using "zPXE" (Cobbler's `zpxe.rexx` script).

3.7.4 Other files in the TFTP root directory

images/*fqdn*/ Kernel and initrd images for the distro being provisioned. All the generated boot loader configurations point at the images in this directory.

pxelinux.cfg/default Default configuration used by PXELINUX when no system-specific configuration exists.

The Beaker administrator can customize this configuration, however it must fall back to booting the local disk by default (perhaps after a timeout) using the `localboot 0` command.

If this file does not exist, Beaker populates it with a simple default configuration that immediately boots the local disk.

ipxe/default Default configuration available for chain loading by iPXE when no system-specific configuration exists.

The Beaker administrator can customize this configuration; however, it must fall back to booting from the local disk by default (perhaps after a timeout) using either `exit`, `sanboot`, or whatever works most reliably for the systems involved. Note that if a script is chain loaded, it will return if that script exits, It may be preferable to allow the called script to perform a boot from local disk by following the `chain` command with an `exit`.

If this file does not exist, Beaker populates it with a simple default configuration that immediately boots from the local disk.

aarch64/grub.cfg Default configuration used by 64-bit ARM systems when no system-specific configuration exists.

The Beaker administrator can customize this configuration, however it should exit after a timeout using the `exit` command.

If this file does not exist, Beaker populates it with a simple default configuration that immediately exits.

ppc/grub.cfg Default configuration used by PowerPC systems when no system-specific configuration exists.

The Beaker administrator can customize this configuration, however it should exit after a timeout using the `exit` command.

If this file does not exist, Beaker populates it with a simple default configuration that immediately exits.

pxelinux.cfg/beaker_menu Menu configuration generated by **beaker-pxemenu** for the `menu.c32` program (part of Syslinux). See *Generating a boot menu* for details.

ipxe/beaker_menu Menu configuration generated by **beaker-pxemenu** for the iPXE scripts. Chain load this to get the full beaker install menu.

grub/efidefault Menu configuration generated by **beaker-pxemenu** for EFI GRUB.

aarch64/beaker_menu.cfg Menu configuration generated by **beaker-pxemenu** for 64-bit ARM systems.

boot/grub2/grub.cfg Default configuration for GRUB2 used by x86 EFI systems when no system-specific configuration exists.

The Beaker administrator can customize this configuration, however it should exit after a timeout using the `exit` command.

If this file does not exist, Beaker populates it with a simple default configuration that immediately exits.

boot/grub2/beaker_menu_x86.cfg Menu configuration file generated by **beaker-pxemenu** for EFI GRUB2. Menu contains only x86_64 distros.

boot/grub2/beaker_menu_ppc64.cfg Menu configuration file generated by **beaker-pxemenu** for EFI GRUB2. Menu contains only ppc64 distros.

boot/grub2/beaker_menu_ppc64le.cfg Menu configuration file generated by **beaker-pxemenu** for EFI GRUB2. Menu contains only ppc64le distros.

distrotrees/ Cached images for the generated menus. The contents of this directory are managed by **beaker-pxemenu**.

3.8 Importing distros

In order for a distro to be usable in Beaker, it must be "imported". Importing a distro into Beaker registers the location(s) from which the distro tree is available in the lab, along with various metadata about the distro.

To import a distro, run `beaker-import` on the lab controller and pass all the URLs under which the distro is available. For example:

```
beaker-import \
    http://mymirror.example.com/pub/fedora/linux/releases/17/Fedora/ \
    ftp://mymirror.example.com/pub/fedora/linux/releases/17/Fedora/  \
    nfs://mymirror.example.com:/pub/fedora/linux/releases/17/Fedora/
```

Distros must be imported separately on each lab controller, and you can import from a different set of URLs in each lab. This allows you to import distros from the nearest mirror in each lab.

When importing, at least one of the URLs has to be type of `http`, `https` or `ftp`. Specifying only `nfs` won't work, since it's currently not supported as a valid primary install method.

Normally a distro will have a `.composeinfo` or `.treeinfo` file, which provides metadata required by **beaker-import**. If those files are not available you can perform a "naked" import by specifying `--family`, `--version`, `--name`, `--arch`, `--kernel`, `--initrd`. See *beaker-import* for more details.

You can check that the distros were added successfully by browsing the Distros page (see *distros*).

3.8.1 Fetching harness packages

The first time you import a new distro family you will need to run **beaker-repo-update** on the server to populate the harness repo for the new distro family. See *beaker-repo-update* for more details.

If the distro family is not currently supported by Beaker (for example, if it is a derivative of Fedora or Red Hat Enterprise Linux with a different name) you can instead create a symlink for the harness repo, pointing at an existing compatible distro family:

```
ln -s RedHatEnterpriseLinux6 /var/www/beaker/harness/MyCustomDistro6
```

3.8.2 Install options for distro features

Beaker uses a number of kickstart metadata variables to determine which features are supported by the distro, in order to generate valid kickstarts and scripts when provisioning.

If the distro is supported by Beaker, the distro family will be recognised by name and the correct install options will be automatically populated. In this case you do not need to explicitly set them. Beaker will generate valid kickstarts without any further intervention.

However, if Beaker does not recognize the distro, it will be assumed to have all the latest features (essentially equivalent to the latest Fedora release). If necessary, you can use the *OS versions page* to set install options for the distro family.

For example, if you import a custom distro based on Red Hat Enterprise Linux 6, you should set the following kickstart metadata variables on your custom distro family. This indicates that the distro does not use systemd or chrony, and that the installer does not support `autopart --type` or `bootloader --leavebootorder`.

```
!has_systemd !has_chrony !has_autopart_type !has_leavebootorder
```

Support for Project Atomic distros

To enable *atomic-host*, two install options must be set for Project Atomic distros: `has_rpmostree` and `bootloader_type=extlinux`.

Refer to the documentation about *kickstart metadata* for a complete list of variables relating to distro features.

3.8.3 Automated jobs for new distros

Beaker has a facility for running scripts whenever a new distro is imported, provided by the `beaker-lab-controller-addDistro` package. After installing that package, scripts placed in the `/var/lib/beaker/addDistro.d` directory will be run each time a distro is imported.

Beaker ships with a script, `/var/lib/beaker/addDistro.d/updateDistro`, which schedules a Beaker job to test installation of the new distro and tags it with `STABLE` if the job completes without error. Use this as a guide for creating more specific jobs that you might find useful.

Note: The `updateDistro` script assumes that the Beaker client is correctly configured on the lab controller. See *installing-bkr-client*.

3.8.4 Generating a boot menu

Beaker includes a command, `beaker-pxemenu`, which can be run on the lab controller to generate a boot menu containing the distros in Beaker. Users in the lab can then perform manual installations by selecting a distro from the menu. Boot menus are generated for `menu.c32` (PXELINUX), EFI GRUB, 64-bit ARM and 64-bit PowerPC.

You can limit the menu to only contain distros tagged in Beaker with a certain tag, by passing the `--tag` option to `beaker-pxemenu`. By default, all distros which are available in the lab are included in the menu.

Note: If you have configured a non-default TFTP root directory in `/etc/beaker/labcontroller.conf`, be sure to pass that same directory in the `--tftp-root` option to `beaker-pxemenu`.

