Beaker User Guide

Release 29.3

Red Hat, Inc.

May 15, 2026

1 Scope of document **1**

2 Audience **3**

3 Contents **5**
3.1 The `bkr` command-line client 5
3.2 Beaker interface 6
3.3 Managing systems in Beaker 20
3.4 Job design 24
3.5 Reserving a system after testing 25
3.6 Workflow 28
3.7 Writing tasks 54
3.8 Tasks provided with Beaker 73
3.9 Troubleshooting 76

HTTP Routing Table **79**

Index **83**

CHAPTER

ONE

SCOPE OF DOCUMENT

This document covers interacting with Beaker through the web and command line interfaces, as well as covering the steps involved in writing new Beaker tasks.

CHAPTER

TWO

AUDIENCE

This document is aimed at developers and quality assurance engineers that need to work with Beaker, writing, running and interpreting the results of Beaker tasks.

CHAPTER

THREE

CONTENTS

3.1 The `bkr` command-line client

3.1.1 Supported platforms

The Beaker command line client is fully supported on recent versions of Fedora and on Red Hat Enterprise Linux 6. Most commands are also supported on Red Hat Enterprise Linux 5 (when this is not the case, it will be noted in the documentation of the affected command by indicating the minimum required version of Python).

3.1.2 Installing and configuring the client

Pre-built Beaker packages are available from the Download section of Beaker's web site. Download the repo file that suits your requirements and copy it to `/etc/yum.repos.d`.

Install the `beaker-client` package:

```
$ sudo yum install beaker-client
```

A sample configuration is installed as `/usr/share/doc/beaker-client-*/client.conf.example`. Copy it to `/etc/beaker/client.conf` or `~/.beaker_client/config` and edit it there.

First, set the URL of your Beaker server *without* trailing slash:

```
HUB_URL = "http://mybeaker.example.com/bkr"
```

You'll then need to configure how your Beaker client authenticates with the Beaker server. You can use either password authentication, or Kerberos authentication. For password authentication, add the following:

```
AUTH_METHOD = "password"
USERNAME = "username"
PASSWORD = "password"
```

If instead Kerberos authentication is preferred:

```
AUTH_METHOD = "krbv"
KRB_REALM = "REALM.EXAMPLE.COM"
```

To verify it is working properly:

```
$ bkr whoami
```

It should print your username.

3.1.3 Using the client

For full details about the `bkr` client and its subcommands, refer to the *Beaker client man pages*. A summary of some common commands is given below.

To create a simple Job workflow, the beaker client comes with the command `bkr workflow-simple`. This simple Job workflow will create the XML for you from various options passed at the shell prompt, and submit this to the Beaker server. To see all the options that can be passed during invocation of `workflow-simple`, use the following command:

```
$ bkr workflow-simple --help
```

A common set of parameters that may be passed to the workflow-simple options would be the following:

```
$ bkr workflow-simple --username=<user> --password=<passwd> --dryrun \
    --arch=<arch> --distro=<distro_name> --task=<task_name> \
    --type=<TYPE> --whiteboard=<whiteboard_name> --debug > my_job.xml
```

To submit an existing Job workflow:

```
$ bkr job-submit job_xml
```

If successful, you will be shown the Job ID and the progress of your Job.

To watch a Job:

```
$ bkr job-watch J:job_id
```

To cancel a Job you have created:

```
$ bkr job-cancel J:job_id
```

To show all Tasks available for a given distro:

```
$ bkr task-list distro
```

To add a Task:

```
$ bkr task-add task_rpm
```

3.2 Beaker interface

The Beaker web interface provides convenient access to monitor, configure and control systems, jobs and other components of a Beaker installation.

Note: The Beaker web interface requires a reasonably modern browser that can appropriately handle Javascript and CSS styling. The use of Firefox 10 or later (including Firefox ESR 10) is recommended, but the interface should work reliably with any modern standards compliant browser.

3.2.1 Systems

Beaker provides an inventory of Systems(these could be a physical machine, laptop, virtual guest, or resource) attached to lab controllers. Systems can be added, removed, have details changed, and be provisioned among other things.

System searching

System searches are conducted by clicking on one of the items of the "System" menu at the top of the page.

- System Searches
  - *All*
    - Will search through all existing Systems listed in Beaker (excluding systems which have been removed).
  - *Available*
    - Will search through only Systems that the currently logged in user has permission to reserve.
  - *Free*
    - Will search through only Systems that the currently logged in user has permission to reserve and are currently free.
  - *Removed*
    - Will search through only Systems that have been removed.

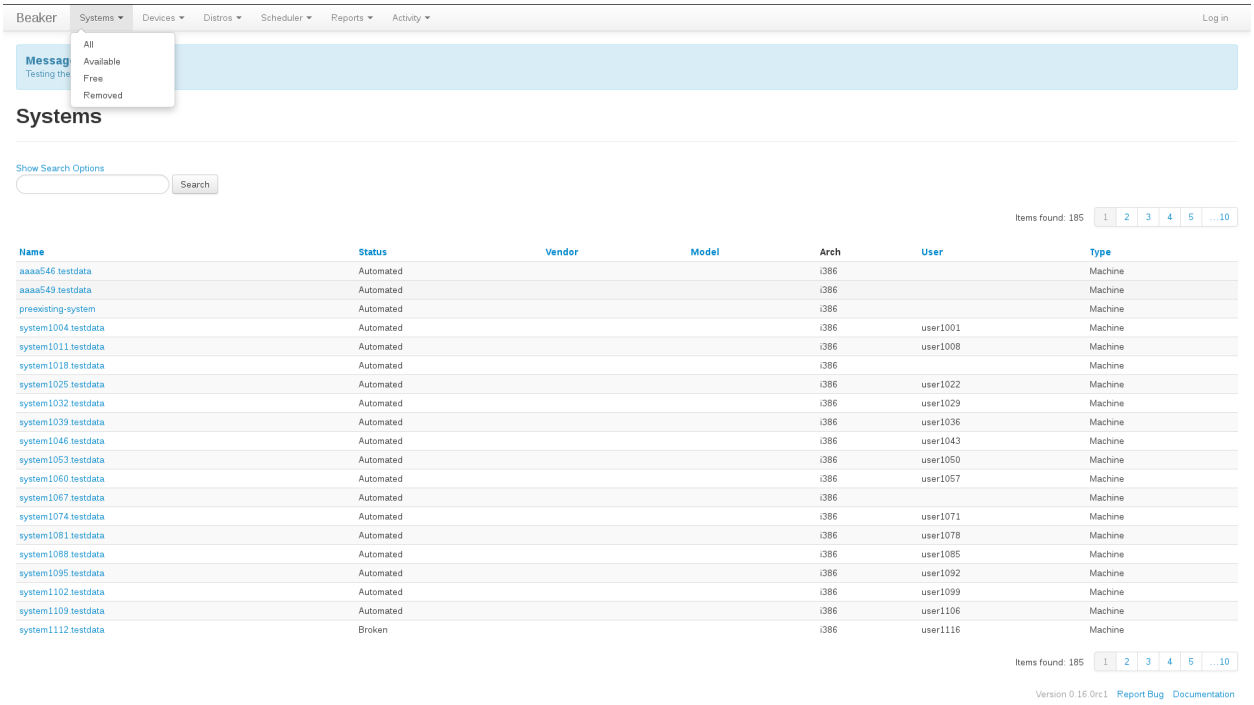


Figure 3.1: System Menu

The search panel has two modes; simple and advanced. The simple search is the default, and the default search is of the System name, using the "contains" operator. To show the advanced search options, click *Show Search Options*.

The first column ("Table") is the attribute on which the search is being performed; The second ("Operation") is the type of search, and the third ("Value") is the actual value to search on. To add another search criteria (row), click the "Add(+)" link just below the "Table" column. When using more than just one search criteria, the default operation between the criteria is an SQL AND operation. The operators change depending on what type of attribute is being searched.

Wildcards

No operator provides explicit wildcards other than the is operation, which allows the * wildcard when searching an attribute which is a string.

The kind of data returned in the System search can be changed by adding/removing the result columns. To do this the "Toggle Result Columns" link is pressed and the columns checked/unchecked.

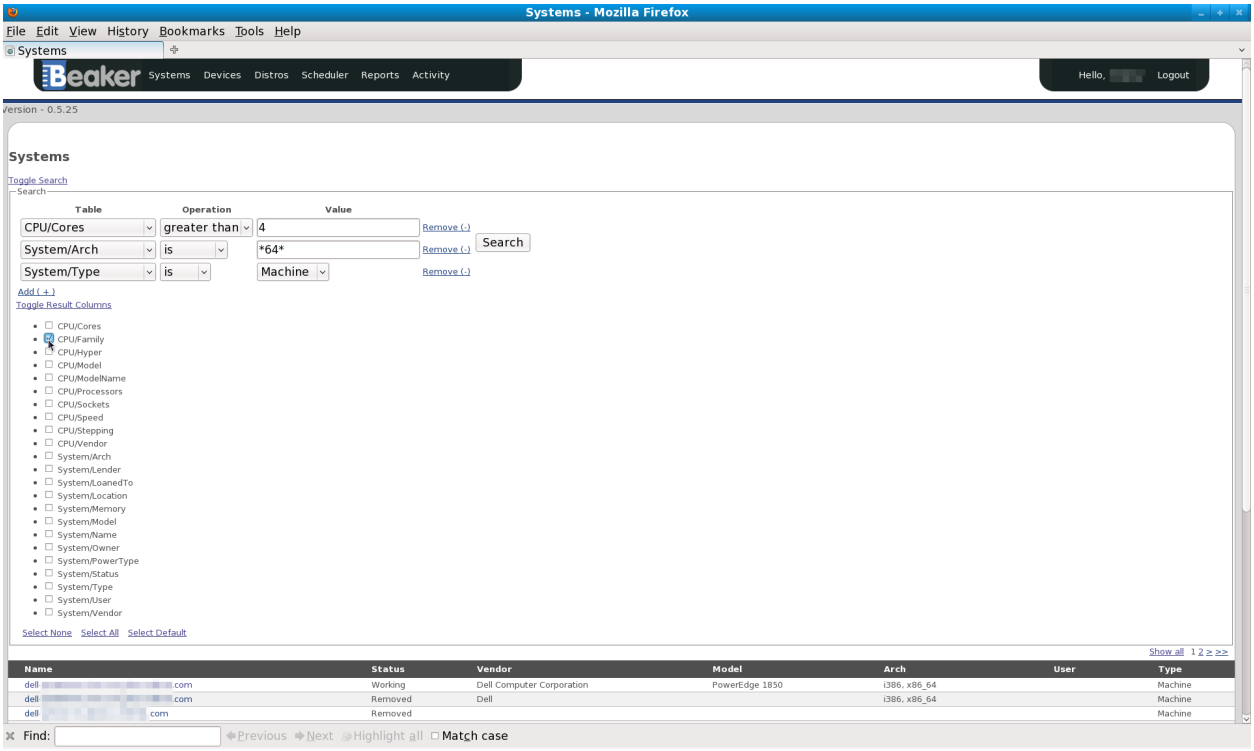


Figure 3.2: Searching for a System

Shortcut for finding systems you are using

The top right hand corner has a menu which starts with Hello, followed by the name of the user currently logged in. Click on this, then down to "My Systems"

Adding a driver disk

Some systems may need a driver disk to be able to install certain releases. In order to use a driver disk you need to host the driver disk under the TFTP server so that it's available during netboot. You also need to tell Beaker which families the driver disk is needed for.

- First step is to install the driver disk on your lab controller(s).
  - The following example assumes tftpboot is under /var/lib/tftpboot; this is true for RHEL6 and newer distros.
  - Make a directory to host the driver disk.

    ```
    # mkdir -p /var/lib/tftpboot/dd/rhel6
    ```

  - Copy the driver disk to this directory.

```
# cp dd.img /var/lib/tftpboot/dd/rhel6
```

- Second step is to set the family install options for the system that needs the driver disk.
  - If you don't have any arch specific install options you need to create one first. Install options are inherited in the order of Arch, Family, and Update.

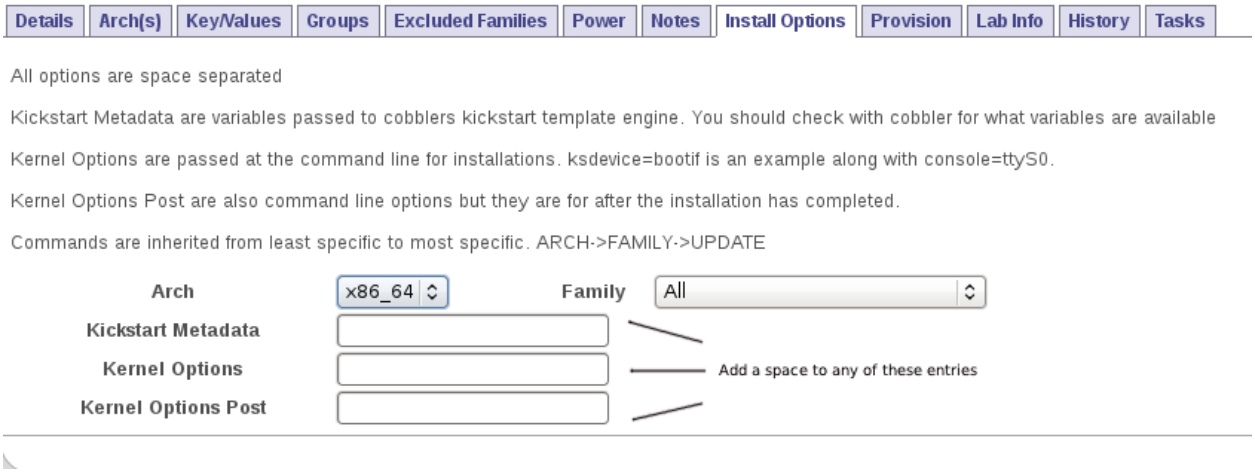


Figure 3.3: Adding a blank install option for arch.

  - Once you have an arch specific entry you can create a family specific entry. The image below shows adding the driver disk we created for RHEL6. Notice that the path is from the chroot of the tftpserver, not /var/lib/tftpboot/.

System details

After finding a System in the search page, clicking on the System name will show the System details. To change these details, you must be logged in as either the owner of the System, or an admin.

- *System Name*: Unique hostname that identifies the machine, also referred to as FQDN (fully qualified domain name).
- *Date Checkin*: When the machine was added to the inventory.
- *Last Inventoried*: Last time this machine had its inventory updated
- *Lender*: Name of the organization that has lent this system to beaker's inventory.
- *Serial Number*: Serial Number of the machine.
- *Condition*: This can be one of the following:
  - *Automated*: In a working state, can have jobs run against it.
  - *Manual*: In a working state, can not have jobs run against it (designed so people can test machine without having other people's jobs run on it).
  - *Broken*: Not in a working state and not available to have jobs run on it.
  - *Removed*: System no longer exists in the inventory.
- *Secret*: Stops other people from seeing this system in the inventory.
- *Lab Controller*: The Lab controller to which it is connected.

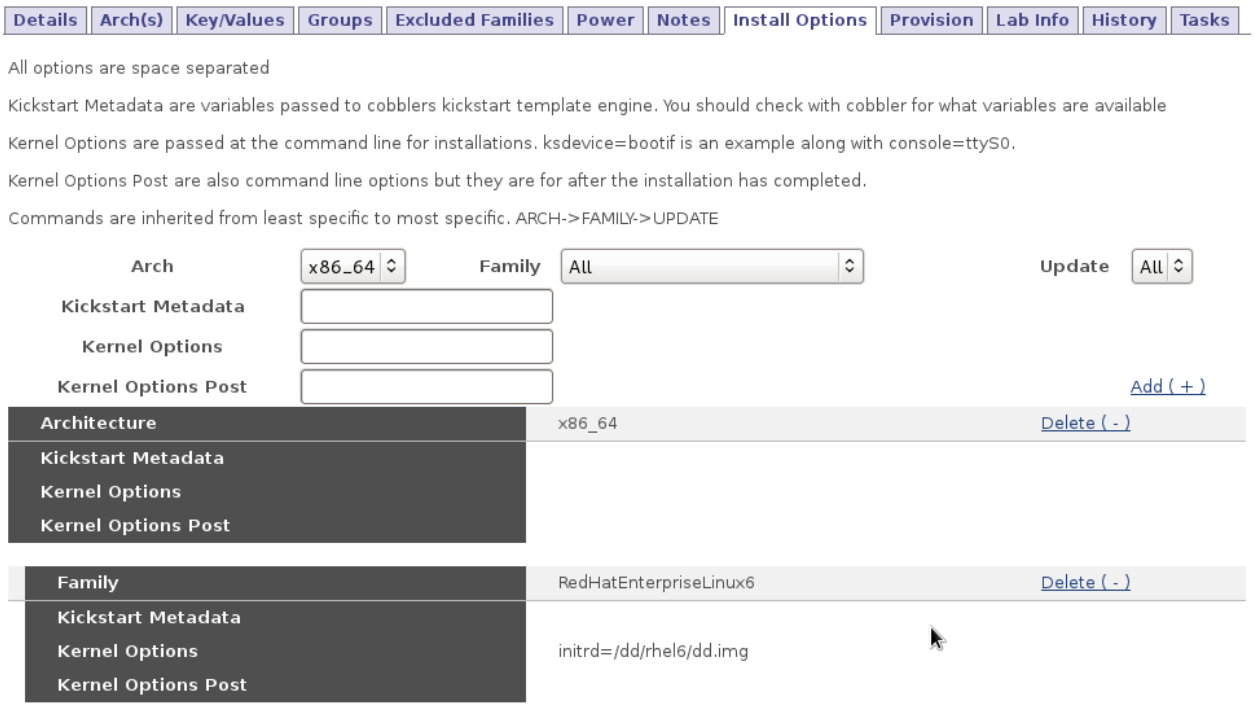


Figure 3.4: Adding a driver disk entry for RHEL6.

- *Type*: This can be one of the following:
  - *Machine*: A physical machine that does not fit the other categories.
  - *Laptop*: A laptop.
  - *Virtual*: A virtual machine, this is just a placeholder that has a hostname and MAC address that corresponds to a DHCP record.
  - *Resource*: Something which is not a computer. i.e a monitor.
  - *Prototype*: New piece of hardware.
- *Last Modification*: The last time the system details were changed.
- *Vendor*: The brand.
- *Model*: The model designation.
- *Location*: The physical location.
- *Owner*: The user who currently owns this machine (by default it is the user who added the entry to inventory, but owners can be reassigned)
- *Loaned To*: The current Loanee. See *Loans*.
- *MAC Address*: The MAC address of the network device.

System details tabs

The system page also has a number of tabs with additional information:

Details Shows the details of the System's CPU, as well as Devices attached to the System.

Arch(es) Shows the architectures supported by the system.

Key/Values Shows further hardware details.

Groups Shows the groups of which this System is a member.

Access Policy Control the level of permission granted to other Beaker users for this system. See *Access policies*.

Excluded Families Are the list of Distros that this System does not support.

Commands Perform remote power operations, as well as clear an existing netboot entry.

Power Config Update a system's power configuration. Only available if you have the correct permissions.

Notes Any info about the system that you want others to see and doesn't fit in anywhere else. If you have admin rights over the system you will be able to add and delete notes, as well as show previously deleted notes.

Install Options These are default options which will be used when a system is provisioned. You can create different options per Arch/Distro combination. See *Install options* for details about the meaning of these options.

Provision Allows the user of this System to install a Distro on it.

Lab Info Will display practical details of the System like cost, power usage, weight etc.

History Shows the activity on this System for the duration of the systems life as an inventory item in Beaker. These activities can also be searched. By default, the simple search does a "contains" search on the Field attribute. Please see *System searching* for details on searching.

System activity

To search through the historical activity of all Systems, navigate to "Activity>Systems" at the top of the page.

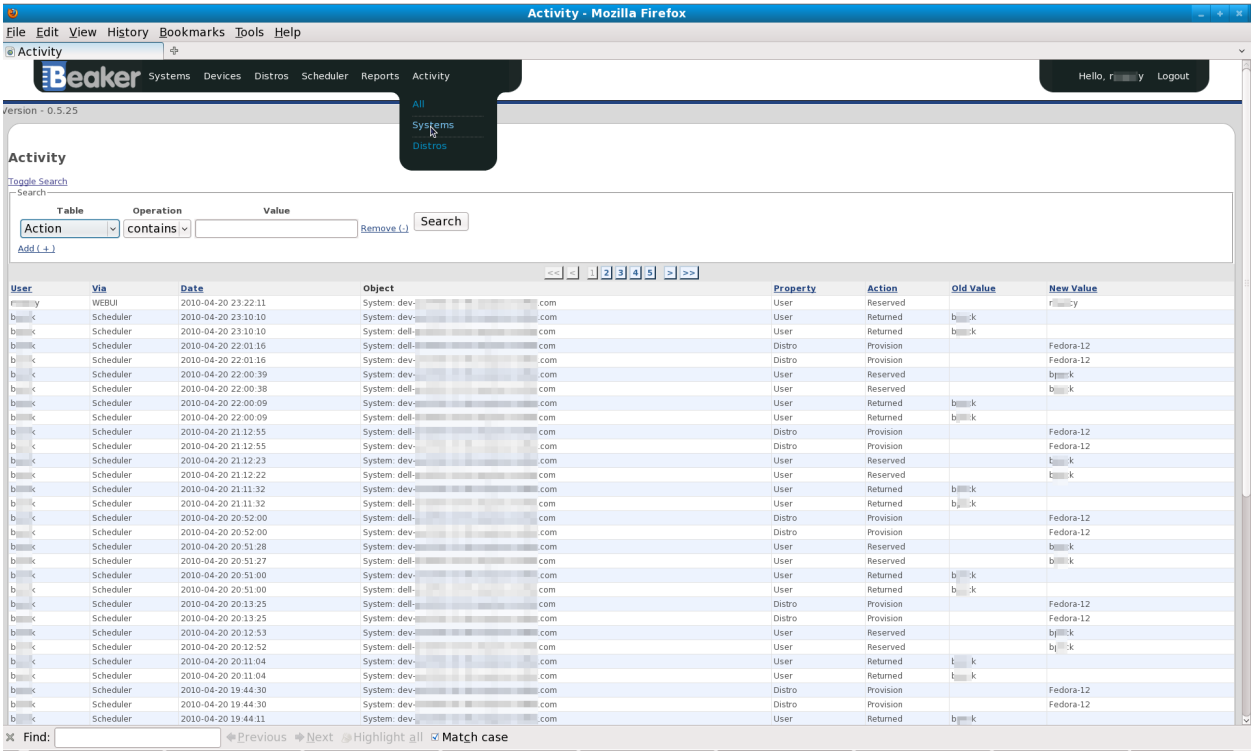


Figure 3.5: Searching through all Systems' activity

To search the history of a specific system, you can also use the "History" tab on the system page. See *System details tabs*.