If you are using a boot menu, you should edit the PXELINUX default configuration `pxelinux.cfg/default` to boot from local disk by default, with an option to load the menu. For example:

```
default local
prompt 1
timeout 200

say ************************************************
say Press ENTER to boot from local disk
say Type "menu" at boot prompt to view install menu
say ************************************************

label local
    localboot 0

label menu
    kernel menu.c32
    append pxelinux.cfg/beaker_menu
```

Similarly, you should edit the default configuration for 64-bit ARM `aarch64/grub.cfg` to exit after a timeout, with an option to load the menu. For example:

```
set default="Exit PXE"
set timeout=10
menuentry "Exit PXE" {
    exit
}
menuentry "Install distro from Beaker" {
    configfile aarch64/beaker_menu.cfg
}
```

If you are using GRUB2 boot menus, you should edit the default configuration for x86 EFI `boot/grub2/grub.cfg` to exit after a timeout, with an option to load the menus. For example:

```
set default="Exit PXE"
set timeout=60
menuentry "Exit PXE" {
    exit
}
menuentry "Install distro from Beaker (x86)" {
    configfile boot/grub2/beaker_menu_x86.cfg
}
```

Likewise, you should edit the default configuration for iPXE :file:'ipxe/default‘ to exit after a timeout with an option to load the menu. Also, iPXE will not load the host specific script by default like PXELINUX, so we direct it to do that in the default script if available, for example:

```
#!ipxe

chain /ipxe/${ip:hexraw} || prompt –key m –timeout 60000 Press ‘m’ to view install menu, any other key
to boot from local disk && set beaker 1 || clear beaker isset ${beaker} && chain /ipxe/beaker_menu ||
iseq ${builtin/platform} pcbios && sanboot –no-describe –drive 0x80 || exit 1
```

If your site imports distros into Beaker infrequently, you may prefer to run `beaker-pxemenu` after importing new distros. Otherwise, you can create a cron job to periodically update the PXE menu:

```
#!/bin/sh
exec beaker-pxemenu --quiet
```

3.9 Beaker interface for administrators

Some functionality in Beaker's web interface is restricted to administrators. Most of this functionality is accessed from the *Admin* menu.

3.9.1 Groups

In addition to the groups functionality available to all users (see `../../user-guide/interface/groups`), administrators have access to certain extra features.

If LDAP is configured for identity management (`identity.ldap.enabled` in the server configuration file), you can flag a group's membership to be populated from LDAP. If this flag is set, Beaker will not allow users to be added or removed from the group manually. Instead, a cron job runs the `beaker-refresh-ldap` command periodically to refresh group membership from LDAP. Administrators with command line access to the main Beaker server may also run `beaker-refresh-ldap` directly to force an immediate update from the LDAP server.

You can also grant special system-wide permissions to groups. These permissions would typically only be granted to special groups for service accounts or privileged users. The following permissions are defined:

tag_distro Permitted to tag and untag distros.

distro_expire Permitted to remove a distro's association with a lab controller.

secret_visible Permitted to view all systems, including other users' systems which are marked as "Secret".

stop_task Permitted to stop, cancel, or abort jobs or recipe sets owned by any user.

change_prio Permitted to increase or decrease the priority of any user's job.

3.9.2 OS versions

The "OS Versions" page shows a list of the major and minor versions of every distro that has been imported into Beaker. Select *Admin* → *OS Versions* from the menu.

To edit a particular OS major version, click its name in the first column. From this page you can edit the following details:

Alias If set, the alias can be used to refer to this OS major version in the `Releases` field of task metadata (see *testinfo-releases*). This is intended mainly as a compatibility mechanism for older tasks which use an obsolete name in the `Releases` field (for example `RHEL3` instead of `RedHatEnterpriseLinux3`).

Note: If you set an alias for an existing OS major version, you cannot import distros under the aliased name. For example, if you set "RHEL6" as an alias for "RedHatEnterpriseLinux6", then attempts to import a new distro whose OS major version is "RHEL6" will fail with the following error message:

```
Cannot import distro as RHEL6: it is configured as an alias for RedHatEnterpriseLinux6
```

To fix the problem, either unset the alias or correct the OS major version in the distro tree you are trying to import.

Install Options These are the default install options when provisioning a distro from this major version. Options may be set for all arches or for each arch individually. Options at this level are overridden by any options set at the distro tree level. See *install-options* for details about the meaning of these options.

3.9.3 Configuration

Some Beaker configuration is stored in the database rather than in the server configuration file, and can be changed without restarting Beaker services. Select *Admin* → *Configuration* from the menu to view and change settings.

3.9.4 Export

The *Admin* → *Export* menu item allows an administrator to export data about the systems and user groups as CSV files. Currently, the following data can be exported:

Systems For every system, its FQDN, its deletion and secret status, lender, location, MAC address, memory, model, serial, vendor, supported architectures, lab controller, owner, status, type and cc fields are exported.

Systems (for modification) In addition to the above fields, this also exports the database identifier for each system. This is useful when you want to rename existing systems (see *Import*).

System LabInfo For every system, the original cost, current cost, dimensions, weight, wattage and cooling data about its lab is exported. If there is no such data available for this system, the corresponding system entry is not exported.

System Power For every system, the power address, username and password, power id and power type are exported.

System Excluded Families The data for systems which are excluded from running jobs requiring certain families of operating systems are exported. The fields exported are the FQDN of the system and the details about the operating system (architecture, family and the update) which is excluded.

System Install Options The data for the systems with custom install options are exported. The fields exported are the FQDN of the system, architecture, the operating system family (and update) and the corresponding install options: ks-meta, kernel options and post kernel options.

System Key/Values For every system, its key value pairs are exported.

System Pools Systems which belong to a system pool are exported along with the corresponding pool names.

User Groups The users and the groups which they are a member of are exported.

3.9.5 Import

The *Admin* → *Import* option is useful for two workflows:

1. Administrator exports the data from a Beaker instance (see *Export*), makes some changes and uploads the modified file to the same Beaker instance.
2. Administrator exports the data from a Beaker instance (see *Export*) and uses it to setup a new Beaker instance (with or without making any changes to the exported data).

The first workflow updates the data about one or more existing systems or users. For the data related to the systems, the system FQDN is used to look up the system in Beaker's database. If however, a system is to be renamed, then the "Systems (for modification)" data should be used since it also exports the database identifier for the system (the corresponding field name is "id") which is then used to look up the system in Beaker's database.

The second workflow is useful when the same set of systems or user groups should be present in a different Beaker instance. In this case, the data exported by "Systems (for modification)" should *not* be used since data about the existing systems may be accidentally overwritten.

Note: The CSV file that can be successfully imported by Beaker must conform to the following guidelines:

- The fields are delimited by commas.
- The values should be quoted with double quotes (for example, `"Rack 1, Lab 2"`).
- Quotes are escaped by doubling them (for example, `"Rack ""A"", Lab 2"`).

3.10 Customizing kickstarts

When Beaker provisions a system, the Beaker server generates an Anaconda kickstart from a template file. Beaker's kickstart templates are written in the Jinja2 templating language. Refer to the Jinja2 documentation for details of the template syntax and built-in constructs which are available to all templates.

Beaker selects a base kickstart template according to the major version of the distro being provisioned, for example `Fedora16` or `RedHatEnterpriseLinux6`. If no template is found under this name, Beaker will also try the major version with trailing digits stripped (`Fedora`, `RedHatEnterpriseLinux`).

The Beaker server searches the following directories for kickstart templates, in order:

- `/etc/beaker/kickstarts`

  Custom templates may be placed here.

- `/usr/lib/python2.7/site-packages/bkr/server/kickstarts`

  These templates are packaged with Beaker and should not be modified.

Beaker ships with kickstart templates for all current Fedora and Red Hat Enterprise Linux distros. The shipped templates include all the necessary parts to run Beaker scheduled jobs. They also provide a mechanism for customizing the generated kickstarts with template "snippets". Administrators are recommended to use custom snippets where necessary, rather than customizing the base templates.

Administrators can test the generated kickstart for a particular system or an already completed recipe using the **beaker-create-kickstart** server-side command.

3.10.1 Kickstart snippets

Each snippet provides a small unit of functionality within the kickstart. The name and purpose of all defined snippets are given below, along with details of how administrators can selectively override them for particular distros, labs and systems.

The following snippets provide required functionality for Beaker integration, and should never be overridden by the administrator. However, they may be useful when implementing custom templates and snippets:

`rhts_pre`; `rhts_post` Scripts necessary for running a Beaker recipe on the system after it is provisioned. Most of the other snippets listed below are included from one of these snippets rather than directly from the kickstart templates.

`beah`; `harness` Handles installation of the default Beaker test harness (`beah`), or installation of an alternative harness.

`clear_netboot` Instructs the lab controller to remove the current netboot configuration for this system.