3.2.2 System pools

Systems can be grouped together into system pools. Any Beaker user can create a system pool. The pool owner can either be a user ("owning user", set by default to the creating user) or modified to be owned by a user group ("owning group"). A system can belong to multiple pools.

Viewing pools

To view a list of all pools, select *Systems* → *Pools* from the menu. To only list pools owned by you, select *Hello* → *My System Pools*.

Creating a new pool

To create a new pool, select *Systems* → *Pools* from the menu and then click the *+Create* link at the bottom left. You'll then be prompted to enter a "Pool Name" and upon clicking *Create*, you will be taken to the pool's page.

Editing a pool

Only a pool owner ("owning user" or "owning group") can edit a system pool.

To edit a pool, select *Hello* → *My System Pools* and click on the name of the pool you wish to edit. From the pool's page, you can add or edit the pool description, change the owner, add and remove systems and update the *pool access policy*.

You can also delete a pool from the pool's page.

Pool activity

To search through the historical activity of all pools, select *Activity* → *System Pools* from the menu.

3.2.3 Distros

Beaker can keep a record of Distros that are available to install on Systems in its Inventory.

Distro searching

The find a particular Distro, click "Distros>All". The default search is on the Distro's name, with a "contains" clause

Distro activity

To search through the historical activity of all Distros, navigate to "Activity>Distros" at the top of the page. The default search is "contains" on the "Property" attribute.

3.2.4 Jobs

The purpose of a Job is to provide an encapsulation for tasks. It is to provide a single point of submission for one or more of these tasks and also reviewing the output and results of running them. The tasks within a Job may or may not be related to each other; although it would make sense to define Jobs based on the relationship of the tasks within it. Once a Job has been submitted you cannot alter its contents, or pause it. You can however cancel it (see *Job results*), and alter a recipe set's priorities (you can only lower the priority level if you are not in the admin group). Adjusting this priority upwards will change which recipe set is run sooner, and vice a versa (See: *Recipes*).

Valid Job Specs

If this is the first time running this Job make sure that at least one system with the specified architecture has access to the specified distro and all the relevant tasks are available to Beaker. To do this, See *System searching*, *Distro searching* and *Task searching* respectively.

Creating your job XML

Beaker jobs are defined using an XML format. For details, see *Job XML*.

You can use the *bkr workflow-simple* client command to generate simple jobs. For more complicated logic, you can write a custom workflow command in Python.

Job submission

There are two ways of submitting a Job through the web UI. They are outlined below.

Submitting a new job

Once you have created an XML Job workflow, you are able to submit it as a new "Job". To do this, go to the "Scheduler > New Job". Click "Browse" to select your XML file, and then hit the "Submit Data" button. The next page shown gives you an opportunity to check/edit your XML before queuing it as a Job by pressing the "Queue" button. See *Job workflow details* for helpful information on writing your Job XML from scratch.

Cloning an existing job

Cloning a Job means to take a Job that has already been run on the System, and re-submit it. To do this you first need to be on the *Job search* page.

Clicking on "Clone" under the Action column will take you to a page that shows the structure of the Job in the XML.

Submitting a slightly different job

If you want to submit a Job that's very similar to a Job already in Beaker,you can use the Clone button to change details of a previous Job and resubmit it!

Searching for jobs

You can search all Beaker jobs from the jobs page. Select *Scheduler* → *Jobs* from the menu. Jobs are listed with the most recent at the top. You can click the *Running*, *Queued*, or *Completed* buttons to filter the list to running, queued, or completed jobs respectively. If you want to look up a specific job, enter its ID in the search box and click *Lookup ID*. Otherwise, you can click *Show Search Options* to search the jobs.

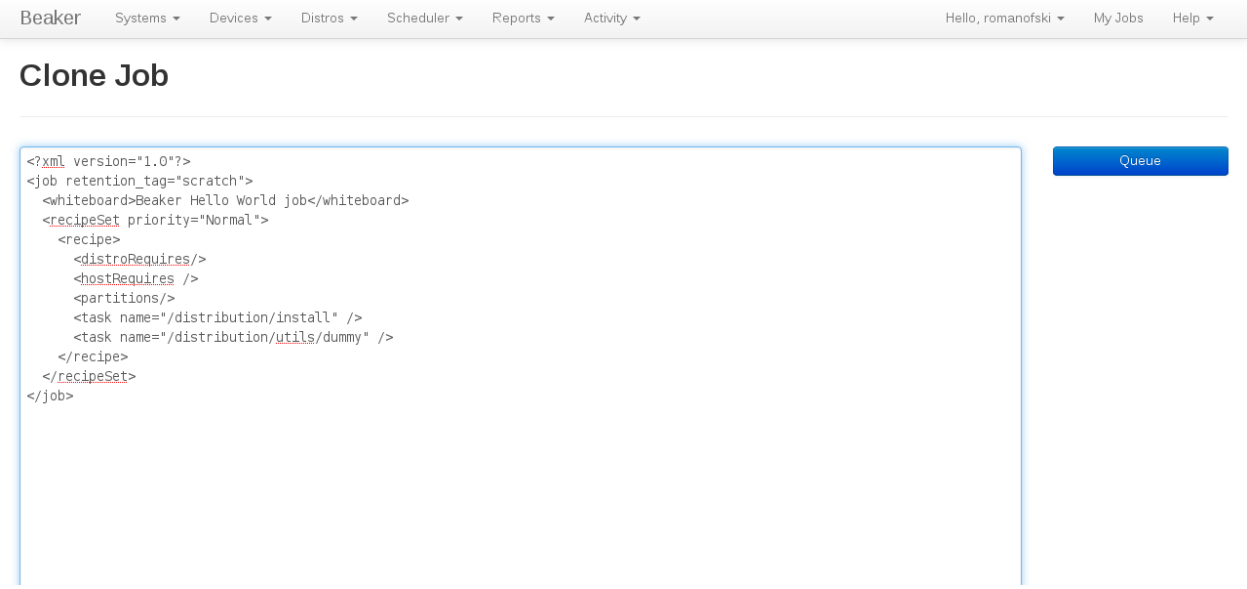


Figure 3.6: Cloning a Job

The "My Jobs" page behaves the same as the jobs page, except it is limited to jobs where you are the owner. Select *Hello → My Jobs* from the menu.

Similarly, the "My Group Jobs" page is limited to jobs submitted for any group of which you are a member. Select *Hello → My Group Jobs* from the menu.

Job results

The whole purpose of Jobs is to view the output of the Job, and more to the point, tasks that ran within the Job. To do this, you must first go to the *Job search* screen. After finding the Job you want to see the results of, click on the link in the "ID" column.You don't have to wait until the Job has completed to view the results. Of course only the results of those Tasks that have already finished running will be available.

The Job results page is divided by recipe sets. To show the results of each Recipe within these recipe sets, click the "Show All Results" button. You can just show the tasks that have a status of "Fail" by clicking "Show Failed Results."

While your Job is still queued it's possible to change the priority. You can change the priority of individual recipe sets by changing the value of "Priority", or you can change all the Job's RecipeSets at once by clicking an option beside the text "Set all RecipeSet priorities", which is at the top right of the page. If successful, a green success message will briefly display, otherwise a red error message will be shown.

Priority permissions

If you are not an Admin you will only be able to lower the priority. Admins can lower and raise the priority

---

Result Details

- *Run*
  - This is the "ID" of the instance of the particular Task.
- *Task*
  - A Task which is part of our current Job.
- *Start*

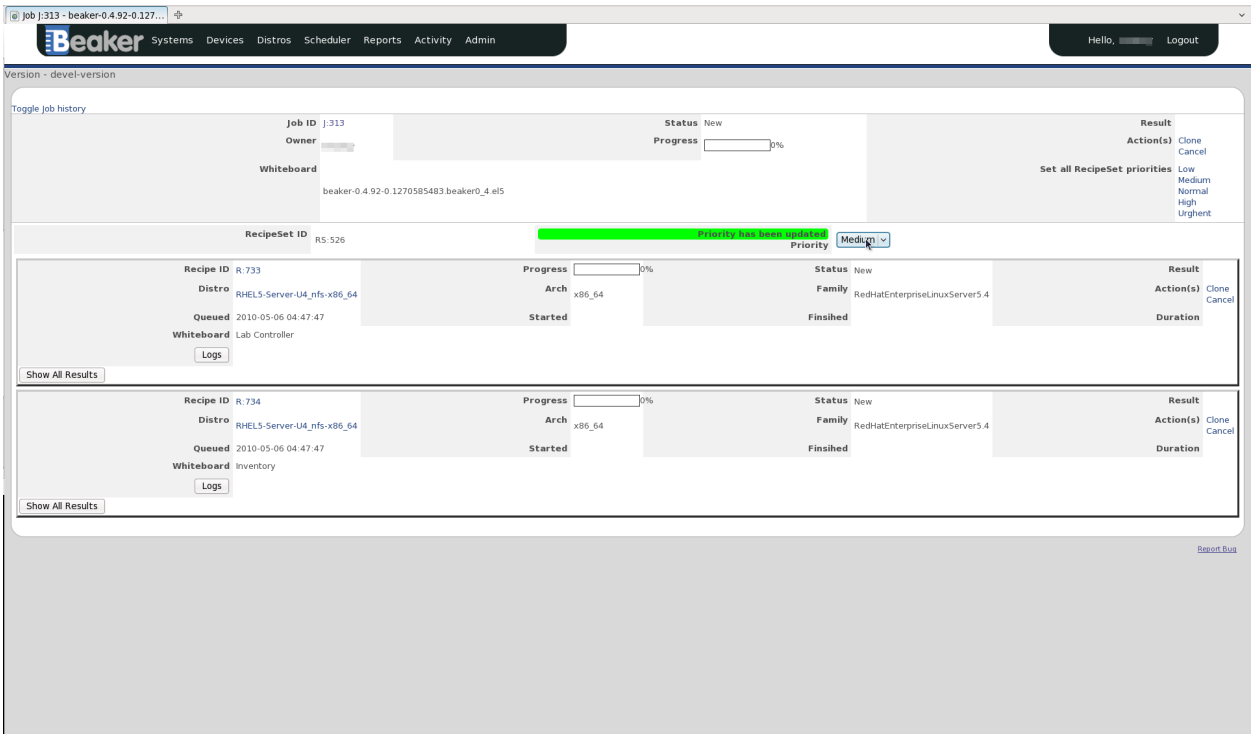


Figure 3.7: Changing the priority of a Job's RecipeSet

  - The time at which the Task commenced.
- *Finish*
  - The time at which the Task completed.
- *Duration*
  - Time the Task took to run.
- *Logs*
  - This is a listing of all the output logs generated during the running of this Task.
- *Status*
  - This is the current Status of the Task. "Aborted","Cancelled" and "Completed" mean that the Task has finished running.
- *Action*
  - The two options here are Cancel and Clone. See *Cloning an existing job* to learn about cloning.

Viewing Job results at a glance

If you would to be able to look at the Result of all Tasks within a particular Job, try the *Matrix Report*.

3.2.5 Recipes

Recipes are contained within a Job (although indirectly, as directly they are contained in a recipe set) and are themselves a container for Tasks. There can be more than one Recipe per Job. The purpose of a Recipe is to group a set of

Tasks into a single logical unit (See: *Job workflow details*).

Recipe searching

The Recipe search is accessed through the "Scheduler" at the top of the page, and clicking on the "Recipe" menu item.

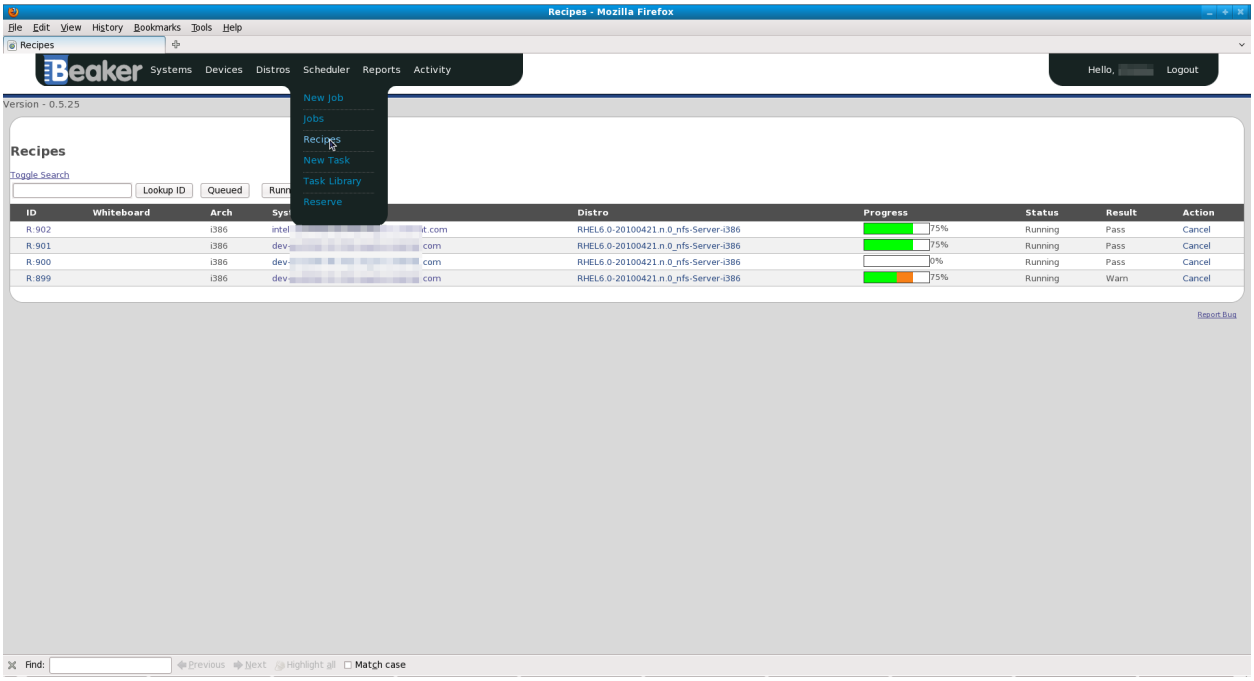


Figure 3.8: Searching for a Recipe

To look up the "Recipe ID" enter a number into the search box and press the Lookup ID button. See *System searching* for details on searching.

Quick Searches

By pressing the "Running", "Queued", or "Completed" buttons you can quickly display Recipes that have a status of running,queued, and completed respectively.

Recipe actions

At any time you wish to cancel the Recipe, you may press the "Cancel" link that is placed under the "Action" column.

3.2.6 Tasks

Task searching

To search for a task, select *Scheduler* → *Task Library* from the menu. The default search is on the "Name" property, with the "contains" operator. Other metadata search attributes are as follows:

- *Description* - This is the description provided when creating the task.

- *Version* - This is the version of the task.
- *Excluded OSMajor* - This searches the Excluded OSMajor list of the task.
- *Excluded Architecture* - This searches the Excluded Architecures list of the task.
- *Test Type* - This is the type of test when task was defined.

Refer to *Task metadata* for further details on metadata attributes.

Once you've found a particular task, you can see its details by clicking on the link in the *Name* column.

On the task page you can use the *Executed Tasks* search to search history of past executions of the task.

Uploading a task

If you already have a task packaged as an RPM, select *Scheduler → New Task* from the menu. Click the *Browse* button to locate the task RPM on your local system, and then click the *Submit Data* button to upload it. See *bkr task-add* for how to do this via the beaker client.

If you are updating an existing task, the version of the new task RPM must be higher than the existing version. This can be achieved by running `make tag` (if the task is stored in version control), or manually adjusting the `TESTVERSION` variable in the task's `Makefile` (see *Makefile variables*).

If you are uploading a new task, the rpm must contain a testinfo.desc file which contains all the mandatory fields described in the metadata section (see *Fields in testinfo.desc*).

3.2.7 Reports

Beaker offers a few different reports. They can be accessed from the Reports menu at the top of the page.

External reports

In some instances it may be preferential to provide Beaker related reports external to the Beaker server itself. If your administrator has configured any links to such reports, they will be displayed here.

Matrix report

The "Matrix" report gives a user an overall picture of results for any given Job, or number of Jobs combined. It shows a matrix of Tasks run and the Arch that they were run on. The "Reports->Matrix" is accessible from the top menu.

There are two ways of defining what Job results to display. You can select the Job by its "Whiteboard", or by its "Job ID". To show a Job's Matrix report from its Whiteboard, click on the Whiteboard text in the Whiteboard select box (or select multiple whiteboards with the *Ctrl* key). If you wish to select the Job by its ID, enter the Job ID into the "Job ID" text area. The Job Whiteboard and the Job ID are mutually exclusive when generating the Matrix report. To change between the two, click on their respective input areas. Click the "Generate" button to create the report.

Filtering Whiteboards

You can filter what is displayed in the "Whiteboard" select box by typing text into the "Filter by" field, and then clicking the *Filter* button

---

Displaying reports of any combination of jobs

Displaying the Matrix reports of any Jobs together, is possible when selecting by "Job ID". Enter in all the relevant "Job ID"s separate by whitespace or a newline.

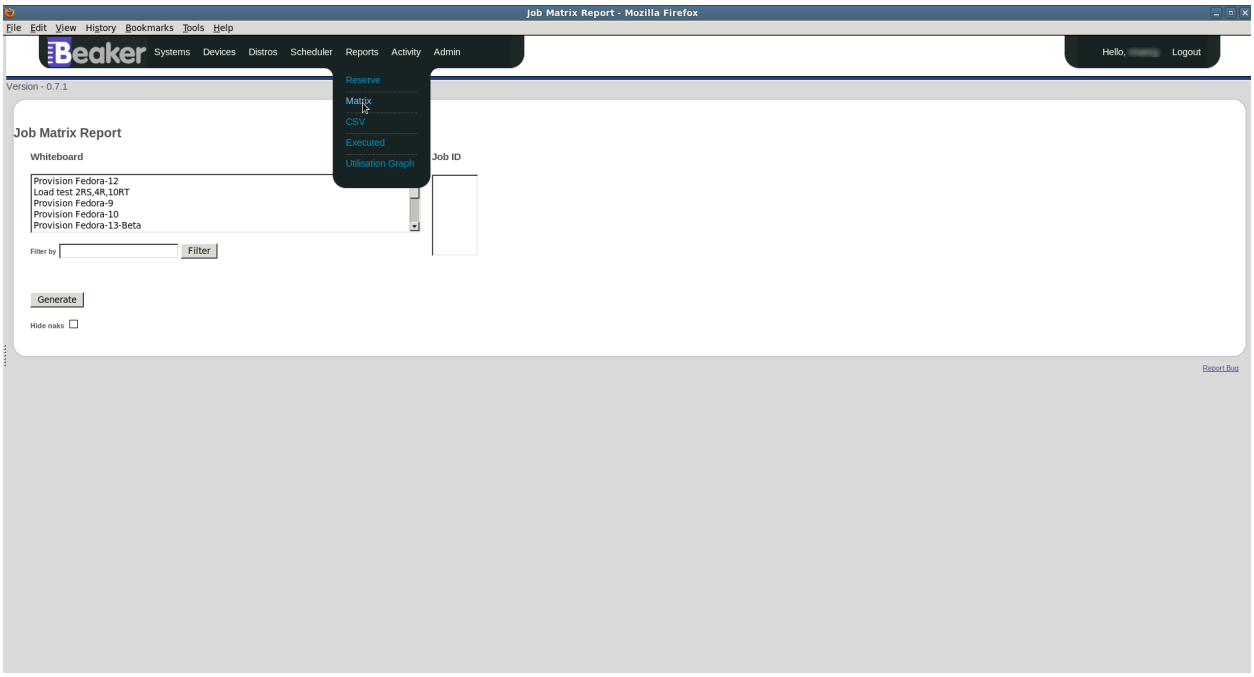


Figure 3.9: Generating a Matrix report from the Job's Whiteboard

The generated Matrix report shows the result of each Task with its corresponding Arch and Recipe Whiteboard. The points in the matrix describe the result of the Task, and how many occurrences of that result there are. Clicking on these results will take you to the "Executed Tasks" page. See *Task searching* for further details.

3.2.8 Groups

Users can be grouped together in groups. Any Beaker user can create a group and add users to it, while Beaker administrators can also define groups that are populated automatically from an LDAP directory.

Viewing groups

To view a list of all groups of which you are a member, select *Hello* → *My Groups* from the menu.

Adding a group

To add a new group, select *Hello* → *My Groups* from the menu and then click the *Add (+)* link at the bottom left. You'll then be prompted to enter a "Display Name" and a "Group Name". The former is the name that users of Beaker will see, and the latter is the name used internally. It's fine to have these names the same, or different.

Editing a group

To edit a group, select *Hello* → *My Groups* and click on the name of the group you wish to edit. From here you can add users as well as changing the group's display name and group name.

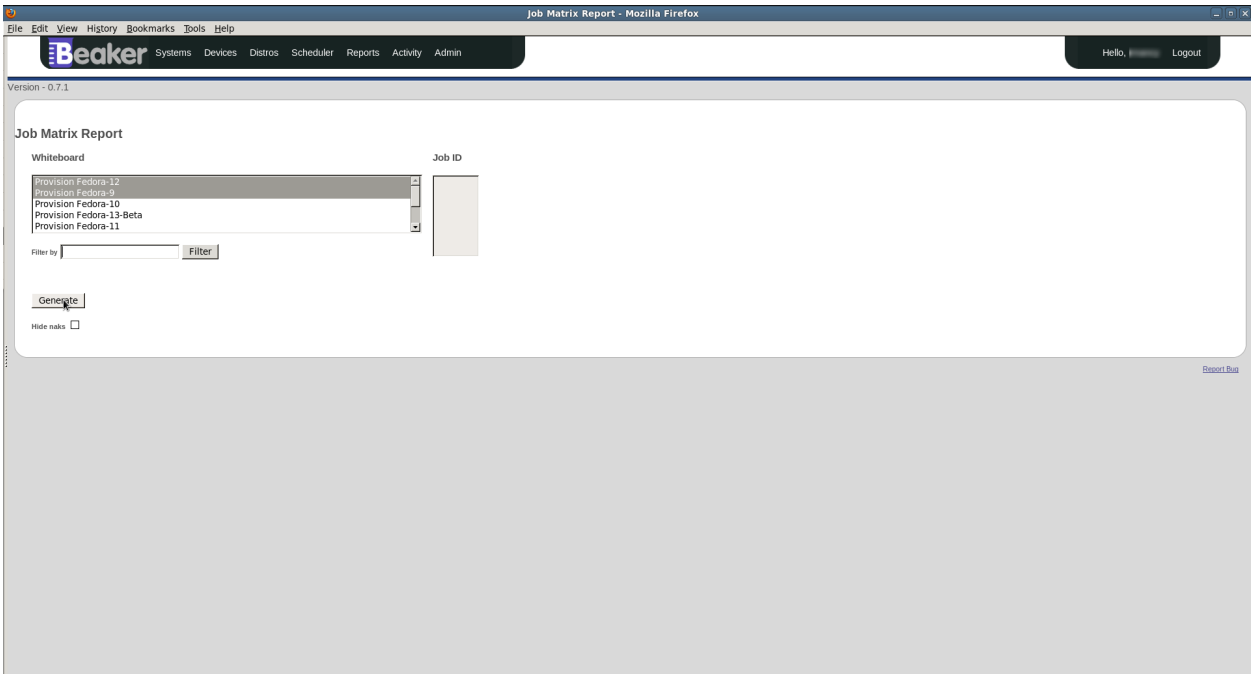


Figure 3.10: Generating a Matrix report from the Job ID

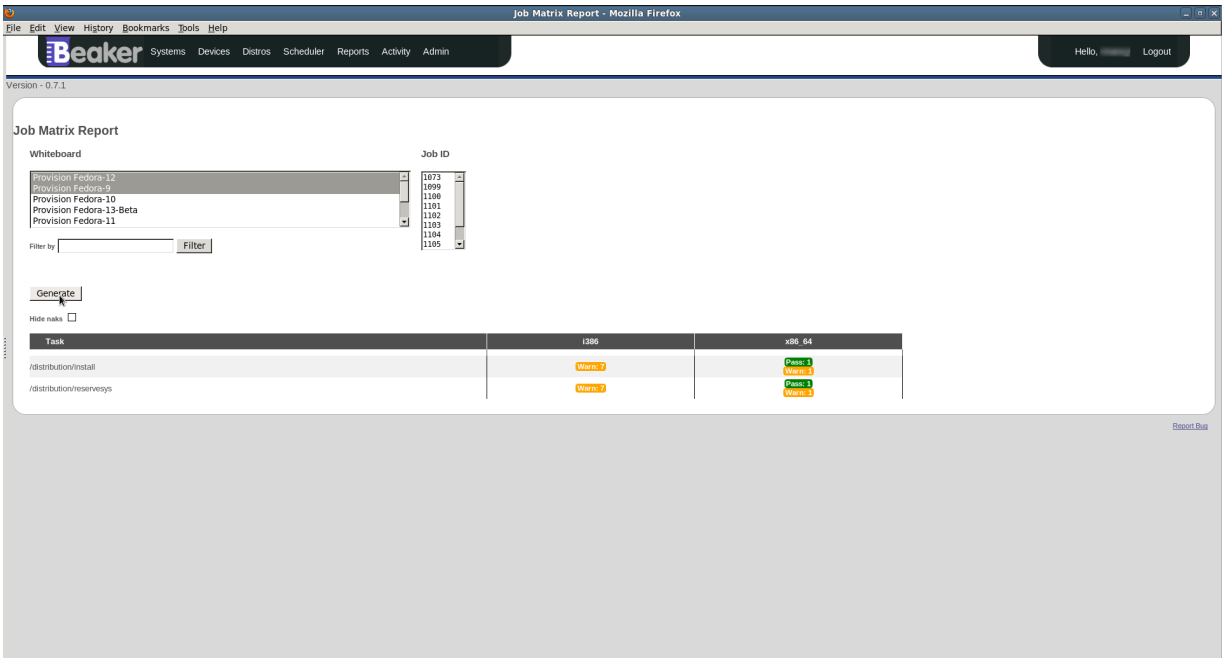


Figure 3.11: Viewing the result of one or more Jobs via the Matrix report

Group activity

To search through the historical activity of all groups, select *Activity* → *Groups* from the menu.

3.2.9 User preferences

The preferences page allows the user to configure their email address, notifications, user interface, submission delegates, SSH public keys, root password for provisioned systems and OpenStack keystone trust for running recipes on OpenStack.

Either a hashed password (in crypt format) or a cleartext password may be entered as a root password. If a plaintext password is entered, it will first be hashed before being stored. This password will be used as the root password on systems provisioned by the user.

If Beaker is configured to limit the validity of users' root passwords, the expiry date and time for your password will be shown here. After that time, you will be required to change or clear it in order to submit jobs or provision systems.

If no password is entered, the Beaker default root password will be used instead.

SSH public keys (e.g. the contents of `~/.ssh/id_rsa.pub`) may be added to a users account. These will be added to the `authorized_keys` file for the root user on provisioned hosts. Submission delegates are other users that are given the ability to submit and manage jobs on behalf of the user. This is intended primarily to grant automated tools the ability to submit and manage jobs on behalf of users, without needing access to those users' credentials, and without granting them the ability to perform other activities as that user (like managing systems or user groups). If you want to use OpenStack instances to run your recipes, you must create a Keystone trust to delegate your roles to Beaker's OpenStack account. Beaker will then use this trust to create OpenStack instances on your behalf.

3.3 Managing systems in Beaker

3.3.1 Adding your system to Beaker

To add a system, go to any system search page, then click the *Add* button at the bottom of the page. Fill in the system's details and click the *Save Changes* button to create a new record for your system. Define which architectures your system supports on the *Arches* tab. Then fill in the other details as described below.

DHCP and DNS

In order to provision the system, Beaker needs to be able to resolve its FQDN consistently. The system must have a static IPv4 address assigned by DHCP and a matching `A` record in DNS which resolves to that address. It is also recommended to add a corresponding `PTR` record so that reverse hostname lookups work correctly on the system after it is provisioned.

If the system has multiple network interfaces, ensure that the MAC address in the DHCP reservation matches the network interface which the system's firmware boots from.

DHCP option 42 ("NTP servers") must be set, so that the system's clock can be synchronized at the start of each recipe. Use a public NTP server if none are available in your lab.

The DHCP configuration must also include suitable netboot options for the system. Typically DHCP option 66 ("TFTP Server Name") is set to the address of the lab controller and option 67 ("Boot File Name") can either be set to `bootloader/`*`fqdn`*`/image` to take advantage of Beaker's custom netboot loader support or a specific bootloader such as `pxelinux.0` (see *boot-loader-images* and *boot-loader-configs*).

Power control

If the system supports remote power control, either through an out-of-band management controller or using a switchable power port, configure the details on the *Power Config* tab. Virtual machines can use the `virsh` power type.

This is a prerequisite for automatic (unattended) provisioning. If the system has no remote power control, you will have to reboot it manually when Beaker provisions it.

Boot order

For automatic provisioning, you must also configure the system's firmware to boot from the network first. Beaker will ensure that the system either boots an installer image over the network or falls back to booting from the local hard disk as appropriate. See *provisioning-process* for details about how automatic provisioning works in Beaker.

Remote console

If your system supports remote console access (serial-over-LAN or similar), you can hook it up to the conserver in your lab. This will allow Beaker to capture console logs and detect kernel panics.

Install options

Use the *Install Options* tab to set default install options for your system, if necessary. See *Install options* to learn about these settings. Refer to the Anaconda documentation for the available kernel options.

If you have connected the system's serial console to conserver, set the `console` kernel option appropriately. For example, assuming the system's serial-over-LAN device appears as the second serial port, set `console=ttyS1,115200n8`.

By default Beaker will apply `ksdevice=bootif` to the kernel options (this is defined in `/etc/beaker/server.cfg`). This setting is suitable for x86-based systems booting PXELINUX, but for other systems (including UEFI-based systems) which have more than one network interface, you must set the `ksdevice` option explicitly, otherwise Anaconda will prompt for which interface to use during installation. If only one network interface has a cable connected, you can set `ksdevice=link`. If more than one interface has a cable connected, you must nominate a specific interface to be used for installation: `ksdevice=00:11:22:33:44:55`. If you want to remove the Beaker default of ksdevice kernel option, you can precede the option with *!* within your recipe's setting, i.e, *kernel_options="!ksdevice"*.

Next steps

To test your system's configuration, try provisioning it (see *Provisioning a system*). You can watch the provisioning process through the console. Please, be patient. The provisioning may take some time.

To populate your system's hardware details in Beaker, you should *create a job* to run the Beaker-provided */distribution/inventory* task on the machine. The easiest way to do this is to use the *bkr machine-test* command to generate and submit an appropriate job definition:

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

Once your system is operational, you may want to use Beaker's *system sharing features* to let others use or administer your system.

3.3.2 Sharing your system with others

By default, when a new system is added to Beaker only the owner has access to use it. You can use loans to temporarily grant another user exclusive access to a system, or set access policy rules for fine-grained control over which Beaker users can use or administer the system.

Loans

Loaning a system to another user gives them exclusive access to reserve the system. While the system is loaned, no other users are permitted to reserve the system, even if they would normally have access to reserve it.

When loaning a system, you can optionally add a comment to record the reason for the loan. When the loan is returned, the comment is automatically cleared.

To loan your system, go to the system page and click the *Loan Settings* button in the *Loaned to* field.

Access policies

You can apply an access policy to your system, in order to grant other Beaker users limited access to use it.

Access policies are represented as a set of rules. Each rule grants a particular permission to a particular user or group. The effective permissions for a user is the union of the permissions granted to them directly and to all of the groups they are a member of.

To apply an access policy to your system, go to the system page and click the *Access Policy* tab, or use the *bkr policy-grant* client command.

In Beaker's web UI access policies are displayed as a table of checkboxes, where each column is a permission and each row is a user or group covered by the policy. When a checkbox is selected it represents a rule granting the permission in the column to the user or group in the row.

The following permissions can be granted in an access policy:

Name	Label	Description
view	View	The user can view the system and find it in search results. If the system is loaned to a user, they are permitted to view it for the duration of the loan. If a user is not permitted to view the system, they cannot interact with it in any way and any other permissions granted to them have no effect.
view_power	View power settings	The user can view the system's power settings. This is a separate permission, and is not granted to all users by default, so that system owners can avoid disclosing the power password for their system.
edit_policy	Edit this policy	The user can edit the access policy to grant or revoke permissions, including adding new users and groups to the policy. Users with permission to edit the policy can grant themselves any of the other permissions and also change the owner of the system.
edit_system	Edit system details	The user can edit system details and configuration, however they cannot take ownership of it or grant new permissions to themselves or any other user.
loan_any	Loan to anyone	The user can lend the system to any Beaker user, including themselves. Users with this permission can also return and update any existing loan, as well as return other user's manual reservations.
loan_self	Loan to self	The user can borrow the system by loaning it to themselves.
control_system	Control power	The user can run power commands and netboot commands for the system *even when they have not reserved it*.
reserve	Reserve	The user can reserve the system, either through the scheduler (if the system is Automated) or manually through the web UI (if the system is Manual).

In Beaker's web UI, the human-friendly label is used to identify permissions. In the command-line client, the symbolic name is used.

The owner of the system is implicitly granted all of the above permissions, and also has the ability to change the system owner.

Administrators of the Beaker instance are implicitly granted all of the same permissions as the system owner except the ability to reserve the system (this ensures admins don't accidentally run automated jobs on arbitrary systems). If an administrator needs to reserve a system and do not already have access to do so, they must first loan it to themselves or grant themselves the relevant permission.

System reservations made through the automated scheduler can only be terminated by cancelling the relevant job rather than by returning the system directly through the web UI or command-line client.

Shared access policies

In addition to setting the access policy on a system directly, you can share the same policy across many systems by using a *pool access policy*.

Start by creating a new system pool, or pick an existing one. Add all the systems as members of the pool. You can then configure each system to use the pool's access policy.

You can add systems to a pool on the *pool page* or by using *bkr pool-add*.

You can set a system to use a pool policy on the *Access Policy* tab of the system page or by specifying the `--pool-policy` option to **bkr system-modify**.

You can update a pool's access policy on the pool page or by specifying the `--pool` option to **bkr policy-grant** and **bkr-policy-revoke**.

Notify CC list

Beaker sends e-mail notifications to the system owner when it detects a problem with the system (see *Broken system detection*) or when a user reports a problem or requests a loan.

You can add one or more e-mail addresses to the notify CC list for your system. Any Beaker notifications about the system will also be sent to those addresses.

The notify CC list does not itself grant any extra permissions over a system. If someone else is helping maintain your system, you may also want to grant them edit_system or loan_any permissions so that they can update your system as needed.

3.3.3 Broken system detection

The Beaker scheduler includes some heuristics to automatically set a system's condition to Broken if the system appears to be unable to successfully run recipes. The system owner is notified by email so that they can take corrective action before setting the system's condition back to Automated.

The feature is intended to prevent a misconfigured or faulty system from ruining a large number of jobs if the system owner isn't able to immediately correct the problem or remove the system from service. However, the heuristics used are very conservative, to avoid false positives caused by a distro bug or a mistake in a Beaker task.

Beaker will set a system's condition to Broken under the following circumstances:

- when a power command for the system fails

Failed power commands are usually caused by incorrect power settings, connectivity problems, or a faulty management controller. In any case, if the system cannot be powered then it cannot execute recipes, so in this case it is always marked broken immediately.

- when there is a run of suspicious aborted recipes

  A recipe is considered "suspicious" (that is, indicating the system might be broken) if all tasks in the recipe are aborted.

  It is considered to be a "run" if there has been two or more consecutive suspicious recipes, with no intervening non-suspicious recipes and no manual changes made to the system's status.

  To reduce the risk of false positives, this heuristic is only applied when the aborted recipes used a "reliable" distro. A distro is considered reliable if it is tagged with the tag given in the `beaker.reliable_distro_tag` config setting. The suggested tag is `RELEASED`. If the `beaker.reliable_distro_tag` setting is unset, this heuristic is not used.

  The run of suspicious recipes must also use at least two different distros in order to trigger this heuristic. This is to reduce the chance that the aborted was caused by a distro bug rather than a problem with the system.

3.4 Job design

The goal of this section is to help you translate the material needs of your particular use case into a suitably designed Beaker job. This means taking the business, technical and management requirements and solving these within the various levels of the job schema.

3.4.1 Access control for jobs

When submitting a job, you can optionally submit it on behalf of a group, or on behalf of another user.

By default, only the submitter can modify job attributes (retention tag, product, whiteboard, priority, ack/nak), cancel the job, or delete the job.

However, when you submit a job on behalf of a group, members of that group have full control over the job. All members of the group will also have SSH based access to the systems used by the job. To learn how to set the group value, see *Job workflow details*.

Submitting a job on behalf of another user means that, aside from allowing the submitter to retain administrative access to the job, Beaker will treat the job as being owned by the named user (the user named in the user attribute) rather than the submitter. Since this gives the submitter access to systems for scheduling purposes as if they were the named user, this is only permitted if the submitter has been *configured as a submission delegate* by that user. This is intended primarily to grant automated tools the ability to submit and manage jobs on behalf of users, without needing access to those users' credentials, and without granting them the ability to perform other activities as that user (like managing systems or user groups).

3.4.2 Log archiving

Preserving log files indefinitely can consume an undesirable amount of space. This behaviour can be controlled by selecting the appropriate "retention tag" setting. Beaker ships with the following default retention tags:

- `scratch`: preserve logs for 30 days
- `60days`: preserve logs for 60 days
- `120days`: preserve logs for 120 days
- `active`: preserve as long as associated product is active

- `audit`: preserve indefinitely (no automatic deletion)

The log deletion utility provided with Beaker can automatically handle deletion of logs for jobs using any of the first three retention tags.

The last two retention tags require that the job be associated with a specific "Product". Product identifiers are treated as an opaque string by Beaker - these two tags are intended for use in conjunction with external tools and processes that are able to determine when a product is no longer active or when an audit has been completed.

3.5 Reserving a system after testing

After the tests have completed, it may be desirable to reserve the test system to collect any additional logs, perform any additional testing or verify the test results manually. Reserving a system prevents Beaker from claiming back the system automatically and can be accomplished using one of the following methods.

3.5.1 Using the `/distribution/reservesys` task

When the */distribution/reservesys task* is added to a recipe, it will reserve the system until the system is explicitly returned or the reservation duration expires. If the reservation is explicitly returned, then execution resumes with the next task, or, if there are no remaining tasks, the recipe is marked as `Completed`. If the reservation duration instead expires, then the recipe is aborted entirely and no further tasks are executed.

An example recipe skeleton is as follows:

```
<recipe>
  ..
  <task name='/distribution/mytask1'/>
  <task name='/distribution/mytask2'/>
  <task name='/distribution/reservesys'/>
  ..
</recipe>
```

The above will reserve the system after the two tasks, `/distribution/task1` and `/distribution/task2` have finished execution. The recipe status will be "Running" during the duration of the system being reserved.

Configuring the reservation behavior

The reservation behavior can be configured with the help of the following task parameters:

- `RESERVE_IF_FAIL`: If this parameter is defined then the result of the recipe is checked. The system is reserved, *only* if the recipe did not pass. If it passed, this "test" is reported back to Beaker as a pass and the next task in the recipe (if any) begins execution. The parameter should be defined as: `<param name="RESERVE_IF_FAIL" value= "True" />` (While *any* non-empty string will have the same effect, using the string `"True"` is strongly recommended). If you want to reserve the system irrespective of the result of the test, do not specify this variable at all.
- `RESERVETIME`: Using this parameter, you can define the duration (in seconds) for which you want to reserve the system up to a maximum of 356400 seconds (99 hours). If this variable is not defined, the default reservation is for 86400 seconds (24 hours). You can return the system early as described in *Returning early and extending reservation*.

For example, to define both these parameters when you specify the task in your job description, you would use the following:

```
<task name="/distribution/reservesys" role="STANDALONE">
  <params>
    <param name="RESERVE_IF_FAIL" value="True" />
    <param name="RESERVETIME" value="172800" />
  </params>
</task>
```

Note: Due to an unfortunate race condition, conditional reservation may be unreliable if the immediately preceding task is the only one that fails in the recipe. Inserting */distribution/utils/dummy* prior to this task may help if the problem of failing to reserve the system occurs regularly.

Notification

Depending on whether you set the `RESERVE_IF_FAIL` parameter appropriately and its implications (as described), once the system has been reserved, you will receive an email with the subject "[Beaker Machine Reserved] test-system.example.com" and other information. This is a notification that the system has now been reserved and you can connect to it (using an SSH client, either using the username and password or your public key).

The notification email is sent from the test system. This implies that in case of a problem in network connectivity between your email server and the test system, the notification email will not be received.

Returning early and extending reservation

The email also includes information about returning the system early or extending your reservation time. To return the system early, execute **return2beaker.sh** from your terminal (after you have logged in to the system).

To extend the reservation time, execute **extendtesttime.sh** and enter the desired extension to the reservation (relative to the current time).