`grubport` Updates the GRUB configuration if the `grubport` kickstart metadata variable is set.

`install_done` Checks in with the lab controller to indicate that installation is complete (post-installation scripts are now proceeding) and to report the FQDN of the newly installed system.

linkdelay Handles the `linkdelay` ks_meta variable.

password Handles setting the root password for the system.

postinstall_done Checks in with the lab controller to indicate that post-installation scripts are complete. This snippet defines its own %post section and is placed after all other %post sections.

pre_anamon; post_anamon Configures anamon, a small daemon which runs during the Anaconda install process and uploads log files to the Beaker scheduler.

rhts_packages Provides a list of packages to be installed by Anaconda, based on the packages required by and requested in the recipe.

rhts_partitions Defines the partitions to be created by Anaconda, based on the partition configuration requested in the recipe.

For the following snippets Beaker ships a default template, which should be sufficient in most cases. However, administrators may choose to override these if necessary.

boot_order On EFI systems, Anaconda adds a new default boot entry that boots from the local disk rather than the network. To ensure Beaker can reprovision the system later, Beaker removes this entry and sets the "BootNext" setting instead. The `rhts-reboot` command also sets "BootNext" to the entry added by Anaconda rather than booting from the network. See *boot-order-details* for more information.

New in version 0.14.4: The boot order adjustment was moved to its own snippet, allowing it to be overridden without replacing the entirety of rhts_post.

install_method Provides the `url` or `nfs` kickstart command which tells Anaconda where to find the distro tree for installation.

lab_env Sets environment variables on the installed system, giving the address of various services within the lab. The exact name and meaning of the environment variables are left up to the administrator, but may include for example build servers, download servers, or temporary storage servers.

post_s390_reboot Reportedly this does not work and should probably be deleted.

print_anaconda_repos Provides the `repo` kickstart commands which tell Anaconda where to find the distro tree's Yum repositories for installation. This includes any custom repos passed in the job XML as well, e.g. `<repo name="repo_id" url="http://server/path/to/repo"/>`

print_repos Sets up the system's Yum repo configuration after install.

readahead_sysconfig Disables readahead, which is known to conflict with auditd in RHEL6.

rhts_devices; rhts_scsi_ethdevices Provides `device` commands (if necessary) which tell Anaconda to load additional device modules.

ssh_keys Adds the Beaker user's SSH public keys to `/root/.ssh/authorized_keys` after installation, so that they can log in using SSH key authentication.

timezone Provides the `timezone` kickstart command. The default timezone is "America/New_York". Administrators may wish to customize this on a per-lab basis to match the local timezone of the lab

The following snippets have no default template, and will be empty unless customized by the administrator:

network Provides extra network configuration parameters for Anaconda.

packages Can be used to append extra packages to the `%packages` section of the kickstart.

system; <distro_major_version> Can be used to insert extra Anaconda commands into the main section of the kickstart.

system_pre; <distro_major_version>_pre Can be used to insert extra shell commands into the %pre section of the kickstart.

system_post; <distro_major_version>_post Can be used to insert extra shell commands into the %post section of the kickstart.

Overriding kickstart snippets

When a snippet is included in a kickstart template, Beaker tries to load the snippet from the following locations on the server's filesystem, in order:

- `/etc/beaker/snippets/per_system/`*`snippet_name`*`/`*`system_fqdn`*
- `/etc/beaker/snippets/per_lab/`*`snippet_name`*`/`*`labcontroller_fqdn`*
- `/etc/beaker/snippets/per_osversion/`*`snippet_name`*`/`*`distro_version`*
- `/etc/beaker/snippets/per_osmajor/`*`snippet_name`*`/`*`distro_major_version`*
- `/etc/beaker/snippets/`*`snippet_name`*
- `/usr/lib/python2.7/site-packages/bkr/server/snippets/`*`snippet_name`*

This allows administrators to customize Beaker kickstarts at whatever level is necessary.

For example, if the system host01.example.com needs to use a network interface other than the default, the following snippet could be placed in `/etc/beaker/snippets/per_system/network/host01.example.com`:

```
network --device eth1 --bootproto dhcp --onboot yes
```

3.10.2 Writing kickstart templates

Kickstart templates and snippets are rendered using the same mechanism as custom kickstart templates from a recipe. Refer to *custom-kickstarts*.

Some extra variables are also available to system templates (that is, templates loaded from disk rather than submitted by users):

config
The Beaker configuration. Call `config.get(name, default=None)` to look up values in Beaker's application-wide configuration.

recipe
The recipe which is being provisioned. If the kickstart is for a system which is being manually provisioned (using the *Provision* tab on the system page) then this variable will be None.

This object has the following attributes:

id Database identifier of the recipe.

system
The system which is being provisioned. This object has the following attributes:

fqdn The fully-qualified domain name of the system.

has_efi True if the system has EFI firmware. Only valid for x86.

owner A user object representing the system owner.

user
A user object representing the job owner. This object has the following attributes:

display_name Full display name of the job owner.

email_address Email address of the job owner.

user_name Username of the job owner.

3.11 Customizing power commands

When executing a power command (for example, rebooting a system) the Beaker lab controller looks for an executable script named after the system's power type (for example, `ipmitool`). The following directories are searched on the lab controller, in order:

- `/etc/beaker/power-scripts`

  Custom power scripts may be placed here.

- `/usr/lib/python2.7/site-packages/bkr/labcontroller/power-scripts`

  These templates are packaged with Beaker and should not be modified.

When a script is found it is executed with the following environment variables set according to the system's power settings in Beaker:

- *power_address*
- *power_id*
- *power_user*
- *power_pass*

Additionally, the power_mode environment variable will be set to either `on` or `off`, depending on the power action.

3.12 Customizing panic detection and install failure detection

The **beaker-watchdog** daemon on the lab controller scans console logs to detect if a recipe has triggered a kernel panic, or if the installation has failed with a fatal error.

You can customize the regexp pattern for detecting kernel panics by setting `PANIC_REGEX` in `/etc/beaker/labcontroller.conf`. The default pattern that ships with Beaker is defined in `/usr/lib/python2.7/site-packages/bkr/labcontroller/default.conf`. The pattern uses *Python regular expression syntax*.

Install failure patterns are read from a directory (rather than using a single pattern like the panic detection). Each file contains a regexp pattern which is checked against the console log. If any pattern matches, the installation is considered to have failed.

Beaker ships a number of default patterns in the `/usr/lib/python2.7/site-packages/bkr/labcontroller/install-` directory. You can define extra custom patterns by placing them in `/etc/beaker/install-failure-patterns/`. A custom pattern overrides a default pattern with the same filename. If a pattern is empty, Beaker ignores it. If you want to disable a default pattern, create an empty file with the same name in `/etc/beaker/install-failure-patterns/`.

3.13 Customizing expired watchdog handling

When the watchdog timer for a recipe expires, by default the **beaker-watchdog** daemon aborts the recipe.

You can supply a custom script to handle watchdog expiry by setting `WATCHDOG_SCRIPT` in `/etc/beaker/labcontroller.conf`. If this option is set, **beaker-watchdog** invokes the named script to handle the watchdog expiry instead.

The watchdog script is executed with three arguments: the system FQDN, the recipe ID, and the currently running task ID. If the script wants to handle the watchdog expiry (for example, by triggering a crash dump to network storage) it should print to stdout the number of seconds to extend the watchdog timer by, and then exit with zero status. A non-zero exit status from the script is interpreted as failure and **beaker-watchdog** will abort the recipe as usual in that case.

Note that if the script requests an extension to the watchdog, it will be invoked again if the recipe is still not finished when the newly extended watchdog time is next reached. Therefore, to avoid infinite watchdog extension, the script must either take care to avoid handling the same recipe ID multiple times, or it must abort the recipe using some external mechanism after exiting.

3.14 Theming the web interface

You can apply custom styles and markup to Beaker's web interface, for example to make it match a common look-and-feel for your organization, or to add extra links to site-specific resources.

3.14.1 Adding custom Less rules

Beaker uses the Less preprocessor to generate CSS. You can define a site-specific Less source file, in order to add extra styles or to re-define variables used by Beaker's styles. Create a new `.less` file in `/usr/share/bkr/server/assets` containing your custom rules, and then update the `site.less` symlink to point to your Less source file. By default this symlink points to `/dev/null`.

Note: If you update the `site.less` symlink to point at a file with a modification time older than Beaker's other `.less` source files, Beaker will not re-generate its CSS. This is due to the asset caching which is based on modification times. In this case you can force the CSS to be re-generated by removing `/var/cache/beaker/assets/*.css` and reloading Apache.

For example, to add a custom background image to Beaker's footer:

```
footer {
    padding-left: 120px;
    background-image: url('my-logo.svg');
    background-repeat: no-repeat;
    background-size: 110px auto;
}
```

For more information about Less language syntax, refer to the Less language documentation.

3.14.2 Injecting custom markup

To inject markup into every page of Beaker's web interface, configure a custom site template in `/etc/beaker/server.cfg`:

```
tg.sitetemplate = "my_package.my_sitetemplate"
```

The site template must be a Kid template, available on the system Python path (for the above example it would be `/usr/lib/python2.x/site-packages/my_package/my_sitetemplate.kid`). The template should contain a root element as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://purl.org/kid/ns#">
</html>
```

Inside the root element, define one or more `py:match` directives to match and replace markup. For example, to add an extra link to Beaker's *Help* menu:

```
<ul py:match="item.get('id') == 'help-menu'" py:attrs="item.items()">
    <li py:replace="item[:]"/>
    <li><a href="https://example.com/beaker">Internal Docs</a></li>
</ul>
```

For more information about Kid template syntax, refer to the Kid documentation.

3.15 Integration with OpenStack

Note: The OpenStack integration for dynamic system provisioning is classified as experimental. It has a number of limitations and may impact the scheduler's performance, therefore it is currently not recommended for use in production on large Beaker instances.

Beaker can optionally be configured to use OpenStack to create virtual machines on demand for running recipes.

When OpenStack integration is enabled, Beaker will attempt to create a new virtual machine for each recipe when it is scheduled. If creating the virtual machine fails, Beaker will fall back to using the regular hardware pool for that recipe. Recipes with hardware requirements in `<hostRequires/>` which cannot be satisfied by a virtual machine are excluded from this process.

3.15.1 Package prerequisites

- python-keystoneclient >= 3.10.0
- python-novaclient >= 7.1.2
- python-glanceclient >= 2.6.0
- python-neutronclient >= 6.1.1

As RHEL 7 may not provide the required version of those packages, you can use Ocata EL7 repositories provided by CentOS in this case. See CentOS OpenStack Wiki for more details.

3.15.2 Configuring OpenStack integration

To enable OpenStack integration, configure the Identity API (Keystone) endpoint, dashboard (Horizon) URL and an OpenStack account of Beaker in `/etc/beaker/server.cfg`:

```
# Use OpenStack for running recipes on dynamically created guests.
# Beaker uses the credentials given here to authenticate to OpenStack
# when creating OpenStack instances on behalf of users.
openstack.identity_api_url = 'https://openstack.example.com:13000/v3'
openstack.dashboard_url = 'https://openstack.example.com/dashboard/'
openstack.username = ""
openstack.password = ""
```

The user domain name when authenticating to OpenStack. Beaker does not provide a default domain name. This option is required if the OpenStack instance has been configured to require a domain name.

openstack.user_domain_name = “”

The OpenStack external network name for the instance. If not provided, Beaker will search for an external network and use the first one it finds.

openstack.external_network_name = “”

By default, Beaker will attempt to set up a floating IP address for a newly created instance to provide a public IP address. This assumes that the IP address assigned when the instance is created is on a private network. If the ‘create_floating_ip’ flag is set to False, the Beaker code will use the IP address assigned when the instance is created as the public IP address of the instance.

openstack.create_floating_ip = True

Currently Beaker does not support multiple OpenStack regions. Beaker expects a single row to exist in the `openstack_region` table, referencing the lab controller which should be used for OpenStack recipes. You must insert the row manually:

```
INSERT INTO openstack_region (lab_controller_id)
SELECT id FROM lab_controller WHERE fqdn = 'lab.example.com';
```

3.15.3 Uploading iPXE image to Glance

In order to boot distro installers on OpenStack instances, Beaker relies on a special image containing the iPXE network boot loader, which then loads its boot configuration from the Beaker server. The **beaker-create-ipxe-image** tool creates and uploads a suitable image to Glance. You must run this tool once after defining an OpenStack region.

The name for each virtual machine is constructed from the `guest_name_prefix` setting (see *Configuration*) combined with the recipe ID. If you have configured multiple Beaker instances to use the same OpenStack instance, make sure you set a distinct value for `guest_name_prefix` to avoid name collisions.

3.16 Integration with Graphite

Beaker can optionally be configured to send metrics to the Graphite real-time graphing system. Beaker sends metrics via UDP for efficiency, and to avoid impacting the performance and reliability of the system, so a version of Graphite with UDP listener support is required.

To enable Graphite integration, configure the hostname and port of the carbon daemon in `/etc/beaker/server.cfg`:

```
carbon.address = ('graphite.example.invalid', 2023)
carbon.prefix = 'beaker.'
```

The `carbon.prefix` setting is a prefix applied to the name of all metrics Beaker sends to Graphite. You can adjust the prefix to fit in with your site’s convention for Graphite metric names, or to distinguish multiple Beaker environments sharing a single Graphite instance.

3.16.1 Aggregating metrics

Beaker does not perform aggregation of metrics, and expects to send metrics to Graphite’s carbon-aggregator daemon (which forwards the metrics to carbon-cache for storage after aggregating them). The `carbon.address` setting should therefore be the address of the carbon-aggregator daemon.

Beaker may send three types of metrics: counters, gauges, and durations. (A duration is equivalent to a gauge except that it is in seconds instead of arbitrary units.) The type appears at the start of the metric name, after the configured prefix. For example, assuming the default prefix `beaker.`, Beaker will periodically report the number of running recipes as `beaker.gauges.recipes_running`.

You should configure suitable aggregation rules for Beaker in `/etc/carbon/aggregation-rules.conf`. The following example assumes the default prefix `beaker.` and 1-minute storage resolution:

```
beaker.durations.<name> (60) = avg beaker.durations.<name>
beaker.counters.<name> (60) = sum beaker.counters.<name>
beaker.gauges.<name> (60) = avg beaker.gauges.<name>
```

3.16.2 System utilization metrics

To provide a real-time view of system utilization, Beaker updates the following gauges:

```
beaker.gauges.systems_idle_automated
beaker.gauges.systems_idle_broken
beaker.gauges.systems_idle_manual
beaker.gauges.systems_manual
beaker.gauges.systems_recipe
```

These metrics describe the current utilization of Automated, Manual and Broken systems in Beaker.

Automated systems are under the control of the Beaker scheduler, and are available to run submitted jobs. They are covered by the `recipe` (currently waiting for other recipes in a recipe set, provisioning the system or running a task as part of a recipe) and `idle_automated` (waiting to be assigned to a recipe) gauges.

Manual systems are available to Beaker users, but not to the scheduler. They are covered by the `manual` (assigned to a specific user) and `idle_manual` (not assigned to anyone) gauges.

Broken systems, covered by the `idle_broken` gauge, are awaiting investigation by system administrators before being placed back in the pool of available systems.

In addition to the metrics for every system known to Beaker, live metrics are also available for systems in the shared pool, which are equally available to all users of a Beaker installation. To access these metrics, replace `.all` with `.shared`.

Each of the system utilization gauges is also available broken down by architecture and by the lab controller that manages that system. For example, information on the idle x86_64 machines can be accessed as:

```
beaker.gauges.systems_idle_automated.by_arch.x86_64
```

As a system may support multiple architectures (e.g. both "i386" and "x86_64", the `by_arch` metrics may not add up to the `all` metrics).

Information on the machines managed by a particular lab controller can be accessed as:

```
beaker.gauges.systems_idle_automated.by_lab.lchost_example_com
```

3.16.3 Recipe queue metrics

To provide a real-time view of the recipe queue, Beaker updates the following gauges:

```
beaker.gauges.recipes_new.all
beaker.gauges.recipes_processed.all
beaker.gauges.recipes_queued.all
beaker.gauges.recipes_scheduled.all
beaker.gauges.recipes_running.all
beaker.gauges.recipes_waiting.all
```

The `new` and `processed` states are transient states used when a job is initially submitted to Beaker. All recipes should move relatively quickly through these states to the `queued` state. If this isn't happening, it is a sign that new jobs are arriving faster than the scheduler is able to process them.

The `queued` state indicates that initial processing of the recipe is complete, and it is ready to be assigned to a system. Depending on the strictness of the recipe's host requirements, and the availability of suitable systems, recipes may spend an extended period of time in this state.

The `scheduled` state indicates that the recipe has been assigned a system (or a virtualized resource), but is waiting for other recipes in the same recipe set to be assigned a resource.

The `waiting` state indicates that the recipe is waiting for the initial reboot of the system that starts the kickstart-based provisioning process. Recipes should move relatively quickly through this state to the `running` state. If this isn't happening, it is a sign that there is a problem somewhere in the Beaker installation (e.g. if the `beaker-provision` service is not running on one of the lab controllers, recipes assigned to that lab will get stuck in this state).

The `running` state indicates that the recipe is either waiting for the provisioning to complete, or is actually executing tasks on the assigned resource.

The number of recipes in `scheduled` and `running` may exceed the number of systems assigned to a recipe (as indicated by the `systems_recipe` gauge) as recipes may be executing on a dynamically created virtual machine.

To observe the utilization of dynamic virtualization resources, replace `.all` with `.dynamic_virt_possible`. These metrics show recipes which are either still under consideration for creation of a dynamic virtual machine, or which have already been assigned one.

Each of the recipe queue gauges is also available broken down by the architecture of the distro tree associated with the recipe. For example, information on the recipes currently in Beaker that require x86_64 machines can be accessed as:

```
beaker.gauges.recipes_queued.by_arch.x86_64
```

3.16.4 Dirty job count

Beaker populates this gauge with the number of jobs currently marked "dirty" in the database:

```
beaker.gauges.dirty_jobs
```

Jobs become "dirty" when their scheduling state has been changed (for example, the user cancels the job, or the harness completes a task) but the scheduler has not yet handled the status update.

A large value for this gauge indicates that there may be a problem with the scheduler causing a backlog of unhandled status updates.

3.16.5 System command metrics

Similar to the recipe queue metrics described above, Beaker provides a real-time view of the system command queue with the following gauges:

```
beaker.gauges.system_commands_queued.all
beaker.gauges.system_commands_running.all
```

The `queued` state represents commands which are in the queue but the **beaker-provision** daemon has not started running them yet. The `running` state represents commands which have started but not finished yet.

A large value for the `queued` gauge indicates that there may be a problem with the **beaker-provision** daemon on a lab controller causing a backlog of queued commands.

In addition, Beaker updates the following counters when a system command has finished (whether successfully or not):

```
beaker.counters.system_commands_completed.all
beaker.counters.system_commands_aborted.all
beaker.counters.system_commands_failed.all
```

Each of the command queue gauges and counters is also available broken down by the lab controller responsible for running the command.

3.16.6 Useful graphs

Below are some links to useful graphs showing the overall health and performance of your Beaker system. These URLs could be used as the basis for a dashboard or given to users. The URLs assume the default metric name prefix `beaker.` with a Graphite instance at `graphite.example.com`.

Utilization of all systems

```
http://graphite.example.com/render/?width=1024&height=400
    &areaMode=stacked
    &target=beaker.gauges.systems_idle_automated.all
    &target=beaker.gauges.systems_idle_broken.all
    &target=beaker.gauges.systems_idle_manual.all
    &target=beaker.gauges.systems_manual.all
    &target=beaker.gauges.systems_recipe.all
```

Utilization of shared systems

```
http://graphite.example.com/render/?width=1024&height=400
    &areaMode=stacked
    &target=beaker.gauges.systems_idle_automated.shared
    &target=beaker.gauges.systems_idle_broken.shared
    &target=beaker.gauges.systems_idle_manual.shared
    &target=beaker.gauges.systems_manual.shared
    &target=beaker.gauges.systems_recipe.shared
```

Recipe queue

```
http://graphite.example.com/render/?width=1024&height=400
    &areaMode=stacked
    &target=beaker.gauges.recipes_new.all
    &target=beaker.gauges.recipes_processed.all
    &target=beaker.gauges.recipes_queued.all
    &target=beaker.gauges.recipes_running.all
    &target=beaker.gauges.recipes_scheduled.all
    &target=beaker.gauges.recipes_waiting.all
```

Recipe throughput

```
http://graphite.example.com/render/?width=1024&height=400
    &target=beaker.counters.recipes_completed
    &target=beaker.counters.recipes_cancelled
    &target=beaker.counters.recipes_aborted
```

3.17 Reporting from the Beaker database

Beaker's integration with Graphite can provide useful insights into the real-time health and performance of your Beaker installation (see *Integration with Graphite*). However, for more in-depth analysis you may prefer to use an external query/reporting tool to extract data directly from Beaker's database.

Beaker's source includes a number of supported reporting queries which may be useful for your Beaker site. They are installed with the `beaker-server` package under `/usr/lib/python*/site-packages/bkr/server/reporting-queries`, or you can browse the

queries online in Beaker's git (be sure to select the correct branch for your version of Beaker). These queries are "supported" in the sense that they are tested by Beaker's test suite, and if the queries require changes in future releases this will be called out in the release notes. Advance warning will also be provided for any such changes on the beaker-devel mailing list.

The supported SQL queries are written using the MySQL SQL dialect, and automatically tested against MySQL. Accordingly, they may require translation in order to be used with tools based on other SQL dialects.

These queries are intended to provide guidance for "interesting questions" that a business intelligence system connected to Beaker may want to answer. They can be tweaked to use different statistical functions, query different date ranges, adapt filtering rules from another supported query, parametrized in a reporting tool, etc.

As noted above, Beaker's database schema is subject to change in future releases. If your external reporting queries stray beyond the schema assumptions captured in the supported queries, then your queries may break without notice when upgrading to a new Beaker release. If this occurs, you must then examine the detailed schema upgrade notes for that release and ensure the reporting tool's queries are updated as necessary.

Suggestions for additional supported queries are welcome, and may be filed as enhancement requests for the Beaker community project in GitHub.

3.18 How do I. . . ?

This section provides pointers for resolving common administrative problems and tasks in Beaker.

3.18.1 . . . set install options for an entire OS major version?

Sometimes you may want to apply install options to all distro trees for a given OS major version, without explicitly setting the options every time you import a distro tree. For example, as of Fedora 18 Anaconda added a new boot option `serial` which makes it copy install-time console settings to the installed system. If your Beaker site is using serial consoles you may want to add `serial` to the kernel options for every Fedora 18 installation. You can do that by editing the install options for that OS major version from the "OS Versions" page. See *OS versions*.

3.18.2 . . . update the inventory details for a system?

The inventory details for a system are gathered automatically using the *inventory-task* task. The easiest way to run this task is to use the `machine-test` workflow to generate and submit an appropriate job definition:

```
bkr machine-test --inventory --family=RedHatEnterpriseLinux6 \
     --arch=x86_64 --machine=<FQDN>
```

Refer to *bkr-machine-test* for more details.

3.18.3 . . . store my log files somewhere other than the lab controller?

Beaker has the option of moving it's log files from the default location of the lab controller, to a remote archive server, via rsync.

If your primary aim in using the archive server is to free up space on the lab controller, mounting a file system backed by bulk storage may be a better solution. However if this is not a preferred option, and if the size of Beaker's job logs files exceeds the storage available to your lab controllers, or if you need to centralize log storage for administrative reasons, an *archive server* may be a suitable approach.

3.19 Administrative command reference

3.19.1 beaker-create-ipxe-image: Generate and upload iPXE boot image to Glance

Synopsis

beaker-create-ipxe-image [*options*]

Description

Generates a bootable image containing the iPXE network boot loader, and a configuration pointing at this Beaker instance.

Beaker uses this image as part of the support for provisioning dynamic VMs in OpenStack. The image needs to be created once when OpenStack integration is enabled. The credentials given here need to have the permission to create a public image in OpenStack.

This command requires read access to the Beaker server configuration. Run it as root or as another user with read access to the configuration file.

Options

--os-username <username>
: OpenStack user name for establishing a new trust between Beaker and the given user.

--os-password <password>
: OpenStack user password for establishing a new trust between Beaker and the given user.

--os-project-name <project-name>
: OpenStack project name for establishing a new trust between Beaker and the given user.

--os-user-domain-name <user-domain-name>
: OpenStack user domain name for establishing a new trust between Beaker and the given user.

--os-project-domain-name <project-domain-name>
: OpenStack project domain name for establishing a new trust between Beaker and the given user.

--no-upload
: Do not upload the generated image to OpenStack. The image temp file is left on disk and its filename is printed. Use this if you need to examine or manipulate the image before uploading it to Glance manually.

Exit status

Non-zero on error, otherwise zero.

Examples

OpenStack integration must be configured (see *Integration with OpenStack*) before running this command.

This command creates an iPXE image in Glance where the OpenStack authentication uses default domain information:

```
beaker-create-ipxe-image \
    --os-username beaker \
    --os-password beaker \
    --os-project-name beaker
```

Use the options shown in this command when OpenStack requires user and project domain names:

```
beaker-create-ipxe-image \
    --os-username beaker \
    --os-password beaker \
    --os-project-name beaker \
    --os-user-domain-name=domain.com \
    --os-project-domain-name=domain.com
```

3.19.2 beaker-create-kickstart: Generate Anaconda kickstarts

Synopsis

beaker-create-kickstart *--user* <username> (*--recipe-id* <recipeid> | *--system* <fqdn> *--distro-tree-id* <distrotreeid>) [*options*]

Description

`beaker-create-kickstart` is used to generate customised kickstarts. Its main purpose is for testing alternative templates, template variables, and kernel options.

The generated kickstart is based off of a combination of a running/completed recipe, system, and distro tree. It can then be further modified with any of the available options. The resulting kickstart will be printed to stdout.

This command requires read access to the Beaker server configuration. Run it as root or as another user with read access to the configuration file.

Options

-u <username>, **--user** <username>
: Used for any user related options in the kickstart (i.e root password).

-r <recipeid>, **--recipe-id** <recipeid>
: Use <recipeid> as the basis of the kickstart.

-f <fqdn>, **--system** <fqdn>
: This system, combined with *--distro-tree-id*, form the basis of the kickstart. Alternatively, *--recipe-id* can be used.

-d <distrotreeid>, **--distro-tree-id** <distrotreeid>
: This distro tree, combined with *--system*, form the basis of the kickstart. Alternatively, *--recipe-id* can be used.

-t <directory>, **--template-dir** <directory>
: Specify an additional <directory> where kickstart templates can be found. Templates in this directory will take precedence over templates in the standard template directories. The templates need to be organized in a directory hierarchy that Beaker understands, see *Overriding kickstart snippets*.

-m <options>, **--ks-meta** <options>
: Pass <options> into the kickstart.

-p <options>, **--kernel-options-post** <options>
Pass <options> to the kernel in the %post section of the kickstart.

Exit status

Non-zero on error, otherwise zero.

Examples

Say you are developing a custom template for the `timezone` snippet, and you want to test the effect it will have on Beaker's kickstarts before you put it live in `/etc/beaker`. Create a new directory, for example `./template-work`, mirroring the structure of snippets under `/etc/beaker`. Your new `timezone` snippet would be placed in `./template-work/snippets/timezone`.

This command will generate a kickstart based on an existing recipe, looking up templates from your custom directory:

beaker-create-kickstart –recipe-id 150 –template-dir ./template-work

You can generate a kickstart for the same recipe but without your custom templates, and then diff them to see what changed:

beaker-create-kickstart –recipe-id 150

You can also use this command to test the effect that install options will have for a particular system, before you set them in Beaker:

beaker-create-kickstart –user admin –system invalid.example.com –distro-tree-id 120 –ks-meta "grubport=0x3f8 ignoredisk=–only-use=vda"

3.19.3 beaker-import: Import distros

Synopsis

beaker-import [*options*] <distro_url> ...

Description

Imports a distro from the given `distro_url`. A valid `distro_url` is nfs://, http:// or ftp://. Multiple `distro_url` can be specified with the primary `distro_url` being either http:// or ftp://.

In order for an import to succeed, a `.treeinfo` or a `.composeinfo` must be present at the supplied `distro_url`. Alternatively, you can also do what is called a "naked" import by specifying `--family`, `--version`, `--name`, `--arch`, `--kernel`, `--initrd`. Only one tree can be imported at a time when doing a naked import.

Options

-j, --json
Prints the tree to be imported, in JSON format

-c <cmd>, **--add-distro-cmd** <cmd>
Command to run to add a new distro. By default this is **/var/lib/beaker/addDistro.sh**

-n <name>, **--name** <name>
Alternate name to use, otherwise we read it from `.treeinfo`

-t <tag>, **--tag** <tag>
Additional tags to add to the distro.

-r, **--run**
Run automated Jobs

-v, **--debug**
Show debug messages

--dry-run
Do not actually add any distros to beaker

-q, **--quiet**
Less messages

--family <family>
Specify family

--variant <variant>
Specify variant. Multiple values are valid when importing a compose>=RHEL7.

--version <version>
Specify version

--kopts <kernel options>
Add kernel options to use for install

--kopts-post <post install kernel options>
Add kernel options to use post install

--ks-meta <ksmeta variables>
Add variables to use in kickstart templates

--preserve-install-options
Do not overwrite the *'Install Options' (Kickstart Metadata, Kernel Options, & Kernel Options Post)* already stored for the distro. This option can not be used with any of –kopts, –kopts-post, or –ks-meta

--buildtime <buildtime>
Specify build time

--arch <arch>
Specify arch. Multiple values are valid when importing a compose

--ignore-missing-tree-compose
If a specific tree within a compose is missing, do not print any errors

Naked tree options

These options only apply when importing without a .treeinfo or .composeinfo.

--kernel <kernel>
Specify path to kernel (relative to distro_url)

--initrd <initrd>
Specify path to initrd (relative to distro_url)

--lab-controller <lab_controller>
Specify which lab controller to import to. Defaults to http://localhost:8000.

Exit status

Non-zero on error, otherwise zero.

If `--ignore-missing-tree-compose` is not specified, a non-zero exit status will be returned if any of the trees cannot be imported.

Examples

When `.composeinfo` and `.treeinfo` are available:

```
$ beaker-import \
    http://mymirror.example.com/pub/fedora/linux/releases/17/Fedora/ \
    ftp://mymirror.example.com/pub/fedora/linux/releases/17/Fedora/  \
    nfs://mymirror.example.com:/pub/fedora/linux/releases/17/Fedora/
```

Naked import:

```
$ beaker-import \
   http://mymirror.example.com/FedoraNew/ \
    --family FedoraNew \
    --name FedoraNew-atomic \
    --arch x86_64 \
    --version 1 \
    --initrd=images/pxeboot/initrd.img \
    --kernel=images/pxeboot/vmlinuz \
```

The above command will import the distro tree at the given URL with the supplied meta-data. The locations of the "initrd" and the "kernel" are relative to this URL.

3.19.4 beaker-init: Initialize and populate the Beaker database

Synopsis

beaker-init [*options*]

Description

Initializes and populates Beaker's database if empty, or upgrades it to the latest schema version.

This command reads the server configuration and connects to the database in the same way as the Beaker application itself does. For new Beaker installations, ensure you have configured the database in `/etc/beaker/server.cfg` before you run this command so that it can connect to the database in order to initialize it.

When initializing an empty database, you must supply the *`--user`*, *`--password`*, *`--email`*, and *`--fullname`* options so that **beaker-init** can create an admin account.

This command requires read access to the Beaker server configuration. Run it as root.

Options

`-c` `<path>`, **`--config`** `<path>`
: Read server configuration from <path> instead of the default `/etc/beaker/server.cfg`.

--user <username>
Create a new user with administrative privileges using the given username.

--password <password>
Set the administrative user's password to the given value. This can be used as an escape hatch in case you are unable to log in to Beaker as an admin.

--email <email>
Update the administrative user's email address to the given value.

--fullname <name>
Human-friendly display name for the administrative user.

--downgrade <version>
Downgrade the database to the given version instead of upgrading.

The version may be given as a Beaker version number with any number of components (for example, `22` or `22.0-1.el6eng`), or it may be given as a schema version identifier as listed in *Downgrading* (for example, `54395adc8646`).

--check
Check if the database schema is up to date, instead of performing any upgrades.

When this option is given the database is not modified. If the database is up to date (that is, running **beaker-init** would not perform any upgrades) then the exit status will be 0. If the database is not up to date then the exit status will be 1.

If this option is combined with *--downgrade* then the check will be performed against the requested downgrade version, not the latest version.

--background
Detach from the terminal and send all log messages to syslog. The pid of the background process is written to `/var/run/beaker-init.pid`, and removed when the background process is complete.

--debug
Show detailed progress information and debugging messages.

Exit status

For normal operations the exit status is zero on success, or non-zero on error.

When the *--check* option is used, the exit status is zero if the database is up to date, 1 if it is requires updates, or some other value on error.

Examples

Populate the database for a new Beaker installation:

```
beaker-init --user admin \
    --password changeme \
    --email dcallagh@redhat.com \
    --fullname 'Dan Callaghan'
```

Upgrade an existing Beaker database, while Beaker is offline (see *Upgrading an existing installation*):

```
beaker-init
```

If your Beaker site does automated deployments with a tool such as Ansible, you can combine the *--background* and *--check* options to perform long-running database upgrades in a robust manner. For example, the following

Ansible tasks invoke **beaker-init** in the background, wait for the pid file to be removed, and then check that the background process completed successfully:

```
- name: start db migration
  command: beaker-init --background --debug

- name: wait for db migration to finish
  wait_for: path=/var/run/beaker-init.pid state=absent

- name: check db migration completed successfully
  command: beaker-init --check
  changed_when: False
```

3.19.5 beaker-log-delete: Delete expired jobs

Synopsis

beaker-log-delete [*options*]

Description

Deletes expired jobs and permanently purges log files from Beaker and/or archive server.

This command reads the server configuration and connects to the database in the same way as the Beaker application itself does. Ensure you have configured the database in `/etc/beaker/server.cfg` before you run this command so that it can connect to the database in order to find expired jobs and remove them.

HTTP server must be able to handle WebDAV DELETE operations on the log directory's base path (HTTP digest and Kerberos authentication are supported).

To enable HTTP digest, configure account in `/etc/beaker/server.cfg`:

```
beaker.log_delete_user = ""
beaker.log_delete_password = ""
```

This command requires read access to the Beaker server configuration. Run it as root.

Options

-c <path>, **--config** <path>
: Read server configuration from <path> instead of the default /etc/beaker/server.cfg.

-v, --verbose
: Print the path/URL of deleted files to stdout

--debug
: Show detailed progress information and debugging messages.

--dry-run
: Expired jobs are not removed.

--limit
: Limit number of expired jobs whose logs will be deleted.

Exit status

For normal operations the exit status is zero on success, or non-zero on error.

Examples

Delete first 50 expired jobs:

```
beaker-log-delete --limit 50
```

Delete expired jobs and display debug messages:

```
beaker-log-delete --debug
```

Expired jobs are only listed and not deleted:

```
beaker-log-delete --dry-run --verbose
```

3.19.6 beaker-repo-update: Update cached harness packages

Synopsis

beaker-repo-update [*options*]

Description

Updates the Beaker server's local cache of harness packages.

The harness and its dependencies are installed on a test system when Beaker is running a scheduled job on it.

The harness packages are generally built separately for every distro family supported by Beaker (Fedora 20, Red Hat Enterprise Linux 7, etc). The **beaker-repo-update** command fetches harness packages for every distro family which exists in Beaker. Therefore, the first time you import a new distro family into Beaker, you should run **beaker-repo-update** in order to cache the harness packages for the new distro family.

This command requires read access to the Beaker server configuration, and write access to the harness package cache. Run it as root.

Options

-b <url>, **--baseurl** <url>
: Fetch harness packages from subdirectories under <url>. By default, packages are fetched from the Beaker web site.

-d <path>, **--basepath** <path>
: Cache harness packages in subdirectories under <path>. By default, packages are cached in `/var/www/beaker/harness`. This location is served by Apache, and used by Beaker to install the harness on test systems.

--debug
: Show detailed progress information and debugging messages.

-c <path>, **--config-file** <path>
: Read server configuration from <path> instead of the default `/etc/beaker/server.cfg`.

Exit status

Non-zero on error, otherwise zero.

If fetching packages fails for a particular distro family, a warning is printed to stderr and the repo is skipped. This is not considered an error.

Examples

Update the cached harness packages after new versions have been released:

```
beaker-repo-update
```

Fetch release candidate packages from the `harness-testing` repositories on the Beaker web site:

```
beaker-repo-update -b https://beaker-project.org/yum/harness-testing/
```

3.19.7 beaker-usage-reminder: Send Beaker usage reminder

Synopsis

beaker-usage-reminder [*options*]

Description

`beaker-usage-reminder` is used to send users email reminders and allow them to keep track of their Beaker systems usage.

To ensure users get their reminders, beaker administrators can setup a cron job on the beaker server to run this command at regular intervals (e.g. daily).

This command requires read access to the Beaker server configuration. Run it as root or as another user with read access to the configuration file.

Options

`--reservation-expiry` `<hours>`
: Warn users about their reservations expiring less than <hours> in the future. The default is 24.

`--reservation-length` `<days>`**`, --waiting-recipe-age`** `<hours>`
: Remind users about their systems which have been reserved for longer than <days> and there is at least one recipe waiting for longer than <hours> for those systems. The defaults are 3 and 1 respectively.

`--delayed-job-age` `<days>`
: Warn users about their jobs which have been queued for longer than <days>. The default is 14.

Exit status

Non-zero on error, otherwise zero.

Examples

To remind users who have reservations expiring in 48 hours:

```
beaker-usage-reminder --reservation-expiry 48
```

To remind users who have systems that have been reserved for longer than 3 days and there is at least one recipe waiting for longer than 1 hour for those systems:

```
beaker-usage-reminder --reservation-length 3 --waiting-recipe-age 1
```

To remind users who have delayed jobs for longer than 3 days:

```
beaker-usage-reminder --delayed-job-age 3
```

3.19.8 beaker-sync-tasks: Tool to sync local Beaker task RPMs from a remote Beaker installation

Synopsis

beaker-sync-tasks [*options*]

Description

`beaker-sync-tasks` is a script to sync local task RPMs from a remote Beaker installation.

Syncing protocol:

- Task doesn't exist in local: copy it.
- Task exists in local: Overwrite it, if it is a different version on the remote
- Tasks which exist on the local and not on the remote are left untouched

Options

-h, --help
Show this help message and exit

--remote `<remote_server>`
Remote Beaker Instance

--force
Do not ask before overwriting task RPMs

--debug
Display messages useful for debugging (verbose)

Exit status

Non-zero on error, otherwise zero.

Examples

Sync tasks from a remote Beaker server and display debug messages:

beaker-sync-tasks –remote=http://127.0.0.1/bkr –debug

Don't prompt before beginning task upload:

beaker-sync-tasks –remote=http://127.0.0.1/bkr –force

More information

Querying the existing tasks: The script communicates with the remote Beaker server via XML-RPC calls and directly interacts with the local Beaker database.

Adding new tasks: The tasks to be added to the local Beaker database are first downloaded in the task directory (usually, /var/www/beaker/rpms). Each of these tasks are then added to the Beaker database and finally createrepo is run.

3.19.9 product-update: Tool to update CPE identifiers for products in Beaker

Synopsis

product-update [*options*]

Description

The **product-update** command updates CPE identifiers for products in Beaker from an XML file or URL.

Beaker administrators can setup a cron job on the beaker server to run this command at regular intervals (e.g. daily).

Options

-c <path>, **--config-file** <path>
Read server configuration from <path> instead of the default /etc/beaker/server.cfg.

-p <file>, **--product-file** <file>
Load product XML data from <file>

--product-url <url>
Load product XML or JSON data from <url>

Exit status

Non-zero on error, otherwise zero.

Examples

Update CPE identifiers for products in Beaker from a URL:

```
product-update --product-url 'https://example.com/api/v1/releases/?fields=id,cpe'
```

/

GET /, **??**

/activity

GET /activity/, **??**
GET /activity/distro, **??**
GET /activity/distrotree, **??**
GET /activity/group, **??**
GET /activity/labcontroller, **??**
GET /activity/system, **??**

/auth

GET /auth/whoami, **??**
POST /auth/login_krbv, **??**
POST /auth/login_oauth2, **??**
POST /auth/login_password, **??**
POST /auth/logout, **??**

/available

GET /available/, **??**

/free

GET /free/, **??**

/groups

GET /groups/, **??**
GET /groups/(group_name), **??**
POST /groups/, **??**
POST /groups/(group_name)/excluded-users/, **??**
POST /groups/(group_name)/members/, **??**
POST /groups/(group_name)/owners/, **??**
POST /groups/(group_name)/permissions/, **??**
DELETE /groups/(group_name), **??**
DELETE /groups/(group_name)/excluded-users/, **??**
DELETE /groups/(group_name)/members/, **??**
DELETE /groups/(group_name)/owners/, **??**
DELETE /groups/(group_name)/permissions/, **??**
PATCH /groups/(group_name), **??**

/healthz

HEAD /healthz/, **??**
GET /healthz/, **??**

/jobs

GET /jobs/(int:id), **??**
GET /jobs/(int:id).junit.xml, **??**
GET /jobs/(int:id)/activity/, **??**
POST /jobs/(int:id)/status, **??**
POST /jobs/+inventory, **??**
DELETE /jobs/(int:id), **??**
PATCH /jobs/(int:id), **??**

/labcontrollers

GET /labcontrollers/(fqdn), **??**
POST /labcontrollers/, **??**
PATCH /labcontrollers/(fqdn), **??**

/mine

GET /mine/, **??**

/pools

GET /pools/, **??**
GET /pools/(pool_name)/, **??**
GET /pools/(pool_name)/access-policy/, **??**
POST /pools/, **??**
POST /pools/(pool_name)/access-policy/, **??**
POST /pools/(pool_name)/access-policy/rules/, **??**
POST /pools/(pool_name)/systems/, **??**
PUT /pools/(pool_name)/access-policy/, **??**
DELETE /pools/(pool_name)/, **??**
DELETE /pools/(pool_name)/access-policy/rules/, **??**
DELETE /pools/(pool_name)/systems/, **??**
PATCH /pools/(pool_name)/, **??**

/power

PUT /power/(fqdn)/, ??

/powertypes

GET /powertypes/, ??
POST /powertypes/, ??
DELETE /powertypes/(id), ??

/recipes

GET /recipes/(int:id), ??
GET /recipes/(int:id)/logs/(path:path), ??
GET /recipes/(recipe_id)/, ??
GET /recipes/(recipe_id)/logs/, ??
GET /recipes/(recipe_id)/logs/(path:path), ??
GET /recipes/(recipe_id)/tasks/(task_id)/logs/, ??
GET /recipes/(recipe_id)/tasks/(task_id)/logs/(path:path), ??
GET /recipes/(recipe_id)/tasks/(task_id)/results/(result_id)/logs/, ??
GET /recipes/(recipe_id)/tasks/(task_id)/results/(result_id)/logs/(path:path), ??
GET /recipes/(recipe_id)/watchdog, ??
GET /recipes/(recipeid)/tasks/(taskid)/comments/, ??
GET /recipes/(recipeid)/tasks/(taskid)/logs/(path:path), ??
GET /recipes/(recipeid)/tasks/(taskid)/results/(resultid)/comments/, ??
GET /recipes/(recipeid)/tasks/(taskid)/results/(resultid)/logs/(path:path), ??
POST /recipes/(recipe_id)/status, ??
POST /recipes/(recipe_id)/tasks/(task_id)/results/, ??
POST /recipes/(recipe_id)/tasks/(task_id)/status, ??
POST /recipes/(recipe_id)/watchdog, ??
POST /recipes/(recipeid)/tasks/(taskid)/comments/, ??
POST /recipes/(recipeid)/tasks/(taskid)/results/(resultid)/comments/, ??
POST /recipes/by-fqdn/(fqdn)/watchdog, ??
POST /recipes/by-taskspec/(taskspec)/watchdog, ??
PUT /recipes/(recipe_id)/logs/(path:path), ??
PUT /recipes/(recipe_id)/tasks/(task_id)/logs/(path:path), ??
PUT /recipes/(recipe_id)/tasks/(task_id)/results/(result_id)/logs/(path:path), ??
PATCH /recipes/(int:id), ??
PATCH /recipes/(int:id)/reservation-request, ??
PATCH /recipes/(recipe_id)/tasks/(task_id)/, ??

/recipesets

GET /recipesets/(int:id), ??
POST /recipesets/(int:id)/status, ??
PATCH /recipesets/(int:id), ??
PATCH /recipesets/by-taskspec/(taskspec), ??

/systems

GET /systems/(fqdn)/, ??
GET /systems/(fqdn)/access-policy, ??
GET /systems/(fqdn)/active-access-policy/, ??
GET /systems/(fqdn)/activity/, ??
GET /systems/(fqdn)/commands/, ??
GET /systems/(fqdn)/executed-tasks/, ??
GET /systems/(fqdn)/notes/(id), ??
POST /systems/, ??
POST /systems/(fqdn)/access-policy, ??
POST /systems/(fqdn)/access-policy/rules/, ??
POST /systems/(fqdn)/commands/, ??
POST /systems/(fqdn)/installations/, ??
POST /systems/(fqdn)/loan-requests/, ??
POST /systems/(fqdn)/loans/, ??
POST /systems/(fqdn)/notes/, ??
POST /systems/(fqdn)/problem-reports/, ??
POST /systems/(fqdn)/reservations/, ??
PUT /systems/(fqdn)/access-policy, ??
DELETE /systems/(fqdn)/access-policy/rules/, ??
PATCH /systems/(fqdn)/, ??
PATCH /systems/(fqdn)/loans/+current, ??
PATCH /systems/(fqdn)/notes/(id), ??
PATCH /systems/(fqdn)/reservations/+current, ??

/tasks

PATCH /tasks/(int:task_id), ??

/users

GET /users/, ??
GET /users/(path:username), ??
GET /users/+self, ??
POST /users/, ??
POST /users/(path:username)/ssh-public-keys/, ??

`POST /users/(path:username)/submission-delegates/`,
??
`PUT /users/(path:username)/keystone-trust`,
??
`PUT /users/+self/keystone-trust`, **??**
`DELETE /users/(path:username)/keystone-trust`,
??
`DELETE /users/(path:username)/ssh-public-keys/(int:id)`,
??
`DELETE /users/(path:username)/submission-delegates/`,
??
`PATCH /users/(path:username)`, **??**

/view

`GET /view/(fqdn)`, **??**

Symbols

B

C

P

S

U