Changes to the test system environment

Besides creating the above scripts on the test system in `/usr/bin`, the task also sets up a custom message in `/etc/motd`. When you login to the reserved system, you will be greeted with a custom message and you will find the two scripts accessible from your shell. These changes are in addition to the RPM packages installed to meet the dependencies for the task.

3.5.2 Using the `<reservesys/>` element

New in version 0.17.0.

If this element is added to a recipe, it will reserve the system after all the tasks have finished execution (or when the recipe is aborted as described below). By default, it reserves the system for 86400 seconds (or 24 hours), but this can be changed using the `duration` attribute. For example, `<reservesys duration="3600"/>` will reserve the system for an hour.

An example recipe skeleton using this approach is as follows:

```
<recipe>
  ..
  <task name='/distribution/mytask1'/>
  <task name='/distribution/mytask2'/>
  <reservesys/>
```

```
  ..
</recipe>
```

After both the tasks have finished execution, the system will be reserved. The recipe status will be "Reserved" during the duration of the system being reserved.

You can also conditionally reserve the system at the end of your recipe by using the attribute `when=""`, with the following values:

`onabort` The system will be reserved if the recipe status is Aborted.

`onfail` The system will be reserved if the recipe status is Aborted, or the result is Fail.

`onwarn` The system will be reserved if the recipe status is Aborted, or the result is Fail or Warn. This corresponds to the existing `RESERVE_IF_FAIL=1` option for the `/distribution/reservesys` task.

`always` The system will be reserved unconditionally.

If this element is given without a `when=""` attribute, it defaults to `when="always"`, matching the behaviour from previous Beaker versions.

The advantage of using this approach is that this will also reserve the system under abnormal circumstances which cause the recipe to be aborted. Circumstances in which this may happen include a hung task, installation failures, kernel panics, the test harness rendered non-functional for some reason and others. Thus, this is a more robust way of reserving a system.

Notification email

Once the system is reserved, an email notification will be sent to the job owner with the subject "[Beaker System Reservation] System: test-system.example.com" The email content is slightly different from the previous case, but includes similar information such as the hostname of the system reserved, the distribution provisioned on the system, how to return the reservation and others. The notification email is sent from the Beaker server, and hence any abnormal condition on the test system doesn't affect this.

Returning early and extending reservation

Once a system is reserved, the remaining duration is shown at the top of the recipe page with the label *Remaining watchdog time*. To return the system, click the *Return the reservation* button on the *Reservation* tab.

On the *Reservation* tab, you can extend the reservation by clicking on the *Extend the reservation* button and enter how much time the reservation should be extended by. You can also extend it by using the **bkr watchdog-extend** command.

Changes to the test system environment

As a consequence of the fact that system reservation using this method is completely external to the test system, the test system will be in a state exactly what it was at the end of executing the last task when it's reserved. A standard Beaker test system message is displayed when you login.

3.6 Workflow

3.6.1 Job XML

You can specify Beaker jobs using an XML file. This allows users a more consistent, maintainable way of storing and submitting jobs. You can specify and save entire jobs, including many recipes and recipe sets in them in XML files and save them as regression test suites and such.

The various elements along with their attributes and the values they can take are described in the RELAX NG schema described in the file beaker-job.rng.

Job workflow details

There are various XML entities in the job definitions created for a workflow. Each job has a root node called the job element:

```
<job group='product-QA'>
</job>
```

The `group` attribute is an optional attribute that indicates the job is being submitted on behalf of a particular group, and will allow all members of the group full access to manipulate the job.

A direct child is the "whiteboard" element. The content is normally a mnemonic piece of text describing the job, and can also be used to generate *matrix reports* that cover multiple jobs:

```
<job group='product-QA'>
<whiteboard>
        Apache 2.2 test
</whiteboard>
</job>
```

The next element is the "recipeSet" (which describes a recipe set. See *Recipes* for full details). A job workflow can have one or more of these elements, which contain one or more "recipe" elements. Whereas tasks within a recipe are run in sequence on a single system, all recipes within a recipe set are run simultaneously on systems controlled by a common lab controller. This makes recipe sets useful for scheduling multihost jobs, where recipes playing different roles (e.g. client, server) run concurrently on separate systems.

When multiple recipe sets are defined in a single job, they are run in no predetermined order, are not necessarily scheduled concurrently and may run on systems controlled by different lab controllers. The advantage of combining them into one job is that they will report a single overall result (as well as a result for each recipe set) and can be managed (e.g. submitted, cancelled) as a single unit.

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
    </recipeSet>
</job>
```

As noted above, the "recipeSet" element contains "recipe" elements. Individual recipes can have the following attributes:

`kernel_options`, `kernel_options_post`, `ks_meta` Install options for this recipe. See *Install options*.

`role` In a multihost environment, it could be either `SERVERS`, `CLIENT` or `STANDALONE`. If it is not important, it can be `None`.

whiteboard Free-form text which describes the recipe.

Here is an example:

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
      </recipe>
    </recipeSet>
</job>
```

Avoid having many recipes in one recipe set

Because recipes within a recipe set are required to run simultaneously, no recipe will commence execution until all other sibling recipes are ready. This involves each recipe reserving a system, and waiting until every other recipe has also reserved a system. This can tie up resources and keep them idle for long amounts of time. It is thus worth limiting the recipes in each recipe set to only those that actually need to run simultaneously (i.e multihost jobs)

Within the `recipe` element, you can specify what packages need to be installed on top of anything that comes installed by default.

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
        <packages>
          <package name="emacs"/>
          <package name="vim-enhanced"/>
          <package name="unifdef"/>
          <package name="mysql-server"/>
          <package name="MySQL-python"/>
          <package name="python-twill"/>
                    </packages>
      </recipe>
    </recipeSet>
</job>
```

If you would like you can also specify your own repository that provides extra packages that your job requires. Use the `repo` tag for this. You can use any text you like for the name attribute.

```
<job group='product-QA'>
 <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
        <packages>
         <package name="emacs"/>
          <package name="vim-enhanced"/>
          <package name="unifdef"/>
          <package name="mysql-server"/>
          <package name="MySQL-python"/>
          <package name="python-twill"/>
        </packages>
```

```
        <repos>
          <repo name="myrepo_1" url="http://my-repo.com/tools/beaker/devel/"/>
        </repos>

      </recipe>
    </recipeSet>
</job>
```

By default the Beaker watchdog will abort a recipe if it detects a kernel panic message on the system's console. It will also abort the recipe if it detects a fatal installer error during the installation. You can control this behaviour using the `<watchdog/>` element. If you want to disable panic detection, for example because your tests are expecting to trigger a kernel panic, add an attribute `panic="ignore"` to the `<watchdog/>` element.

To actually determine what distro will be installed, the `<distroRequires/>` needs to be populated. Within, we can specify such things as as `<distro_arch/>`, `<distro_name/>` and `<distro_method/>`. This relates to the Distro architecture, the name of the Distro, and it's install method (i.e nfs,ftp etc) respectively. The `op` determines if we do or do not want this value i.e = means we do want that value, `!=` means we do not want that value.

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
        <packages>
          <package name="emacs"/>
          <package name="vim-enhanced"/>
          <package name="unifdef"/>
          <package name="mysql-server"/>
          <package name="MySQL-python"/>
          <package name="python-twill"/>
        </packages>

        <repos>
          <repo name="myrepo_1" url="http://my-repo.com/tools/beaker/devel/"/>
        </repos>
        <distroRequires>
          <and>
            <distro_arch op="=" value="x86_64"/>
            <distro_name op="=" value="RHEL5-Server-U4"/>
            <distro_method op="=" value="nfs"/>
          </and>
        </distroRequires>
      </recipe>
    </recipeSet>
</job>
```

`<hostRequires/>` has similar attributes to `<distroRequires/>`

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
        <packages>
           <package name="emacs"/>
          <package name="vim-enhanced"/>
```

```
          <package name="unifdef"/>
          <package name="mysql-server"/>
          <package name="MySQL-python"/>
          <package name="python-twill"/>
        </packages>
        <repos>
          <repo name="myrepo_1" url="http://my-repo.com/tools/beaker/devel/"/>
        </repos>
        <distroRequires>
          <and>

            <distro_arch op="=" value="x86_64"/>
            <distro_name op="=" value="RHEL5-Server-U4"/>
            <distro_method op="=" value="nfs"/>
          </and>
        </distroRequires>
        <hostRequires>
          <and>
            <arch op="=" value="x86_64"/>
            <hypervisor op="=" value=""/>
          </and>
        </hostRequires>
      </recipe>
    </recipeSet>
</job>
```

Bare metal vs hypervisor guests

Beaker supports direct provisioning of hypervisor guests. These hypervisor guests live on non volatile machines, and can be provisioned as a regular bare metal system would. They look the same as regular system entries, except their `Hypervisor` attribute is set. If your recipe requires a bare metal machine, be sure to include <hypervisor op="=" value=""/> in your <hostRequires/>

If your recipe requires the presence of a specific device on the host, you may specify that using the `<device>` element (within `<hostRequires>`) using a syntax such as:

```
<device op="=" type="network" />
```

The above device specification will try to find a host which has a network card to run your recipe on. If you wanted that the network card should be from a specific vendor, you would specify it, like so:

```
<device op="=" type="network" vendor_id="8086" />
```

The other possible values of `type` include (but are not limited to): `cpu`, `display`, `scsi`, `memory` and `usb`. There are a number of other attributes that you can use to specify a device: `bus`, `driver`, `device_id`, `subsys_vendor_id`, `subsys_device_id` and `description`.

The `op` attribute can take one of the four values: `!=`, `like`, `==`, `=`, with the last two having serving the same functionality. The `!=`, `=` and `==` operators should be used when you want an exact match of your device specification. For example, if to ask Beaker to run your recipe on a host with *no* USB device, you would use the following specification:

```
<device op="!=" type="USB" />
```

On the other hand, if you are only partially sure about what the device specification you are looking for, you would use the `like` operator. For example, the following specification will try to find a host with a graphics controller:

```
<device op="like" description="graphics"/>
```

You can of course combine more than one such <device> elements. The next example shows an entire <hostRequires> specification:

```
<hostRequires>
  <and>
    <system_type op="=" value="Machine"/>
    <device op="=" type="network" description="Extreme Gigabit Ethernet" />
    <device op="=" type="video" description="VD 0190" />
  </and>
</hostRequires>
```

The above specification will try to find a host which is a Machine with a network interface (with description as "Extreme Gigabit Ethernet") and with a video device with the description as "VD 0190".

If you want your recipe to run on a particular system and you know its FQDN, you can configure host filtering by setting hostname and assign FQDN to it. The job will run on that machine provided it is in available state. The following example allows you to configure a machine with a specific host name:

```
<hostRequires>
  <and>
    <system_type op="=" value="Machine"/>
    <hostname op="=" value="my.hostx123.example.com"/>
  </and>
</hostRequires>
```

Another option to using hostname is entering wildcard '%' syntax in the name for chosing system(s):

```
<hostRequires>
  <and>
    <system_type op="=" value="Machine"/>
    <hostname op="like" value="my.%hostx%"/>
  </and>
</hostRequires>
```

Inventoried Systems Only

It is worthwhile to note here that if you submit device specifications in your <hostRequires>, Beaker will match the specifications against the current inventory data it has for the systems. For this data to be available for a system, it is necessary that the *Inventory task* has been run on it at some point of time before your job specification has been submitted. What this basically means is that unless a system has been inventoried, Beaker won't be able to find it, even if it has the particular device you are requesting. It's a good idea to first search if there is any system at all with the device you want to run your recipe on. (See: *System searching*).

Warning: There is an ability to force a job to run on a specific system. This capability is intended for administrators to perform troubleshooting. It will cause the job to run on a machine even if the system is in *broken, manual, or excluded* condition. This is not the desired behavior for the majority users so this configuration should be avoided. Use of force= configuration is documented below but it's intended for use by system administrators.

To force your recipe to run on a particular system and you know its FQDN, skip the host filtering described earlier and force the scheduler to pick a particular system for your recipe using the force="" attribute. For example, the following XML will force the recipe to be scheduled on my.host.example.com:

```
<hostRequires force="my.host.example.com" />
```

When the force="" attribute is present, the scheduler will use the named system even if its condition is set to Broken or Manual.

The `force=""` attribute is mutually exclusive with other host filtering criteria. It is invalid to specify both in `<hostRequires/>`.

All that's left to populate our XML with, are the 'task' elements. The two attributes we need to specify are the `name` and the `role`. You can find which tasks are available by *searching the task library*. Also note that we've added in a `<param/>` element as a descendant of `<task/>`. The `value` of this will be assigned to a new environment variable specified by `name`.

```
<job group='product-QA'>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
    <recipeSet>
      <recipe kernel_options="" kernel_options_post="" ks_meta="" role="None" whiteboard="Lab Control
        <packages>
          <package name="emacs"/>
          <package name="vim-enhanced"/>
          <package name="unifdef"/>
          <package name="mysql-server"/>
          <package name="MySQL-python"/>
          <package name="python-twill"/>
        </packages>

        <repos>
          <repo name="myrepo_1" url="http://my-repo.com/tools/beaker/devel/"/>
        </repos>
        <distroRequires>
          <and>
            <distro_arch op="=" value="x86_64"/>
            <distro_name op="=" value="RHEL5-Server-U4"/>
            <distro_method op="=" value="nfs"/>
          </and>
        </distroRequires>

        <task name="/distribution/check-install" role="STANDALONE">
          <params>
                <param name="My_ENV_VAR" value="foo"/>
           </params>
         </task>

      </recipe>
    </recipeSet>
</job>
```

By default, the kickstart fed to Anaconda is a generalized kickstart for a specific distro major version. However, there are a couple of ways to pass in a customized kickstart.

One method is to pass the `ks` key/value to the `kernel_options` parameter of the `recipe` element. Using this method the kickstart will be used by Anaconda unaltered.

```
<recipe kernel_options='ks=http://example.com/ks.cfg' />
```

Alternatively, the kickstart can be written out within the `recipe` element.

```
<kickstart>
  install
  key --skip
  lang en_US.UTF-8
  skipx
  keyboard us
```

```
  network --device eth0 --bootproto dhcp
  rootpw --plaintext testingpassword
  firewall --disabled
  authconfig --enableshadow --enablemd5
  selinux --permissive
  timezone --utc Europe/Prague

  bootloader --location=mbr --driveorder=sda,sdb
# Clear the Master Boot Record
  zerombr
# Partition clearing information
  clearpart --all --initlabel
# Disk partitioning information
  part /RHTSspareLUN1 --fstype=ext3 --size=20480 --asprimary --label=sda_20GB --ondisk=sda
  part /RHTSspareLUN2 --fstype=ext3 --size=1 --grow --asprimary --label=sda_rest --ondisk=sda
  part /boot --fstype=ext3 --size=200 --asprimary --label=BOOT --ondisk=sdb
# part swap --fstype=swap --size=512  --asprimary --label=SWAP_007 --ondisk=sdb
  part / --fstype=ext3 --size=1 --grow --asprimary --label=ROOT  --ondisk=sdb

  reboot

  %packages --excludedocs --ignoremissing --nobase
</kickstart>
```

When passed a custom kickstart in this manner, Beaker will add extra entries into the kickstart. These will come from install options that have been specified for that system, arch and distro combination; partitions, packages and repos that have been specified in the `recipe` element; the relevant snippets needed for running the harness. For further information on how Beaker processes kickstarts and how to utilize their templating language, see *kickstarts*.

3.6.2 Virtualization workflow

The virtualization testing framework in Beaker utilizes libvirt tools, particularly virt-install program to have a framework abstracted from the underlying virtualization technology of the OS. The crux of the virtualization test framework is a *guest recipe*. Each virtual machine is defined in its own `<guestrecipe/>` element and the guest recipes are a part of the host's recipe. To illustrate, let's say, we would like to create a job that will create a host and 2 guests, named guest1 and guest2 respectively. The skeleton of the recipe will look like this:

```
<recipe>
        ...

        <guestrecipe guestname=guest1 ...>
          ...
          (guest1 test recipe)
          ...
        </guestrecipe>
        <guestrecipe guestname=guest2 ...>
          ...
          (guest2 test recipe)
          ...
        </guestrecipe>
       ...
</recipe>
```

Here is a complete job description corresponding to the above skeleton:

```
<job retention_tag="scratch">
  <whiteboard>
```

```
  An example job for testing Virtualization Workflow
</whiteboard>
<recipeSet priority="Normal">
  <recipe role="RECIPE_MEMBERS" whiteboard="">
    <guestrecipe guestargs="--ram=1024 --vcpus=2 --file-size=20 --kvm" guestname="guest1" role="RE
      <autopick random="false"/>
      <watchdog panic="None"/>
      <packages/>
      <ks_appends/>
      <repos/>
      <distroRequires>
        <and>
          <distro_name op="=" value="RHEL-6.3"/>
          <distro_variant op="=" value="Server"/>
          <distro_arch op="=" value="x86_64"/>
        </and>
      </distroRequires>
      <hostRequires>
        <system_type value="Virtual"/>
      </hostRequires>
      <partitions/>
      <task name="/distribution/check-install" role="None"/>
    </guestrecipe>
    <guestrecipe guestargs="--ram=1024 --vcpus=2 --file-size=20 --kvm" guestname="guest2" role="RE
      <autopick random="false"/>
      <watchdog panic="None"/>
      <packages/>
      <ks_appends/>
      <repos/>
      <distroRequires>
        <and>
          <distro_name op="=" value="RHEL-6.3"/>
          <distro_variant op="=" value="Server"/>
          <distro_arch op="=" value="x86_64"/>
        </and>
      </distroRequires>
      <hostRequires>
        <system_type value="Virtual"/>
      </hostRequires>
      <partitions/>
      <task name="/distribution/check-install" role="None"/>
    </guestrecipe>
    <autopick random="false"/>
    <watchdog panic="ignore"/>
    <packages/>
    <ks_appends/>
    <repos/>
    <distroRequires>
      <and>
        <distro_name op="=" value="RHEL-6.3"/>
        <distro_variant op="=" value="Server"/>
        <distro_arch op="=" value="x86_64"/>
      </and>
    </distroRequires>
    <hostRequires>
      <system_type value="Machine"/>
      <key_value key="HVM" op="=" value="1"/>
    </hostRequires>
```

```
      <partitions/>
      <task name="/distribution/check-install" role="STANDALONE"/>
      <task name="/distribution/virt/install" role="STANDALONE"/>
      <task name="/distribution/virt/start" role="STANDALONE"/>
      <task name="/distribution/reservesys" role="STANDALONE"/>
    </recipe>
  </recipeSet>
</job>
```

The above job sets up two guest systems `guest1` and `guest2` and runs the `/distribution/check-install` task in each of them to indicate whether or not the installation worked and upload the relevant log files.

Anything that can be described inside a recipe can also be described inside a guest recipe. This allows the testers to run any existing Beaker test inside the guest just like it'd be run inside a baremetal machine.

Guest console logging

The contents of the guest's console log depends on what Operating System the host is running. Anything from Red Hat Enterprise Linux 5 (or equivalent) and up (except 5.3) will log the console output from the start of installation. Earlier versions will not.

If the guest is running Red Hat Enterprise Linux 6, the console logging will be directed to both ttyS0 and ttyS1

When Beaker encounters a guestrecipe it does create an environmental variable to be passed on to virtinstall test. The tester-supplied elements of this variable all come from the guestrecipe element. Consequently, it's vital that the tester fully understand the properties of this element. guestrecipe element guestname and guestargs elements. guestname is the name of the guest you would like to give and is optional. If you omit this property then the Beaker will assign the hostname of the guest as the name of the guest. guestargs is where you define your guest. The values given here will be same as what one would pass to virt-install program with the following exceptions:

- Name argument must not be passed on inside guestargs. As mentioned above, it should be passed with guestname property..
- Other than name , -mac, -location, -cdrom (-c) , and -extra-args ks= must not be passed. Beaker does those based on distro information passed inside the guestrecipe.
- In addition to what can be passed to virt-install, extra arguments -lvm or -part or -kvm can also be passed to guestargs, to indicate lvm-based or partition- based guests or kvm guests (instead of xen guests).
- If neither one of -lvm or -part options are given, then a filebased guest will be installed. If -kvm option is not given then xen guests will be installed. See below for lvm-,partition-based guests section for more info on this topic.
- The virtinstall test is very forgiving for the missed arguments, it'll use some default when it can. Currently these arguments can be omitted:
  - -ram or -r , a default of 512 is used
  - 1.-nographics or -vnc, if the guest is a paravirtualized guest, then -nographics option will be used, if the guest is an hvm guest, then -vnc option will be used.
  - 1.-file-size or -s, a default of 10 will be used.
  - -file or -f, if the guest is a filebased guest, then the default will be /var/lib/xen/images/${guestname}.img . For lvm-based and block-device based guest, this option MUST be provided.

KVM vs. Xen guests

Starting with RHEL 5.4, both Xen and KVM hypervisors are shipped with the distro. To handle this situation, guest install tests take an extra argument (-kvm) to identify which type of guests will be installed. By default, kernel-xen

kernel is installed hence the guests are Xen guests. If -kvm is given in the guestargs, then the installation program decides that kvm guests are intended to be tested, so boots into the base kernel and then installs the guests. There can only be one hypervisor at work at one moment, and hence the installation test expects them all to be either kvm or xen guest, but not a mix of both.

Dynamic partitioning/LVM

Telling Beaker to create partitions/lvm. On Beaker, each machine has its own kickstart for each OS family it supports. In it the partitioning area is marked so that it can be overwritten to allow having dynamic partitions/lvms in your tests.

The easiest way to specify dynamic partitions is to use the xml workflow and specify it in your xml file. Syntax of the partition tags is below:

```
<partition
  type = type                  <!- required ->
  name = name                  <!- required ->
  size = size in GB            <!- required ->
  fs   = filesystem to format  <!- optional, defaulted to ext3 ->
</>
```

<partitions> is the xml element that holds all partition elements.

- <partition> is the xml element for the partitioning. You can have multiple partition elements in a partitions element. It has type, name, size and fs text contents all of which except for fs is required. Detailed information for each are:
  - *type*: Type of partition you'd like to use. This can be either part of lvm .
  - *name*: If the type is part, then this will be the mount point of the partition. For example, if you would like the partition to be mounted to /mnt/temppartition then just put it in here. For the lvm type, this will be the name of the volume and all custom volumes will go under its own group, prefixed with TestVolumeGroup? . For example, if you name your lvm type as "mytestvolume", it's go into /TestVolumeGroup??/mytestvolume.
  - *size*: The size of the partition or volume in GBs .
  - *fs*:This will be the filesystem the partition will be formatted in. If omitted, the partition will be formatted with ext3. By default, anaconda mounts all partitions. If you need the partition to be unmounted at the time of the test, you can use the blockdevice utility which is a test that lives on /distribution/utils/blockdevice . This test unmounts the specified partitions/volumes and lets users manage custom partitions thru its own scripts.

Dynamic partitioning from your workflow

If you are using a different workflow and would like to add dynamic partitioning capability, you can do it by utilizing addPartition() call to the recipe object. An example can be :

```
rec = BeakerRecipe()
# create an ext3 partition of size 1 gig and mount it on /mnt/block1
rec.addPartition(name='/mnt/block1', type='part', size=1)
# create an lvm called mylvm with fs ext3 and 5 gig size
rec.addPartition(name='mylvm', type='lvm', size=5)
# change the default fs from ext3 to ext4
rec.addPartition(name='/mnt/block4ext4', type='part', fs='ext4dev', size=1)
# create an lvm but change the default fs from ext3 to ext4.
rec.addPartition(name='mylvm4ext4', type='lvm', fs='ext4dev', size=5)
```

Helper programs installed with Virtinstall

Virtinstall test also installs a few scripts that can later on be utilized in the tests. These are completely non-vital scripts, provided only for convenience to the testers.

guestcheck4up:

- Usage: guestcheck4up <guestname>
- Description: checks whether or not the guest is live or not.
- Returns: 0 if guest is not shutoff, 1 if it is.

guestcheck4down:

- Usage: guestcheck4down <guestname>
- Description: checks whether or not the guest is live or not.
- Returns: 0 if guest is shutoff, 1 if it is not.

startguest:

- Usage: startguest <guestname> [timeout]
- Description: Starts a guest and makes sure that it's console is reachable within optional $timeout seconds. If timeout value is omitted the default is 300 seconds.
- Returns: 0 if the guest is started and a connection can be made to its console within $timeout seconds, 1 if it can't.

stopguest:

- Usage: stopguest <guestname> [timeout]
- Description: stops a guests and waits for shutdown by waiting for the "System Halted." string within the optional $timeout seconds. If timeout is omitted , then the default is 300 seconds.
- Returns: 0 if the shutdown was successful, 1 if it wasn't.

getguesthostname:

- Usage: getguesthostname <guestname>
- Returns: A string that contains the hostname of the guest if successful, or an error string if it's an error.

wait4login:

- Usage: wait4login <guestname> [timeout]
- Description: It waits until it gets login: prompt in the guest's console within $timeout seconds. If timeout argument is not given, it'll wait indefinitely, unless there is an error!
- Returns: 0 on success , or 1 if it encounters an error.

fwait4shutdown:

- Usage: wait4shutdown <guestname> [timeout]
- Description: It waits until it gets shutdown message in the guest's console within $timeout seconds. If timeout argument is not given, it'll wait indefinitely, unless there is an error!
- Returns: 0 on success , or 1 if it encounters an error.

3.6.3 Running tests in a Container

New in version 0.18.3.

Beaker supports running the tests in a Docker container instead of the host system. After the host system is provisioned, the test harness container is created which then executes the specified tasks in the recipe. Restraint is the default test harness used in the container.

Note: This is an experimental feature and limitations most likely exist other than the ones mentioned later in this guide.

An example Beaker job using this feature is as follows:

```
<recipeSet priority="Normal">
  <recipe kernel_options="" kernel_options_post="" ks_meta="no_default_harness_repo contained_harness
    <autopick random="false"/>
    <watchdog panic="ignore"/>
    <packages/>
    <ks_appends/>
    <repos>
      <repo name="restraint" url="http://10.64.41.123/localrepo/"/>
    </repos>
    <distroRequires>
      <and>
        <distro_family op="=" value="Fedora20"/>
        <distro_variant op="=" value="Fedora"/>
        <distro_name op="=" value="Fedora-20"/>
        <distro_arch op="=" value="x86_64"/>
      </and>
    </distroRequires>
    <hostRequires>
      <system_type value="Machine"/>
    </hostRequires>
    <partitions/>
    <task name="/restraint/true" role="None">
      <fetch url="git://git.example.org/~asaha/restraint_tasks?master#true"/>
    </task>
  </recipe>
</recipeSet>
```

The key points in the above recipe worth noting are:

- The `no_default_harness_repo` ksmeta variable tells Beaker to not include the repository for Beaker's test harness.
- `contained_harness` ksmeta variable: This tells beaker that we want to run the test harness in a Docker container.
- The host distro as specified by `<distroRequires/>` is Fedora 20.
- The harness repo needs to be specified as an additional repo using the `<repo/>` element. This is different from a "traditional" Beaker recipe since Beaker can figure out the harness repo to be used from the distro being used for the job.
- Since this recipe does not specify which distro image to use for the test harness container, Beaker tries to retrieve the docker image corresponding to the host distro i.e. Fedora 20 in this case.

Specifying the harness docker image

To specify a different image, use the `harness_docker_base_image` ksmeta variable. For example, the following recipe will run the test harness in a CentOS 7 container while the host system is running Fedora 20:

```
<recipeSet priority="Normal">
<recipe kernel_options="" kernel_options_post="" ks_meta="no_default_harness_repo contained_harness h
    <autopick random="false"/>
    <watchdog panic="ignore"/>
    <packages/>
    <ks_appends/>
    <repos>
      <repo name="restraint" url="http://10.64.41.123/localrepo/"/>
    </repos>
    <distroRequires>
      <and>
        <distro_family op="=" value="Fedora20"/>
        <distro_variant op="=" value="Fedora"/>
        <distro_name op="=" value="Fedora-20"/>
        <distro_arch op="=" value="x86_64"/>
      </and>
    </distroRequires>
    <hostRequires>
      <system_type value="Machine"/>
    </hostRequires>
    <partitions/>
    <task name="/restraint/true" role="None">
      <fetch url="git://git.example.org/~asaha/restraint_tasks?master#true"/>
    </task>
  </recipe>
</recipeSet>
```

The image specified by `harness_docker_base_image` is expected to be in a form usable in a Dockerfile's FROM instruction. One thing to keep in mind is that the distro should use systemd as the process manager.

Test harness container entrypoint

Beaker relies on systemd to initialize the test harness in the container and hence the harness container will execute **/usr/sbin/init** (using Dockerfile's `CMD` instruction) on startup. Hence it is the harness's responsibility to "enable" itself during the installation. This can however be changed with the `contained_harness_entrypoint` ksmeta variable. For example, if your test harness is a standalone binary, you may not want to use systemd for it. Specifying `contained_harness_entrypoint=/usr/bin/myharnessd` will ensure that the harness runs **/usr/bin/myharnessd** instead of **/usr/sbin/init**.

Limitations

This is an experimental feature and currently the following known limitations exist:

- The host OS must use systemd process manager and capable of running Docker containers.
- The harness repository must be specified in the recipe
- Tests which reboot are not supported
- Tests which may want to spawn other containers are not supported
- Running the test harness in a non-systemd distro is not tested

- Multi-host testing is not supported

To learn more about the above mentioned ksmeta variables, see *Kickstart metadata (ks_meta)*.

3.6.4 Running tests on an Atomic host

New in version 0.18.3.

Beaker supports running tests on host operating systems based on the Project Atomic pattern. The tests are run in a Docker container instead of the host when such an OS is used. If you are not familiar with this Beaker feature, see *Running tests in a Container* to learn more. Familiarity with this feature is assumed for the rest of this guide.

An example recipe which uses an atomic host OS is as follows:

```
<recipe kernel_options="" kernel_options_post="" ks_meta="no_default_harness_repo harness_docker_base
  <autopick random="false"/>
  <watchdog panic="ignore"/>
  <packages/>
  <ks_appends/>
  <repos>
    <repo name="restraint" url="https://beaker-project.org/yum/harness/CentOS7/"/>
  </repos>
  <distroRequires>
    <and>
      <distro_family op="=" value="MyAtomicHost7"/>
      <distro_variant op="=" value=""/>
      <distro_name op="=" value="My Atomic Host-7"/>
      <distro_arch op="=" value="x86_64"/>
    </and>
  </distroRequires>
  <hostRequires>
    <system>
      <memory op="&gt;" value="1500"/>
    </system>
    <system_type value="Machine"/>
  </hostRequires>
  <partitions/>
  <task name="/test-tasks/uname" role="STANDALONE"/>
</recipe>
```

The above recipe specifies that the test harness should be run in a CentOS 7 container. Two additional ksmeta variables have to be specified:

- `ostree_repo_url`: This variable is used to specify the `rpm-ostree` repository
- `ostree_ref`: This variable is used to specify the `rpm-ostree` remote ref

Note: You may also note that the above recipe specifies `<memory op=">" value="1500"/>` in the `<hostRequires>`. This is to specify that we want the host system to have at least 1500 MB memory. You may or may not need this to successfully execute a recipe.

3.6.5 Provisioning a system

If you would like to use one of these System you will need to provision it. Provisioning a System means to have the system loaded with an Operating System and reserved for the user. There are a couple of ways of doing this, which are outlined below.

Provision by system

Go to the System details page (see *System details*) of a System that is free (see *System searching*). For systems marked as Manual or Broken, click on Take in the Current User field. After successfully taking the System, click the Provision tab of the System details page to find the form that allows the System to be provisioned directly (without going through the job scheduler).

For systems in Automated mode, additional options are available. By default, the Provision tab will support scheduling a job through the scheduler that just runs the standard */distribution/reservesys*. However, if the user has the system loaned directly to them, they may also take the system and provision it directly, as in the Manual and Broken case. This allows the system to be provisioned without a time limit, whereas provisioning by running a job is covered by the scheduler's watchdog timers and will return the system automatically when the duration expires.

If the "Take" option does not appear, it usually means you do not currently have the relevant access to reserve the system. The Provision tab will contain an appropriate message when this is the case.

Returning a System

After taking a system, you can return it by going to the above mentioned system details page, clicking on the "Return" link in the "Current User" field. However this is only allowed if the system was provisioned manually (i.e not via a recipe)

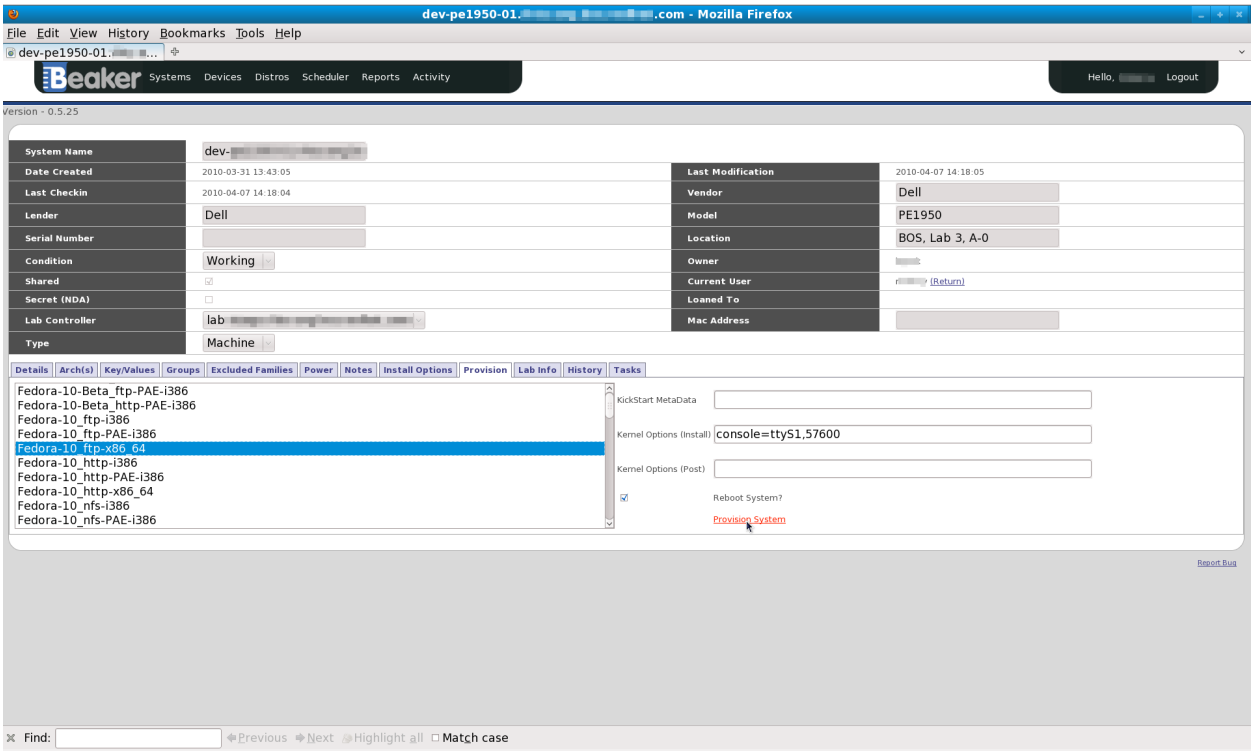


Figure 3.12: Provision by System

Provision by distro

Go to the *distro search page* and search for a Distro you would like to provision onto a System. Once you have found the Distro you require, click Provision System, which is located in the far right column of your search results. If the "Provision System" link is not there, it's because there is no suitable System available to use with that Distro.

The resulting page lists the Systems you can use. Systems with "Reserve Now" in the far right column mean that no on else is using them and you can reserve them immediately, otherwise you will see "Queue Reservation"; which means that someone is currently using the System but you can be appended to the queue of people wanting to use this System.

After choosing your System and clicking on the the aforementioned links, you will be presented with a form with the following fields:

- *System To Provision* This is our System we will provision.
- *Distro To Provision* The Distro we will be installing on the System.
- *Job Whiteboard* This is a reference that will be displayed in Jobs list. You can enter anything in here.
- *KickStartMetaData* Arguments passed to the KickStart script.
- *Kernel Options* (Install)
- *Kernel Options* (Post)

Pressing the "Queue Job" button will submit this provisioning as a Job and redirect us to the details of the newly created Job.

Reserve Workflow

The Reserve Workflow page is accessed from the top menu by going to "Scheduler > Reserve". The Reserve Workflow process allows the ability to select which System and Distro is to be provisioned based on the following:

- *Arch* Architecture of the System we want to provision.
- *Distro Family* The family of Distro we want installed.
- *Method* How we want the distro to be installed.
- *Tag* The Distro's tag.
- *Distro* Based on the above refinements we will be presented with a list of Distro's available to be installed.

Selecting values for the above items should be done in a top to bottom fashion, staring at "Arch" and ending with "Distro".

Once the Distro to be installed is selected you have the option of showing a list of System's that you are able to provision ("Show Systems" button), or you can have Beaker automatically pick a system for you ("Auto pick System"). If you choose "Show Systems" you will be presented with a list of Systems you are able to provision. Ones that are available now show the link "Reserve now" beside them. This indicates the System is available to be provisioned immediately. If the System is currently in use it will have the link "Queue Reservation" instead. This indicates that the System is currently in use, but can be provisioned for a later time.

Whether you choose to automatically pick a system or to pick one yourself, you will be presented with a page that asks you for the following options:

- Job Whiteboard
- KickStart MetaData
- Kernel Options (Install)
- Kernel Options (Post)

See *Provision by distro* where the above are explained. Once you are ready you can provision your System with your selected Distro by pressing "Queue Job".

3.6.6 Customizing partitions and volumes

When Beaker installs the distro at the start of each recipe, it will use the default disk layout ("automatic partitioning" with the `autopart` kickstart command).

The installer's automatic partitioning behaviour varies across releases, but in most cases the installer will assign all available disks to a single LVM volume group, with a swap volume, a 50GB root volume, and a home volume using all remaining space. Refer to the installer documentation for details about the automatic partitioning behaviour in each release.

Adding custom partitions

If the automatic partitioning behaviour is not suitable, your recipe can activate Beaker's custom partitioning logic by passing extra custom partitions in the `<partitions/>` element. For example, this will produce a 25GiB XFS-formatted filesystem mounted at `/var/tmp`:

```
<recipe>
  ...
  <partitions>
    <partition type="part" name="var/tmp" fs="xfs" size="25" />
  </partitions>
  ...
</recipe>
```

Each `<partition/>` element represents a custom disk partition and filesystem which will be created during the installation and then mounted when the recipe runs. Instead of using the `autopart` kickstart command, Beaker will emit suitable `part` commands to produce the desired partition layout.

Note: When Beaker's custom partitioning logic is activated, the root (`/`), `/boot`, and swap volumes are always created. Do not specify custom partitions for these.

The `<partition/>` element has the following attributes:

`type` The default value `part` produces a simple hard disk partition containing a filesystem directly. The value `lvm` instead produces a partition containing an LVM physical volume, with a *separate* LVM volume group containing a single LVM logical volume containing a filesystem.

`name` Mount point of the volume, without leading slash.

`fs` Filesystem type which will be used when formatting the partition, for example `ext4`, `xfs`, or `btrfs`. This follows the kernel naming scheme for filesystems, and the possible values depend on the distro. If this attribute is omitted, the installer will use the distro default filesystem type.

`size` Size of the partition in GiB.

There are also a number of kickstart metadata variables which influence the behaviour of the custom partitioning logic: `ondisk`, `fstype`, `rootfstype`, and `swapsize`. Refer to *Kickstart metadata (ks_meta)*. Note that if your recipe defines any of these variables, the custom partitioning logic will be applied *even if* your recipe does not contain any `<partition/>` elements.

Suppressing `autopart` and specifying partitioning commands directly

You can define the `no_autopart` kickstart metadata variable to suppress the `autopart` command which Beaker injects into the kickstart by default. If you *also* avoid all of the above custom partitioning mechanisms described above, this will result in a kickstart containing *no* partitioning commands.

Normally the installer considers this to be an error, because there are no instructions about how to lay out the disks.

However you can combine this with a `<ks_append/>` element (see *Appended kickstart content*) to append your own raw partitioning commands directly. For example, this will produce a 20GiB root volume and the remaining disk space will be allocated to a separate volume mounted at `/var/lib/mysql`, both using the default filesystem type for the distro:

```
<recipe ks_meta="no_autopart">
  ...
  <ks_appends>
    <ks_append>
part /boot --recommended
part / --size=20480
part /var/lib/mysql --grow
    </ks_append>
  </ks_appends>
  ...
</recipe>
```

3.6.7 Customizing the installation

Beaker provides a number of ways to customize the distro installation which happens at the start of each recipe.

Install options

In Beaker, install options are a set of three related argument strings in the form `foo=bar baz=qux`. Kernel options are passed on the kernel command line when the installer is booted. Post-install kernel options (labelled *Kernel Options Post* in the web UI) are set in the boot loader configuration, to be passed on the kernel command line *after* installation. Kickstart metadata variables are passed to the kickstart template engine, and can be used to control the content of the kickstart in various ways.

All three options can be set:

- by administrators at the OS version and distro tree levels (see *admin-os-versions*)
- by system owners on a per-system basis (see *System details tabs*)
- by job submitters in each individual recipe (see *Job workflow details*)

Beaker combines all the install options in the order listed above to determine the effective install options for each recipe. To unset an option of a previous setting, place *!* before the option. For example, the following will remove the beaker default kernel option setting of *ksdevice=bootif* from a user's job:

```
<recipe ... kernel_options="!ksdevice" ...>
```

Kernel options

Most kernel options are passed as-is on the kernel command line when the installer is booted. Refer to the distro documentation for details about kernel options supported by the installer.

The following kernel options are treated specially by Beaker:

ks=<url> This option is passed as-is on the kernel command line. It specifies the kickstart file for Anaconda.

> Beaker performs no extra processing on this kernel option, however if it is present Beaker skips all of the normal mechanisms for kickstart generation using templates and variables (described below). The kickstart used for provisioning will be the one given in this option.

initrd=<tftp path> Extra initrd/initramfs image to load, in addition to the initrd image for the distro installer. Use this to apply updates to the installer image, or to supply additional drivers for installation.

If the boot loader supports multiple initrd images, Beaker extracts the `initrd=` option from the kernel command line and appends it to the boot loader configuration.

devicetree=<tftp path> Alternate device tree binary to load. Use this to supply a different device tree binary than the one built into the kernel.

If the boot loader supports passing a device tree to the kernel (currently only GRUB for AArch64), Beaker extracts the `devicetree=` option from the kernel command line and appends it to the boot loader configuration.

netbootloader=<tftp path to bootloader> Netboot loader image to use. Beaker creates a symlink so the TFTP path `bootloader/`*`fqdn`*`/image` serves the specified image.

Set this option if you want to boot an alternative image. For example, if the administrator has made an older version of PXELINUX available in the TFTP root as `pxelinux-311.0`, you can boot it using `netbootloader=pxelinux-311.0`.

By default Beaker uses the most suitable boot loader for the chosen distro and architecture:

- i386/x86_64: `pxelinux.0`
- ia64: `elilo-ia64.efi`
- ppc: `yaboot`
- aarch64: `aarch64/bootaa64.efi`

For ppc64 and ppc64le, for Fedora, RHEL 7.1 and later:

- `boot/grub2/powerpc-ieee1275/core.elf`

and for RHEL 7.0 and earlier:

- `yaboot`

Note that this option will have no effect if the system has a hard-coded boot loader filename in the DHCP configuration. For configurable netboot loader support the DHCP configuration must specify the filename as `bootloader/`*`fqdn`*`/image`. See *DHCP and DNS*.

Kickstart metadata (ks_meta)

The following variables are supported and can be selected by placing them in the `ks_meta` attribute in the `recipe` element. In many cases, these variables correspond to the similarly-named kickstart option.

For example in the job XML:

```
...
<recipe kernel_options="" kernel_options_post="" ks_meta="harness=restraint hwclock_is_utc" role="No
...
```

auth=<authentication configuration options> Authentication configuration to use. For example, `auth='--enableshadow --enablemd5'`. See *`authconfig(8)`* to learn more.

autopart_type=<fstype> Partioning scheme for automatic partitioning (must be one of `lvm`, `btrfs`, `plain` and `thinp`). On supported distros, it is passed as `--type <fstype>` to the `autopart` kickstart command. On distros where `autopart` does not support the `--type` option, this is ignored.

beah_rpm=<pkgarg> Name of the Beah RPM to be installed. The value can be any package specification accepted by yum (for example it can include a version, such as `beah-0.6.48`). The default is `beah` which installs the latest version from the harness repos. This variable has no effect when using alternative harnesses.

beah_no_console_output If specified, Beah will not send any log messages to /dev/console. The log messages will still be available in the systemd journal (on systemd-based distros).

beah_no_ipv6 If specified, Beah will function in IPv4 only mode even if IPv6 connectivity is possible.

bootloader_type Specify an alternative bootloader. It is passed on to the `bootloader` kickstart command.

dhcp_networks=<device>[;<device>...] Configure additional network devices to start on boot with DHCP activation. The device should be given as a kernel device name (for example, `em1`) or MAC address.

Note that the network device used for installation is always set to start on boot with DHCP activation.

conflicts_groups This is a list of comps.xml group ids (without the @ symbol) which contain packages conflicting with the rest of the package set, e.g. Samba 3 vs. Samba 4. Empty list is a valid value as well so templates can iterate over the list without testing if the variable has been defined. Usually applicable for RHEL and CentOS.

contained_harness If specified, runs the test harness and hence the tasks in a Docker container. The test harness to be run defaults to "restraint". A different test harness can be specified using the `harness` variable. Also see `contained_harness_entrypoint` below.

The host distro and architecture must support Docker for this to be possible.

contained_harness_entrypoint=<entrypoint> Specify how the harness should be started. This defaults to "/usr/sbin/init" and expects "systemd" to be the process manager. Alternatively, another binary can be specified. The entry point must be in one of the forms understood by Docker's CMD instruction.

This is only required if the test harness is run in a Docker container. See `contained_harness` above.

contained_harness_ro_host_volumes=</volume1>[,</volume2>..] Specify the host volumes to be mounted as read-only inside the container. The default volumes mounted as read-only are `/var/log/messages`, `/etc/localtime` and `/etc/timezone`.

For example, `contained_harness_ro_host_volumes='/var/run,/etc'` will then mount `/var/run` and `/etc` as read-only volumes.

contained_harness_rw_host_volumes=</volume1>[,</volume2>..] Specify the host volumes to be mounted with write permissions inside the container. The default volumes with write permissions are `/mnt` and `/root`.

For example, `harness_rw_host_volumes='/myvolume'` will then only mount the `/myvolume` with write permissions.

disable_repo_<id> Disable repo of the given id. By default, the kickstart will include repo. Repo is also created in yum.repos.d/ and enabled. Set this variable if you want to omit `repo` command in kickstart and disable it in yum.repos.d/. You can find repo id for a particular distro tree under the *Repos* tab on the distro tree page. For example disable_repo_CRB.

disable_<type>_repos Disable repos of the given types. Valid types include `variant`, `addon`, `optional`, and `debug`. By default, the kickstart will include all repos of the given type. Repos are also created in yum.repos.d/ and enabled. Set this variable if you want to omit `repo` command in kickstart and disable them in yum.repos.d/. You can find repo id for a particular distro tree under the *Repos* tab on the distro tree page. For example disable_debug_repos.

disable_onerror Don't add the `%onerror` section to the kickstart. By default the section is added for RHEL 7 and newer. It handles the installation failure reporting it to Beaker and aborting the recipe. This option disables this functionality.

ethdevices=<module>[,<module>...] Comma-separated list of network modules to be loaded during installation.

firewall=<port>:<protocol>[,<port>:<protocol>...] Firewall ports to allow, for example `firewall=imap:tcp,1234:ucp,47`. If this variable is not set, the firewall is disabled.

firstboot=<option> Option to be passed directly to the `firstboot` kickstart command. The complete option string must be given, including `--` and `=`.

fstype Filesystem type for all filesystems. Default is to allow the installer to choose.

grubport=<hexaddr> Hex address of the I/O port which GRUB should use for serial output. If this variable is set, the value will be passed to the `--port` option of the `serial` command in the GRUB configuration. Refer to serial in the GRUB manual.

harness=<alternative harness> Specify the test harness to use instead of the default test harness, "beah". With the `contained_harness` variable specified, this defaults to "restraint".

To learn more, see the *alternative-harnesses*.

harness_docker_base_image=<image> If specified, uses this docker image to build the Docker container that the test runs in. The <image> is expected to be in a form usable in a Dockerfile's FROM instruction. If `contained_harness_entrypoint` is not specified, the distro should use "systemd" as the process manager.

If not specified, Beaker will attempt to build the container by fetching the same image as that of the host distro from the Docker public registry. Thus, if Fedora 20 is used on the host machine, the image used will be: "registry.hub.docker.com/fedora:20".

hwclock_is_utc If defined, the hardware clock is assumed to be set in UTC rather than local time. It's defined by default for guest recipes and dynamic VMs.

ignoredisk=<options> Options to be passed directly to the `ignoredisk` kickstart command. The complete option string must be given, including `--` and `=`.

Use this to skip certain disks during the installation, for example `ignoredisk=--drives=sdb,sdc`, or to use only certain disks for the installation, for example `ignoredisk=--only-use=sda,sdb`.

install_task_requires If defined, Beaker will include the packages required by every task in the recipe. The packages will be populated in the kickstart `%packages` section, causing Anaconda to install them.

For Fedora 29 and RHEL8 and later versions, this variable is *not* defined by default. The default harness, Restraint, will install task requirements before executing each task.

This variable *is* defined by default for older distros, where the default harness is Beah. Beah does not install task requirements, instead it relies on the requirements being populated in %packages and installed by Anaconda.

keyboard=<layout> Keyboard layout to use. Default is `us`.

ks_keyword Change the normal kickstart kernel command line keyword from 'ks' to user specified value. This can be useful for instance if wanting to install an operating system like Debian. Could do `ks_keyword=preseed` and then the kickstart file will be generated but on kernel command it would be present as `preseed=http://example.com/kickstart/18` as an example. A custom preseed template will need to be created and put in `/etc/beaker/kickstarts/`. For example if doing a Debian install the file `/etc/beaker/kickstarts/Debian` should be created.

lang=<localeid> Locale to use. Default is `en_US.UTF-8`.

manual Omits most kickstart commands, causing Anaconda to prompt for details. The effect is similar to booting from install media with no kickstart. Typically it is also necessary to set `mode=vnc`. For systems with console log monitoring enabled, it will also be necessary to switch off *installation failure monitoring*.

method=<method> Installation method to use. Default is `nfs`, supported alternatives include `http` and `nfs+iso`. The specific installation methods supported for a particular distro tree in a particular lab will depend on how the distro was imported into Beaker. The available methods can be determined through the web UI by looking at the URL schemes listed for the distro tree.

mode=<mode> Installation mode to use. Valid values are `text` (curses-like interface), `cmdline` (plain text with no interaction), `graphical` (local X server), and `vnc` (graphical interface over VNC). The default mode is either `text` or `cmdline`, depending on arch and distro.

no_autopart Omits the `autopart` command. By default when no specific partitions are requested for a recipe, the kickstart will include `autopart` which causes the installer to automatically select a suitable partition layout. Set this variable if you want to supply explicit partitioning commands in some other way, for example in a `<ks_append/>` section.

no_ks_template The whole kickstart has to be defined by user in `<kickstart />` tag. Default Beaker's templating is not used; however, user does have access to snippets and restricted context.

no_networks Omits the `network` command. By default when no specific network is requested for a recipe, the kickstart will include `network --bootproto=dhcp` which causes the installer to obtain it's networking configuration from DHCP server. Set this variable if you want to supply explicit network setting in kernel options.

no_repo_<id> Omits repo of the given id. You can find repo id for a particular distro tree under the *Repos* tab on the distro tree page. For example no_repo_CRB-debuginfo.

no_<type>_repos Omits repos of the given type. Valid types include `variant`, `addon`, `optional`, and `debug`. You can find which repo types are available for a particular distro tree under the *Repos* tab on the distro tree page.

no_clock_sync Omits additional packages and scripts which ensure the system clock is synchronized after installation.

no_default_harness_repo If you have your own repository providing a test harness, this variable can be used to prevent Beaker from configuring the default Beaker harness repository in the kickstart.

no_disable_readahead By default Beaker disables readahead collection, because it is not generally useful in Beaker recipes and the harness interferes with normal data collection. If this variable is set, Beaker omits the snippet which disables readahead collection.

no_updates_repos Omits the fedora-updates repo for Fedora.

liveimg Specify a relative path to an image file or rpm that contains a squashfs image. For more details, see the *kickstart documentation <http://pykickstart.readthedocs.io/en/latest/kickstart-docs.html#liveimg>*

ostree_repo_url Specify the repo location for rpm-ostree. See `has_rpmostree` below.

ostree_ref Specify the remote ref for rpm-ostree. See `has_rpmostree` below.

packages=<package>:<package> Colon-separated list of package names to be installed during provisioning. If this variable is set, it replaces any packages defined by default in the kickstart templates. It also replaces any packages requested by the recipe, including task requirements.

In a recipe, considering using the `<package/>` element instead. This augments the package list instead of replacing it completely.

password=<encrypted> Root password to use. Must be encrypted in the conventional *`crypt(3)`* format.

pkgoptions=<options> Options to pass to the `%packages` section in the kickstart file. If this variable is set, it overrides the default option `--ignoremissing`. See the *kickstart documentation <http://pykickstart.readthedocs.io/en/latest/kickstart-docs.html#chapter-7-package-selection>* for the available options to `%packages`.

remote_post=<url> Specify a URL to a script to be executed post-install. The script must specify a interpreter using the `#!` line if not a bash script. This is especially useful for systems set to Manual mode. If you are scheduling a job, a simpler alternative is to embed a `%post` scriptlet directly in your job XML using the `<ks_append/>` element.

rootfstype Filesystem type for the root filesystem. Default is to allow the installer to choose.

scsidevices=<module>[,<module>...] Comma-separated list of SCSI modules to be loaded during installation.

selinux=<state> SELinux state to set. Valid values are `--disabled`, `--permissive`, and `--enforcing`. Default is `--enforcing`.

skipx Do not configure X on the installed system. This is needed for headless systems which lack graphics support.

skip_taskrepo Configure `skip_if_unavailable` yum repo attribute for task repository. Default value is 0.

static_networks=<device>,<ipv4_address>[;...] Configure one or more network devices to start on boot with static IPv4 addresses. The device should be given as a kernel device name (for example, `em1`) or MAC address. The IPv4 address should be given with its netmask in CIDR notation (for example, `192.168.99.1/24`).

Note that the network device used for installation is always set to start on boot with DHCP activation.

swapsize Size of the swap partition in MB.

timezone=<tzname> Time zone to use. Default is `America/New_York` unless overridden by the administrator.

yum_install_extra_opts Extra command-line options which will be passed to all invocations of `yum install` which Beaker produces in the generated kickstart.

On RHEL3 and RHEL4 this variable defaults to "-d 1" which inhibits Yum's progress bar made up of hashes which can take a long time to print. On newer releases, where Yum's progress bar produces less output, this variable is undefined.

Distro features

The following kickstart metadata variables are used to test for installer or distro features. Beaker populates these variables automatically by inspecting the distro name and version. They can be overridden if necessary for custom distros.

boot_partition_size Recommended size of the `/boot` partition according to the product documentation. This is only populated for RHEL 6 and older releases, where the installer does not support the `--recommended` option for the `part` command. In newer releases the installer correctly determines the recommended size.

docker_package The package name for Docker container engine is `docker-io` on Fedora 20/21 and `docker` starting with Fedora rawhide (bugzilla report), CentOS 7 and RHEL 7.

end Set to `%end` on distros which support it, or to the empty string on older distros.

has_autopart_type Indicates that the `autopart` kickstart command accepts a `--type` option.

has_chrony Indicates that chrony is available in the distro.

has_dhcp_mtu_support Indicates that the DHCP client obeys option 26 for controlling the network interface MTU. Support for this DHCP option is required for a recipe to run successfully in OpenStack. All modern distributions from RHEL 5 onwards support this.

has_gpt_bios_support Indicates that the installer is capable of formatting disks using GPT on x86 systems with BIOS firmware.

This support is needed for disks larger than 2TB and it requires an extra "BIOS boot" partition to be defined.

has_key Indicates that the distro requires the `key` command. This command exists only on RHEL 5 and CentOS 5.

has_leavebootorder Indicates that the `bootloader` command accepts a `--leavebootorder` option.

has_repo_cost Indicates that the `repo` command accepts a `--cost` option.

has_reqpart Indicates that the installer supports the `reqpart` command for adding platform-specific partition requirements.

has_rpmostree If specified, Beaker assumes that the specified distribution is rpm-ostree based (an Atomic host, for example). The test harness is run inside a Docker container and the tests are run inside it instead of the host system. The OSTree location and ref must be specified using `ostree_repo_url` and `ostree_ref` respectively.

Also, see `harness_docker_base_image` and `contained_harness_entrypoint` above.

has_systemd Indicates that the distro uses systemd rather than SysV init.

has_unsupported_hardware Indicates that the `unsupported_hardware` kickstart command is accepted.

yum Unset, except on older distros which require the yum package to be fetched and installed.

Appended kickstart content

In your job XML you can specify extra content to be appended to the generated kickstart, using the `<ks_appends/>` element. For example:

```
<recipe>
    ...
    <ks_appends>
        <ks_append><![CDATA[
%post
echo "This is my extra %post script"
%end
        ]]></ks_append>
    </ks_appends>
</recipe>
```

Custom kickstart templates

You can also specify a complete kickstart template in your job XML, using the `<kickstart/>` element. Note that if a custom template is supplied, the other customization mechanisms described above (`ksmeta=` and `<ks_appends/>`) will have no effect, unless the custom template also obeys those customizations.

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.

All kickstart metadata variables are available to the kickstart template. That includes variables set on the recipe, the system, the distro, the OS major, and system-wide in the Beaker configuration. It also includes distro feature variables (see *Distro features* above) which are particularly useful in kickstart templates for handling differences between distros and versions.

A number of additional Beaker-specific Jinja filters, tests, and variables are defined in the template environment. They are described below.

Jinja filters

dictsplit (*delim=', '*, *pairsep=':'*)
Returns a dict based on a sequence of key-value pairs encoded in a string, like this:

```
type:mdraid,part:swap,size:256
```

parsed_url()
Parses a URL using `urlparse.urlparse()`.

shell_quoted()
Quotes a string using `pipes.quote()`, suitable for interpolation as an argument into a shell command.

split(*delim=None*)
Splits on whitespace, or the given delimiter. See `string.split()`.

urljoin(*relativeurl*)
Resolves a relative URL against a base URL. For example:

```
{{ 'http://example.com/distros/'|urljoin('RHEL-6.2/') }}
```

will evaluate to `http://example.com/distros/RHEL-6.2/` in the kickstart.

Jinja tests

arch(*arch*, ...)
Tests whether a distro tree's arch matches any of the given arches. For example:

```
{% if distro_tree is arch('i386', 'x86_64') %}
```

osmajor(*osmajor*, ...)
Tests whether a distro matches any of the given OS major names. For example:

```
{% if distro is osmajor('CentOS6', 'RedHatEnterpriseLinux6') %}
```

In most cases it is preferable to use a distro feature variable rather than hard-coding all possible OS major names.

osversion(*osversion*, ...)
Tests whether a distro matches any of the given OS versions. For example:

```
{% if distro is osversion('CentOS6.0', 'RedHatEnterpriseLinux6.0') %}
```

Template variables

absolute_url(*path*, ***kwargs*)
A function which returns the absolute URL to the given path within the Beaker application. *kwargs* are converted to query parameters.

chr(*i*)
The built-in `chr` function, which returns a byte with the given integer value.

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

name Name of the distro, for example "Fedora-Server-21_Alpha".

osversion Object representing the distro's version.

osversion.osminor OS minor version (the portion after the first period).

osversion.osmajor Object representing the OS major version.

osversion.osmajor.name Name portion of the OS major version, for example "Fedora".

osversion.osmajor.number Numerical portion of the OS major version, for example "21". Note that this is a string, not an integer, because it might be "rawhide".

osversion.osmajor.osmajor Complete OS major version string, for example "Fedora21".

distro_tree
The distro tree which is being provisioned. This object has the following attributes:

arch Object representing the CPU architecture which this tree was built for.

arch.arch Name of the CPU architecture which this tree was built for, for example "x86_64".

url_in_lab(lab_controller) A method which returns a URL for this distro tree in the given lab.

variant Name of the distro variant, for example "Server". This may also be empty.

harnessrepo=<repo_name>,<repo_url>
Repository used to download harness RPMs from. For example:

```
{% if harnessrepo %}
    {% set repo_name, repo_url = harnessrepo.split(',', 1) %}
    repo --name={{ repo_name }} --baseurl={{ repo_url }}
{% endif %}
```

job_whiteboard
The value of the job whiteboard.

kernel_options_post
Post-install kernel options from the install options.

ks_appends
List of string containing extra kickstart content supplied by the job submitter in the `<ks_appends/>` element.

lab_controller
The lab controller where the system is being provisioned.

fqdn The fully-qualified domain name of the lab controller.

netaddr
The Python netaddr module for manipulating network addresses.

ord(*c*)
The built-in `ord` function, which returns the integer ordinal of the given character.

re
The Python `re` module, for evaluating regular expressions.

recipe
The recipe to be run on the provisioned system. This object has the following attributes:

id The recipe id. This is used when configuring the harness, and in many harness-related APIs.

whiteboard The recipe whiteboard, a user-supplied description of this recipe.

role This recipe's role in a multi-host recipe set, for example `SERVERS`.

recipe_whiteboard
The value of the recipe whiteboard.

snippet(*name*)
A function which evaluates the named snippet and returns the result. If no template is found for the snippet, returns a comment to that effect.

This is also available as a Jinja statement, for example:

```
{% snippet 'network' %}
```

taskrepo=<repo_name>,<repo_url>
Repository used to download tasks RPMs from. This is independent for each job. For example:

```
{% if taskrepo %}
    {% set repo_name, repo_url = taskrepo.split(',', 1) %}
    repo --name={{ repo_name }} --baseurl={{ repo_url }}
{% endif %}
```

var (*name*)
A function which returns the value of the template variable with the given name.

3.7 Writing tasks

3.7.1 An example task: checking for ext4 support

To get a basic idea of how we can use **beaker-wizard**, we will create a new task which will check whether the platform supports the ext4 filesystem. We use restraint as a test harness, since it allows us to define Beaker jobs with tasks retrieved from a git repository.

If you have a basic understanding of test frameworks and don't want to install the **beaker-wizard**, you can jump right to the *Implementing the test* part of the tutorial.

Prerequisites

The *beaker-wizard* utility provides a guided step by step method to create a task without the need to manually create all of the necessary files.

In case you do not have **beaker-wizard** available, install it by following the installation guide for the *bkr client*.

Generating the skeleton using beaker-wizard

Create a directory with the name `ext4-task`. The directory will hold metadata and task executable. From your terminal, type:

```
$ mkdir ext4-task
$ cd ext4-task
$ beaker-wizard --current-directory
```

and follow **beaker-wizard**. If in doubt choose the default values offered by the wizard. As a **test-name** use `ext4-test`. Finally, press the enter key to create the task:

```
File PURPOSE written
File runtest.sh written
File Makefile written
```

Implementing the test

In the `ext4-test` directory, you will notice that the three files: `PURPOSE`, `runtest.sh`, and a `Makefile` have been created. The test itself is kept in `runtest.sh` executed by the test harness using the `Makefile`, while `PURPOSE` provides information for humans.

`PURPOSE` and `Makefile` are actually not needed when using the **restraint** test harness. Alternatively, you can create a metadata file instead.

The test is written using BeakerLib commands. The functionality is divided into three stages: setup, start and cleanup, as indicated by the `rlPhaseStartSetup`, `rlPhaseStartTest` and `rlPhaseStartCleanup` functions respectively.

Remove the setup (`rlPhaseStartSetup`) and teardown (`rlPhaseStartCleanup`) phases, since they're not needed for this simple test. Replace the actual test code (starting with `rlPhaseStartTest`) so that the file looks like this:

```
#!/bin/bash
# Include Beaker environment
. /usr/bin/rhts-environment.sh || exit 1
. /usr/share/beakerlib/beakerlib.sh || exit 1

rlJournalStart
    rlPhaseStartTest
        rlRun "cat /proc/filesystems | grep 'ext4'" 0 "Check if ext4 is supported"
    rlPhaseEnd
rlJournalPrintText
rlJournalEnd
```

The BeakerLib manual provides an extensive reference of what utility functions are available to test authors.

Testing the task from a git repository

In order to test your task, create a public git repository (e.g. Fedora's Pagure, github, etc) and publish your code.

Next we will submit a Beaker job to run our newly published task. We will need to write a job definition using Beaker's *job XML* syntax. As mentioned above, we want to select the **restraint** harness which is capable of fetching our task directly from its git repository. The job will contain one recipe, using a version of Fedora, with one task in the recipe:

```
 <job>
   <whiteboard>
     ext4 test
   </whiteboard>
   <recipeSet>
     <recipe ks_meta="harness='restraint beakerlib'">
       <distroRequires>
           <distro_family op="=" value="Fedora23"/>
           <distro_arch op="=" value="x86_64"/>
       </distroRequires>
       <hostRequires />

       <task>
         <fetch url="<URL OF YOUR TASK REPOSITORY>" />
       </task>

     </recipe>
   </recipeSet>
 </job>
```

Running the task

You can then submit the job (see *Job submission*). After the job has completed, you can access the logs as described in *Job results*. You will see that on success, the `taskout.log` file will provide verbose information about the progress of the test and it's result.

The overall workflow of creating a task for a test, submitting a job to run the test and accessing the test results is illustrated in *chronological-overview*.

Next steps

The Beaker meta tasks git repository provides tasks which are in use daily by the Beaker team. They can give you further information on how you can write tasks. The task described in this tutorial can be inspected in the same repository under examples. If you run into problems when scheduling your task in Beaker, the *Troubleshooting* section might be of interest to you. Further information on the test harness used in this tutorial can be found in the Restraint documentation.

3.7.2 Execution environment for tasks

This sections describes the commands and environment variables which are available in the execution environment for tasks. At a bare minimum, a task should use **rhts-report-result** to report a Pass or Fail result when it finishes.

The BeakerLib shell library provides many convenience functions on top of these commands, including functions to structure complicated tasks into separate phases.

Commands

The following commands are available in `PATH` when a task is executed. The task can use these to interact with Beaker.

These commands originated in the Red Hat Test System (RHTS), the precursor to Beaker, which is why their names begin with the prefix "rhts-".

rhts-abort

```
rhts-abort -t <type>
```

Aborts the current recipe, recipe set, or job. A setup task might call this if it detects a failure that will prevent the rest of the recipe from running.

-t `<type>`
: Valid values are `recipe`, `recipeset`, or `job`. Abort the recipe, the containing recipe set, or the entire job. This option is required.

rhts-backup

```
rhts-backup <file> [<file> ...]
```

Copies the given files or directories to a temporary storage area for the duration of the task. The files will be copied back to their original location at the end of the task, using the corresponding **rhts-restore** program.

For example, a task which needs to modify `/etc/hosts` could back it up before modifying it:

```
rhts-backup /etc/hosts
echo "127.0.0.1 testing-bad-values" >/etc/hosts
# test some more things here...
```

When the task finishes, **rhts-restore** copies the original version of `/etc/hosts` back, leaving the system in a clean state for the next task.

rhts-power

```
rhts-power <fqdn> <action>
```

Sends a power command to another system in the recipe set.

<fqdn>
FQDN of the system to be power-controlled.

<action>
Power command to send. These correspond to the power commands available in Beaker. Valid values are `on`, `off`, `reboot`, and `interrupt`.

rhts-reboot

Saves the harness state, and then reboots the system. Tasks should use this command instead of the conventional **reboot** command.

The task script should test the `REBOOTCOUNT` environment variable before rebooting, to avoid infinite loops. For example:

```
if [ "$REBOOTCOUNT" -eq 0 ] ; then
    # do some setup work here
    rhts-reboot
fi
# do the real work here
```

rhts-report-result

```
rhts-report-result <testname> <result> <outputfile> [<metric>]
```

Reports a Pass or Fail result to Beaker. A task can report multiple results (for example, for different phases or cases) but it should report at least one result.

<testname>
Test name for the result being reported. By convention the result of the entire task is reported as `$TEST`, and any individual phases or cases within the task are reported as a path underneath `$TEST` (for example, `$TEST/setup`, `$TEST/test-case-5`). However Beaker accepts any string for the test name.

<result>
Result to report. Valid values are `PASS`, `WARN`, `FAIL`, or `SKIP`.

<outputfile>
File name of the captured output or logging relevant to this result. The file will be uploaded to Beaker and made available alongside the reported result in Beaker's interface.

<metric>
Optional integer "score" or "metric" to be shown alongside this result. Beaker assigns no meaning to this score, the task can use it for any purpose. Some example uses include: the score from a performance test run, the number of test cases executed, or the exit status of a failing command.

rhts-restore

Restores files which were previously backed up using **rhts-backup**. This command is run automatically when a task finishes, there is normally no need to invoke it explicitly in the task.

rhts-run-simple-test

```
rhts-run-simple-test [-u <user>] <testname> <command>
```

Runs the given command, with output redirected to `$OUTPUTFILE`, and reports a Pass or Fail result according to the exit status of the command. If you have another program or script which does the real work for the task (for example, a test suite runner), you can use **rhts-run-simple-test** as a wrapper around it.

-u <user>
: Run the command as *user*.

<testname>
: Test name to use when reporting the result to Beaker. See **rhts-report-result**.

<command>
: Command to run. Note that shell word splitting is applied to *command*, so any additional arguments should be passed as a single word.

rhts-submit-log

```
rhts-submit-log -l <file>
```

Uploads a log file to Beaker. The file will be available in the Beaker results for the current task.

rhts-sync-block

```
rhts-sync-block -s <state> [--timeout <timeout>] [--any] <fqdn> [<fqdn> ...]
```

Blocks until the given systems in this recipe set have reached a certain state. Use this command, along with **rhts-sync-set**, to synchronize between systems in a multihost recipe set.

Refer to *Multihost tasks* for a more detailed guide.

-s <state>
: Wait for the given state. If this option is repeated, the command will return when any of the states has been reached. This option is required.

--timeout <timeout>
: Return a non-zero exit status after *timeout* seconds if the state has not been reached. By default no timeout is enforced and the command will block until either the given state is reached on all specified systems or the recipe is aborted by the local or external watchdog.

--any
: Return when any of the systems has reached the given state. By default, this command blocks until *all* systems have reached the state.

<fqdn> [<fqdn> ...]
: FQDN of the systems to wait for. At least one FQDN must be given. Use the role environment variables to determine which FQDNs to pass.

rhts-sync-set

```
rhts-sync-set -s <state>
```

Sets the given state for this system. Other systems in the recipe set can use **rhts-sync-block** to wait for a state to be set on other systems.

States are scoped to the current task. That is, states set by the current task will have no effect in subsequent tasks. You can use the matching commands **rhts-recipe-sync-set** and **rhts-recipe-sync-block** to set and wait for states that are global for the recipe instead.

Internal commands

The following commands are used internally by the harness and should not be invoked by tasks directly:

- **rhts-db-submit-result**
- **rhts-extend**
- **rhts-system-info**
- **rhts-test-checkin**
- **rhts-test-update**

Environment variables

The harness sets a number of environment variables in the execution environment for tasks. The task can use these to adjust its behaviour as needed.

Task parameters (given in the Beaker job XML using `<params/>`) are also passed to the task as environment variables.

Note that these environment variables *will not* be set when you log in to the system as a user over SSH or on the console.

TEST
: The name of the current task. `TASKNAME` is an alias for this variable.

TESTPATH
: Path to the directory containing this task.

TESTRPMNAME
: NVRA (Name-Version-Release.Arch) of the current task RPM. Deprecated: do not rely on tasks being packaged as RPMs.

KILLTIME
: Expected run time of this task in seconds. This is declared in the TestTime field in the task metadata (see *TestTime*), and is the length of time by which the harness extends the watchdog at the start of the task.

FAMILY
DISTRO
VARIANT
ARCH
: Details of the Beaker distro tree which was installed for the current recipe.

SUBMITTER
: Email address of the Beaker user who submitted the current job.

JOBID
RECIPESETID
RECIPEID

TASKID
Beaker database IDs for the current job, recipe set, recipe, and recipe-task respectively. `TESTID` and `RECIPETESTID` are deprecated aliases for `TASKID`.

TESTORDER
Integer counter for tasks in the recipe. The value increases for every subsequent task, and every peer task in the recipe set will have the same value, but note that it *does not* increase by 1 for each task.

REBOOTCOUNT
Number of times this task has rebooted. The counter starts at zero when the task is first run, and increments for every reboot. If a task triggers a reboot, it can test this variable to decide which phase of the test to enter so that it doesn't loop infinitely.

RECIPETYPE
The type of the recipe. Possible values are `guest` for a guest recipe, or `machine` for a host recipe. See *Virtualization workflow*.

GUESTS
Deprecated. The recommended means of looking up details of guest recipes is to fetch the recipe XML from the lab controller and parse it (see `GET /recipes/(recipe_id)/`).

RECIPE_MEMBERS
Space-separated list of FQDNs of all systems in the current recipe set.

RECIPE_ROLE
The role for the current recipe. See *Multihost tasks*.

ROLE
The role for the current task. See *Multihost tasks*.

HYPERVISOR_HOSTNAME
The hostname of a guest recipe's host. This is retrieved at recipe run time, and is not dynamically updated (i.e if you migrate your guest this variable will not be updated).

Additionally, one environment variable will be set for each recipe role defined in the recipe set. The name of the environment variable is the role name, and its value is a space-separated list of FQDNs of the systems performing that role. Similarly, each task role is set as an environment variable, but note however that task roles are only shared amongst recipes of the same type. That is, task roles for guest recipes are not visible to host recipes, and vice versa. See *Multihost tasks* for further details.

The following environment variables are set system-wide by Beaker at the start of the recipe.

LAB_CONTROLLER
FQDN of the lab controller which the current system is attached to.

BEAKER
FQDN of the Beaker server.

BEAKER_JOB_WHITEBOARD
Whiteboard of the current job.

BEAKER_RECIPE_WHITEBOARD
Recipe whiteboard for the current recipe.

3.7.3 Task metadata

This section describes the metadata which must be defined in each Beaker task. The beaker-wizard utility will help you to populate this metadata correctly when creating a new task (see *An example task*). A sample Makefile is also included in the `rhts-devel` package as `/usr/share/doc/rhts-devel-*/Makefile.template`.

These details apply to tasks written using the default harness (`beah`). Programs written using an *alternative harness* will likely be configured differently — consult the documentation for the specific harness in use.

Makefile variables

The following environment variables must be exported in the task's Makefile. These variables are used by `rhts-make.include` and its anxillary scripts when building the task RPM.

`TEST` The name of the task. The name is a hierarchical path beginning with a slash (`/`), similar to a filesystem path. For example, `/distribution/mypackage/test-suite`. The task name is available as `TEST` in the task's environment.

The task name is prefixed with `/mnt/tests` to form the directory where it will be installed on the system under test. This directory is available as `TESTPATH`.

The task should report results relative to its name (see *rhts-report-result*).

`TESTVERSION` The version of the task. This becomes the version of the task RPM when it is built. As a consequence, it may contain only numbers, digits, and period (`.`).

`FILES` A whitespace-separated list of all the files to be included in the task RPM. This must include at least `testinfo.desc` (typically given as `$(METADATA)`), `runtest.sh`, and `Makefile`. If a task uses any additional scripts or data, those files must be listed here.

`BUILT_FILES` Files which are generated or compiled by other rules in the Makefile should be listed in this variable, rather than in `FILES`, so that they are built when the task RPM is built.

`TEST_DIR` This can be used with the `Path` metadata field described in the next section. It is not defined by the user but defined by rhts tools to concatenate `/mnt/tests` with the name of the test or `$(TEST)`.

Fields in `testinfo.desc`

The `testinfo.desc` file declares metadata about the task. The metadata is extracted by Beaker and made available in the task library. The harness also uses this metadata, for example to determine the allowed watchdog time for the task.

This file is typically generated by the Makefile as part of the build process, although it can also be edited and committed directly to source control.

The following fields are recognized by Beaker. They are ordered first with mandatory fields then optional.

Description

`Description` (required) must contain exactly one string.

For example:

```
Description: This test tries to map five 1-gigabyte files with a single process.
Description: This test tries to exploit the recent security issue for large pix map files.
Description: This test tries to panic the kernel by creating thousands of processes.
```

License

`License` (required) identifies the type of license protection this task has. These include `GPL, GPLv2, GPLv2+, GPL3, Red Hat Internal, or Red Hat Confidential`. A sample configuration of this parameter appears as follows:

```
License:        GPLv2+
```

For further details on GNU Licensing, refer to http://www.gnu.org/licenses/. When 'Red Hat Internal or Confidential' is configured, this task rpm is not available for public use.

Name

`Name` (required) It is assumed that any result-reporting framework will organize all available tests into a hierarchical namespace, using forward-slashes to separate components of names (analogous to a path). This field specifies the namespace where the task results will be recorded in Beaker, and serves as a unique ID for the task. Since task names are used to define filesystem paths (for example, tasks will be installed under the path `/mnt/tests/<task-name>`), they are limited to characters that are usable within a file system path.

For fast indexing purposes, task names are limited to 255 characters.

Owner

`Owner` (required) is the person responsible for this task. Acceptable values are a subset of the set of valid email addresses, requiring the form: "Owner: human readable name <username@domain>".

Path

`Path` (required) specifies which directory the task rpm on the test system is installed. This is an informational attribute only and appears when displaying tasks when using the Task Library UI. If the user gets onto the test system, they can go to this directory to review their installed rpm data. To define this in the Makefile, it should appear as follows:

```
Path:           $(TEST_DIR)
```

TestTime

`TestTime` (required) represents the upper limit of time that the `runtest.sh` script should execute before being terminated. That is, the harness or lab controller should automatically abort the test after this time period has expired. This is to guard against cases where a test has entered an infinite loop or caused a system to hang. This field can be used to achieve better test lab utilization by preventing the test from running on a system indefinitely.

The value of the field should be a number followed by either the letter "m" or "h" to express the time in minutes or hours. It can also be specified in seconds by giving just a number. In most cases, it is recommended to provide a value in at least minutes rather than seconds.

The time should be the absolute longest a test is expected to take on the slowest platform supported, plus a 10% margin of error. Setting the time too short may lead to spurious cancellations, while setting it too long may waste lab system time if the task does get stuck. Durations of less than one minute are *not* recommended, as they usually run some risk of spurious cancellation, and it's typically reasonable to take a minute to abort the test after an actual infinite loop or deadlock.

For example:

```
TestTime: 90   # 90 seconds
TestTime: 1m   # 1 minute
TestTime: 2h   # 2 hours
```

TestVersion

(required) This is the version of the task provided in the loaded RPM. Configuration of this attribute should appear as follows where this is an initial release:

```
TestVersion:     0.1              # OR
TestVersion:     $(TESTVERSION)   # if defined in Makefile
```

Architectures

`Architectures` (optional) provides the ability to classify tasks for specific architectures. You can provide a list of excluded architectures or a list of exclusive architectures. For an excluded list, each architecture provided must be proceeded with a minus sign(-). This includes all architectures except those listed. For an exclusive list, no proceeding sign is required. You can only configure an excluded or exclusive list and not a combination of both.

If the task is expected to only run on x86_64 architecture, then configure the following:

```
Architectures: x86_64
```

If the task is expect to NOT run on architecture x86_64 nor i386, do as follows:

```
Architectures: -x86_64 -i386
```

The list of architectures you can choose from can be found in Distros Section.

Bugs

`Bugs` (optional) allows user to identify which bugs filed in Bugzilla are associated to this task. The following are sample configurations:

```
# Single Line
Bug: 9999999    OR Bugs: 9999999

# Multiple Bugzillas
Bugs: 77777777 88888888

# Or multiple Bugzillas on multiple lines
Bugs: 77777777
Bugs: 88888888
```

Destructive

`Destructive` (optional) is used to classify tasks which are destructive. Determination of what classifies as destructive is up to the user defining the test. To define this task as destructive, configure the following:

```
Destructive: Yes
```

Since tasks can be filtered by the *bkr task-list* CLI, it is recommended to define the task with this attribute with Yes or No; otherwise, it will not be found.

Environment

`Environment` (optional) is used to pass task environment data to test harnesses. The following can be set to alter defaults in beah:

```
Environment:     RHTS_PORT=<your chosen port> else beah uses random port(7080-7099)
```

This field can occur multiple times within the metadata. So you can configure the following:

```
Environment:    META_VAR1=<your var1 data>
Environment:    META_VAR2=<your var2 data>
Environment:    META_VAR3=<your var3 data>
```

Priority

`Priority` (optional) allows user to classify a task's priority. This has no affect on the execution of the job. Recommended values are as follows:

```
Low, Medium, Normal, High, Manual
```

The following is a sampling to configure this attribute:

```
Priority: High
```

Provides

`Provides` (optional) allows the task creator to specify the capabilities that the task RPM provides upon install. In addition to the default `Provides` generated by RPM, every task provides a virtual capability derived from the task name. For example, the `/distribution/check-install` task also provides `test(/distribution/check-install)`.

You can specify additional capabilities by adding new `Provides` lines (using a similar syntax to `Requires`). For example, if your task provides equivalent or better functionality than an old task, you can add a `Provides` such as the one below:

```
Provides: test(/old/task/name)
```

Releases

(optional) Some tests are only applicable to certain distribution releases. For example, a kernel bug may only be applicable to RHEL3 which contains the 2.4 kernel. Limiting the release should only be used when a task will not execute on a particular release. Otherwise, the release should not be restricted so that your test can run on as many different releases as possible.

You can populate the optional `Releases` field in two different ways. To exclude certain releases but include all others, list the releases each prefixed with a minus sign (-). To include certain releases but exclude all others, list the included releases.

For example, if your task runs only on RHEL6 and RHEL7:

```
Releases: RedHatEnterpriseLinux6 RedHatEnterpriseLinux7
```

Or, if your task is expected to run on any release except for RHEL3 & RHEL4:

```
Releases: -RedHatEnterpriseLinux3 -RedHatEnterpriseLinux4
```

Releases are identified by their OS major version. You can browse a list of OS versions in Beaker by selecting *Distros* → *Family* from the menu. For example:

- RedHatEnterpriseLinux3

- RedHatEnterpriseLinux4
- RedHatEnterpriseLinuxServer5
- RedHatEnterpriseLinuxClient5
- RedHatEnterpriseLinux6
- RedHatEnterpriseLinux7
- RedHatEnterpriseLinux8
- Fedora17

Your Beaker administrator may have configured compatibility aliases for some OS versions, which you can also use in the `Releases` field. Refer to Distro->Family selection in the Web User Interface to review *alias to OSMajor name* mapping or *admin-os-versions* in the Administration Guide to modify alias names.

Requires

`Requires` (optional) indicates one or more the packages that are required to be installed on the test machine for the test to work. The package being tested (if any) is automatically included via the `RunFor` field. Aside from the package under test and the test harness itself, anything `runtest.sh` needs for execution must be included here.

This field can occur multiple times within the metadata. Each value should be a space-separated list of package names, or of kickstart package group names preceded with an @ sign. Each package or group must occur within the distribution tree under test (specifically, it must appear in the `comps.xml` file).

For example:

```
Requires: gdb
Requires: @legacy-software-development
Requires: @kde-software-development
Requires: -pdksh
```

The last example above shows that we don't want a particular package installed for this test. Normally you shouldn't have to do this unless the package is installed by default.

In a lab implementation, the dependencies of the packages listed can be automatically loaded using yum.

Note that unlike an RPM spec file, only dependencies on actual package names are permitted (depending on a "virtual" provides is not supported — however, see *RhtsRequires* for a limited exception). Furthermore, even if some dependencies cannot be resolved, Beaker will attempt to execute the task anyway (this simplifies some issues with cross-version tasks as described below).

If a task dependency ever changes in a backwards incompatible way, one of the approaches below may be helpful:

- if only a dependency has changed name, specify both the names of dependencies in the `Requires:` field (enabling this is the reason that missing packages are silently ignored).
- it may be possible to work around the differences by logic in the section of the `Makefile` that generates the `testinfo.desc` file.
- for major changes, split the test, so that each incompatible version is handled by a separate task in a sub-directory, with the common files built from a shared directory in the `Makefile`.

When writing a multihost test involving multiple roles client(s) and server(s), the union of the requirements for all of the roles must be listed here.

RhtsOptions

(optional) You can indicate that your task does *not* need to be run inside the `rhts-compat` service:

```
RhtsOptions: -Compatible
```

This option has no effect on newer distros. See *The rhts-compat service*.

RhtsRequires

`RhtsRequires` (optional) indicates the other beaker tests that are required to be installed on the test machine for the test to work.

This field can occur multiple times within the metadata. Each value should consist of a task name in the form `test(<task-name>)`. Each task dependency named this way must exist in the Beaker task library or the task will be aborted.

For example:

```
RhtsRequires: test(/distribution/rhts/common)
```

RunFor

`RunFor` (optional) allows entries in the Beaker task library to be associated with specific packages for test execution and reporting purposes. It is only relevant for tasks that are specifically written as tests for particular packages rather than as general utilities.

When testing a specific package, that package should be listed in this field. If the test might reasonably be affected by changes to another package, the other package should also be listed here. If a package changes names but the task remains applicable, then all of the package's names should be listed here.

This field is optional and can occur multiple times within the metadata. The value should be a space-separated list of package names.

Type

(optional) To classify the type of task, one of the following is recommended:

```
Regression, Performance, Stress, Certification, Security,
Durations, Interoperability, Standardscompliance, Customeracceptance,
Releasecriterium, Crasher, Tier1, Tier2, Alpha,
KernelTier1, KernelTier2, Multihost, MultihostDriver, Install,
FedoraTier1, FedoraTier2, KernelRTTier1, KernelReporting, Sanity, Library
```

Configuration of this attribute should appear as follows if you've chosen `Sanity` as your `type`:

```
Type:            Sanity
```

3.7.4 Task Makefile targets

The following Makefile targets are defined by the `rhts-make.include` which is included by every task's Makefile. You can use these targets when updating your task.

The Makefile assumes your task is tracked by Git, Subversion, or CVS. Other version control systems are not supported. If you are developing a new task which is not hosted in any remote version control system, run `git init` to create a new local git repository to track your work.

`make tag` Tags the next release of the task. Run this after committing your changes, so that your updated task has a higher RPM Version-Release for uploading to Beaker.

`make rpm` Builds an RPM for this task. You can submit the RPM to Beaker, and it will be downloaded by the harness when a recipe includes this task.

`make bkradd` Builds an RPM and submits it to Beaker using *bkr task-add*.

3.7.5 Multihost tasks

Feature Requirements

The multihost feature is brought to you by Beaker and the test harness (e.g. Restraint). Multihosting is mostly performed on the test harness itself. This section provides an overview of how multihost works with the currently supported test harness. If your test harness does not support task synchronization, then this section can be ignored. If you are using only the test harness and not beaker, this section describes a work-around.

Introduction

Beaker has support for tasks that run on multiple hosts to test the interactions of a client and a server. When a multihost task is run in the lab, one or more machines will be allocated to each role in the test. Each machine has its own recipe and these recipes are defined within a recipeset in a single job.

Each machine under test may want to synchronize to ensure that they start the test together, and may need to synchronize at various stages within the test. While Beaker doesn't assign any particular semantics to role names (it just uses them to set the corresponding task environment variables), there are three common roles used in many multi-host tests: client, server and driver.

For many purposes, all you will need are client and server roles. For a test involving one or more clients talking to one or more servers, a typical approach would be for the clients to block while the servers get ready. Once all servers are ready, the clients perform the testing they need using the services provided by the server machines. The servers block waiting for clients to finish while in parallel carrying out work requested by the clients. In conclusion, the clients eventually report results back to the test. Once all clients have finished testing, the server tests unblock and report their results.

Each participant in a test will report results within the same job and report to different places within the result hierarchy. For example, the server part of the test may PASS if it survives the load, but the client part might FAIL upon, say, getting erroneous data from the server; this would lead to an overall FAIL for the test.

If you have a more complex arrangement, it is possible to have a driver machine which controls all of the testing.

All of the participants in a multihost test share a single `runtest.sh`, which performs every role within the test (e.g. the client role and server role). When a multihost test is run in the lab, the framework automatically sets environment variables to allow the various participants to know what their role should be, which other machines they should be talking to, and what roles those other machines are performing in the test. You will need to have logic in your `runtest.sh` to examine the role variables and perform the necessary role accordingly.

You can choose to perform your own synchronization or have the harness (e.g. restraint) perform task synchronization for you. If you want the harness to perform synchronization, the role=SERVERS and role=CLIENTS are required in your job recipes/tasks. The harness performs synchronization of servers and clients at the completion of the task. The tasks will all block at the end until the last one has finished. They will then move onto the next task but not in unison since some may be sleeping before rechecking the status of others.

Environment Variables for Synchronization

The following environment variables are shared by all instances of the runtest.sh within a recipeset for harness controlled synchronization. If you choose different role names than CLIENTS/SERVERS, your tasks are responsible for synchronization actions as described later. However, the harness with beaker's help, will still export role enviroment variables with your chosen names and wlll separate FQDNs with a space.

CLIENTS
: Contains a space-separated list of FQDNs of clients within this recipeset (that is, systems running recipes marked with the `CLIENTS` role in the job XML).

SERVERS
: Contains a space-separated list of FQDNs of servers within this recipeset (that is, systems running recipes marked with the `SERVERS` role in the job XML).

DRIVER
: Contains the FQDN for the driver of this recipe set, if any (that is, a system running a recipe marked with the `DRIVER` role in the job XML).

When writing your own synchronization, you can use the variable `HOSTNAME` in your `runtest.sh` to determine its identity. `HOSTNAME` is set by the harness task environment plugin, and is unique for each host within a recipeset. Your test can use `HOSTNAME` to search the client, server, or driver environment variables to determine its role. An example of this is shown later in *Handling Environment Variables in your synchronized runtest.sh*.

When you are developing your test outside the lab environment, only `HOSTNAME` is set for you (when sourcing the `rhts-environment.sh` script). One way to run multihost tests outside a Beaker instance is to copy the test to multiple development machines, set up `CLIENTS`, `SERVERS`, and `DRIVER` manually within a shell on each machine, and then manually run the `runtest.sh` on each one, debugging as necessary. Alternatively, support for standalone testing may be added directly to the test script, as described in *Standalone execution of multihost tests*.

Simple job.xml for Synchronization

In its simplest form, a job with multihost testing may look something like:

```
<job>
  <RecipeSet>
     <recipe role='MYSERVERS'>
        <task name='/distribution/check-install'/>
        <task name='/my/multihost/test'/>
     </recipe>
     <recipe role='MYCLIENTS'>
        <task name='/distribution/check-install'/>
        <task name='/my/multihost/test'/>
     </recipe>
  </RecipeSet>
</job>
```

Note: For brevity some necessary parts are left out in the above job definition. See *Job XML* for details.

As there is only one recipe in the recipe set with each defined role, submitting the job above will export environmental variables `MYSERVERS` and `MYCLIENTS` set to their respective FQDNs.

If you replace the roles with SERVERS/CLIENTS, the harness will perform synchronization operations for you. In this case, roles MYSERVERS/MYCLIENTS are used instead to begin showing you how to perform synchronization on your own without harness end-of-task synchronization.

Any multihost testing must ensure that the task execution order aligns correctly on all machines. This includes synchronization controlled by the harness. For example, the below will fail:

```
<recipe>
  <task role='STANDALONE' name='/distribution/check-install'/>
  <task role='STANDALONE' name='/my/test/number1'/>
  <task role='MYSERVERS'  name='/my/multihost/test'/>
</recipe>
<recipe>
  <task role='STANDALONE' name='/distribution/check-install'/>
  <task role='MYCLIENTS'  name='/my/multihost/test'/>
</recipe>
```

This will fail because the multihost test is the third test on the server side and it's the second test on the client side. To fix this, you can pad in dummy test cases on the side that has fewer test cases. There is a dummy test that lives in /distribution/utils/dummy for this purpose. So, the above can be fixed as:

```
<recipe>
 <task role='STANDALONE' name='/distribution/check-install'/>
 <task role='STANDALONE' name='/my/test/number1'/>
 <task role='MYSERVERS'  name='/my/multihost/test'/>
</recipe>
<recipe>
 <task role='STANDALONE' name='/distribution/check-install'/>
 <task role='STANDALONE' name='/distribution/utils/dummy'/>
 <task role='MYCLIENTS'  name='/my/multihost/test'/>
</recipe>
```

Handling Environment Variables in your synchronized `runtest.sh`

In the sample job.xml provided previously, the `runtest.sh` in `/my/multihost/test` test might look like:

```
Server() {
    # .. server code here
}

Client() {
    # .. client code here
}

if test -z "$JOBID" ; then
    echo "Variable jobid not set! Assume developer mode"
    MYSERVERS="test1.example.com"
    MYCLIENTS="test2.example.com"
    DEVMODE=true
fi

if [ -z "$MYSERVERS" -o -z "$MYCLIENTS" ]; then
    echo "Can not determine test type! Client/Server Failed:"
    RESULT=FAILED
    report_result $TEST $RESULT
fi

if $(echo $MYSERVERS | grep -q $:envvar:`HOSTNAME`); then
    TEST="$TEST/Server"
    Server
fi

if $(echo $MYCLIENTS | grep -q $:envvar:`HOSTNAME`); then
    TEST="$TEST/Client"
```

```
    Client
fi
```

We have `Server()` and `Client()` functions which will be executed by recipes with the `MYSERVERS` and `MYCLIENTS` role respectively.

Then we test for `JOBID` which indicates if the script is running inside a Beaker instance; otherwise, it's being run on the test developer's local workstation or any other non-Beaker system.

The tests comparing `MYSERVERS` and `MYCLIENTS` to `HOSTNAME` determine what code to run on this particular machine. As mentioned before, since only one recipe in our recipe set uses each role, the `MYSERVERS` and `MYCLIENTS` environmental variables will be set to their respective machines' names and exported on both machines.

Writing User-Defined Synchronization in your `runtest.sh`

For most meaningful multi-host tests, there has to be some sort of coordination and synchronization between the machines and the execution of the test code on both sides. While in some cases, this may be handled by a dedicated recipe with the `MYDRIVER` role, the restraint harness offers two utilities for this purpose: *rstrnt-sync-set* and *rstrnt-sync-block*.

The *rstrnt-sync-set* command is used to set a state on a machine. The *rstrnt-sync-block* command is used to block the execution of the task until a certain state on certain machine(s) is reached. Those familiar with parallel programming can think of this as a barrier operation. A brief overview of the usage of these utilities follows:

- **rstrnt-sync-set**: This command sets the state of the current machine to an arbitrary text string.

  Syntax: `rstrnt-sync-set -s STATE`

- **rstrnt-sync-block**: This command blocks execution and doesn't return until the specified `STATE` is set on the specified machine(s).

  Syntax: `rstrnt-sync-block -s STATE [-s STATE1 -s STATE2] machine1 machine2 ...`

For details on the options provided by the restraint harness, refer to Restraint Command documentation and search for these commands.

The role related environment variables are useful here as they contain the hostnames of all recipes in the recipeset with that role. For example, you can wait for all recipes with the `MYSERVERS` role to set their state to `"READY"` by running:

```
rstrnt-sync-block -s READY $MYSERVERS
```

By default the **rstrnt-sync-block** utility will block until the local or external watchdog is triggered if the expected state is never achieved. If this behavior isn't desired, the *–timeout* option can be used instead. In that case, a zero return code indicates that the desired state was reached, while a non-zero return code indicates the operation timed out.

These commands require a bit of manual intervention when run in the standalone execution environment for Beaker task development, as the Beaker lab controller normally coordinates the barrier operation. See *Standalone execution of multihost tests*.

Sample `runtest.sh` for a user synchronized multihost task

```
#!/bin/sh
# Source the common test script helpers
. /usr/bin/rhts_environment.sh

# Save STDOUT and STDERR, and redirect everything to a file.
```

```
exec 5>&1 6>&2
exec >> "${OUTPUTFILE}" 2>&1

client()
{
    echo "-- wait the server to finish."
    rstrnt-sync-block -s "DONE" ${MYSERVERS}

    user="finger1"
    for i in ${MYSERVERS}
    do
        echo "-- finger user \"$user\" from server \"${i}\"."
        ./finger_client "${i}" "${user}"
        # It returns non-zero for failure.
        if [ $? -ne 0 ]; then
            rstrnt-sync-set -s "DONE"
            report_result "${TEST}" "FAIL" 0
            exit 1
        fi
    done

    echo "-- client finishes."
    rstrnt-sync-set -s "DONE"
    result="PASS"
}

server()
{
    # Start server and check it is up and running.
    /sbin/chkconfig finger on && sleep 5
    if ! netstat -a | grep "finger" ; then
        rstrnt-sync-set -s "DONE"
        report_result "${TEST}" "FAIL" 0
        exit 1
    fi
    useradd finger1
    echo "-- server finishes."
    rstrnt-sync-set -s "DONE"
    rstrnt-sync-block -s "DONE" ${MYCLIENTS}
    result="PASS"
}

# ---------- Start Test -------------
result="FAIL"
if echo "${MYCLIENTS}" | grep "${:envvar:`HOSTNAME`}" >/dev/null; then
    echo "-- run finger test as client."
    TEST=${TEST}/client
    client
fi
if echo "${MYSERVERS}" | grep "${:envvar:`HOSTNAME`}" >/dev/null; then
    echo "-- run finger test as server."
    TEST=${TEST}/server
    server
fi
echo "--- end of runtest.sh."
report_result "${TEST}" "${result}" 0
exit 0
```

Standalone execution of multihost tests

Multihost tests can be more easily executed outside a Beaker instance by altering their behavior based on the JOBID variable (or any other documented variable which is set when running inside a Beaker instance).

For a two machine test that uses the CLIENTS and SERVERS roles, you could create a pair of local virtual machines and add the following lines at the beginning of your runtest.sh script:

```
# decide if we're running standalone or in a Beaker instance
if test -z $JOBID ; then
        echo "Variable JOBID not set, assuming standalone"
        CLIENTS="client-vm.example.com"
        SERVERS="server-vm.example.com"
else
        echo "Variable JOBID set, we're running inside Beaker"
fi
echo "Clients: $CLIENTS"
echo "Servers: $SERVERS"

# ... rest of test script
```

3.7.6 The rhts-compat service

On Red Hat Enterprise Linux 6 and earlier releases, tasks are executed by a special service called rhts-compat. This service is last in the startup order and it intentionally "hangs" forever without fully starting up, so that tasks can be run attached to the console. As a consequence, when the rhts-compat service is running it is not possible to log in to the system's console.

If you don't need this compatibility mode for your tasks, you can disable it in the *task metadata* by adding:

```
RhtsOptions: -Compatible
```

Or you can disable it in your Beaker job definition by adding a task parameter:

```
<task ...>
  <params>
    <param name="RHTS_OPTION_COMPATIBLE" value=""/>
  </params>
</task>
```

The service can also be disabled system-wide by setting an environment variable in /etc/profile.d/task-overrides-rhts.sh, for example using a kickstart snippet:

```
<ks_append>
%post
cat >>/etc/profile.d/task-overrides-rhts.sh <<EOF
export RHTS_OPTION_COMPATIBLE=
export RHTS_OPTION_COMPAT_SERVICE=
EOF
%end
</ks_append>
```

On distros using systemd (which includes Red Hat Enterprise Linux 7 and all supported Fedora releases) the rhts-compat service cannot be used and is always unconditionally disabled.

3.8 Tasks provided with Beaker

Besides the custom tasks which Beaker users would write for a specific testing scenario, there are a number of tasks which are distributed and maintained along with Beaker. Among these, the `/distribution/check-install` and `/distribution/reservesys` tasks are essential for Beaker's operation. The `/distribution/inventory` task is not essential for Beaker's operation, but it is required for accurate functioning of Beaker's ability to schedule jobs on test systems meeting user specified hardware criteria. The `/distribution/beaker/dogfood` task runs Beaker's test suite (hence, the name *dogfood*) and is perhaps only useful for meeting certain specific requirements of the Beaker developers.

3.8.1 /distribution/check-install

The purpose of this task is to report back on the system install (provisioning). It is usually added before any scenario specific tasks so that it is run immediately after the system has been provisioned.

This task collects and reports various information about the installed system which may be useful in debugging any problems with the installer or the distro.

3.8.2 /distribution/reservesys

The `/distribution/reservesys` task reserves a system for a specific time frame to aid post-test analysis. You would usually append this task in your recipe so that the system is available for you to login after the other tasks have been run, but before the system is returned to Beaker. *Reserving a system after testing* describes system reservation in detail.

3.8.3 /distribution/inventory

The `/distribution/inventory` task is useful for the administrator of a Beaker installation to gather detailed hardware data about Beaker's test systems. Hardware devices which are probed include disk drives, graphics hardware and network devices. When this task is run on a test system, it retrieves this information and sends it to the Beaker server where it is updated in the main database.

This data can then be used by Beaker to schedule a job for which a specific hardware requirement may have been specified (See: *device specification in jobs*). Hence, it is a good idea to run this task on every system to ensure that the hardware details are correctly updated in Beaker's database.

3.8.4 /distribution/command

The `/distribution/command` task runs an arbitrary shell command, given in the `CMDS_TO_RUN` parameter.

This task is useful for inserting ad hoc tests or behaviour into a recipe for experimentation purposes, without needing to modify an existing task or write a new one.

For example, to log the CPU information of the system under test:

```
...
<task name="/distribution/command">
    <params>
        <param name="CMDS_TO_RUN" value="cat /proc/cpuinfo" />
    </params>
</task>
...
```

3.8.5 /distribution/beaker/dogfood

The `/distribution/beaker/dogfood` task runs Beaker's test suite (unit tests and selenium tests) on a test system. It can be configured to either run the tests from the development branch of Beaker or the most recent released version.

This task is used by the Beaker developers to run the test suite every time a new patch is pushed to the development branch to help prevent any regressions in the code base.

3.8.6 /distribution/utils/dummy

This is a placeholder task used to align task execution across different recipes in a multi-host recipe set. See *Multihost tasks* for details.

3.8.7 /distribution/virt/install

The `/distribution/virt/install` task is responsible for installing a virtual machine (defined as a 'guest recipe' in Beaker). It does this via `virt-install`. The task is defined in the host recipe, often along with `/distribution/virt/start`. For example:

```
<task name="/distribution/check-install" role="SERVERS">
  <params/>
</task>
<task name="/distribution/virt/install" role="SERVERS">
  <params/>
</task>
<task name="/distribution/virt/start" role="SERVERS">
  <params/>
</task>
```

Be aware that `/distribution/virt/start` and `/distribution/virt/install` should never be defined in the guest recipe itself.

3.8.8 /distribution/virt/image-install

This task is an experimental alternative to the regular `/distribution/virt/install` task for installing guest recipes. Rather than booting the installer inside the guest and running through a complete installation, this task fetches a cloud image and boots that.

The `CLOUD_IMAGE` task parameter should be the URL for a suitable cloud image. The image must have the `cloud-init` package pre-installed and enabled. This task approximates the effect of the guest kickstart by generating a suitable user-data file for cloud-init.

Note that there are a number of limitations when using this task:

- The distro tree selected by Beaker for the guest recipe is effectively ignored. The distro used in the guest is determined solely by what image is given.
- Similarly, it is the job submitter's responsibility to use a suitable local mirror for the cloud image. (Fetching the image over an expensive WAN link is not desirable but Beaker will not prevent it.)
- Not all parts of the guest kickstart are accurately applied, since the installer is skipped. The task extracts `%packages` and `%post` sections, and it also handles the `repo`, `rootpw`, and `selinux` commands.

3.8.9 /distribution/virt/start

The `/distribution/virt/start` task is used for starting a virtual machine, via `virsh start`. Please see */distribution/virt/install* for examples on how to use it with `/distribution/virt/install`.

3.8.10 /distribution/rebuild

This task is for experimental mass rebuilds of an entire distribution from source, for example using a newer or modified build toolchain. It fetches source RPMs from a given yum repo and rebuilds them all in mock.

Packages are rebuilt in alphabetical order. This task does not attempt to build packages in dependency order, nor does it inject the build results back into the build root.

The following task parameters are accepted:

SOURCE_REPO URL of the yum repo to fetch source RPMs from.

MOCK_REPOS Space-separated list of URLs of the yum repos to include in the build root. Typically this should include the entire distribution or the build tag for it. You can also add extra repos containing patched packages.

MOCK_CHROOT_SETUP_CMD Command to be run when mock sets up the chroot. The default value is suitable for Fedora: `install @buildsys-build`. The group name may need adjusting for other distros.

MOCK_TARGET_ARCH Target architecture for builds. By default this will match the arch of the recipe where this task is running.

MOCK_CONFIG_NAME Name of the mock configuration to use or generate (excluding `.cfg` file extension). If this parameter is set and the configuration exists, it will be used as is. Otherwise the configuration will be generated based on the parameters above.

SKIP_NOARCH If set to a non-empty value, skip building any SRPMs which produce only noarch packages.

KEEP_RESULTS If set to a non-empty value, keep the results (RPMs and log files) produced by each build in `/mnt/tests/distribution/rebuild/results/`*`packagename`*`/`. You can use a subsequent task in the recipe to examine the results or copy the RPMs elsewhere.

SRPM_BLACKLIST SRPMs to skip. This parameter must be a whitespace-separated list of bash glob patterns. Each pattern is matched against the SRPM filename (including .src.rpm extension). If any pattern matches, the SRPM is skipped. For example `kernel*` will skip any SRPMs beginning with kernel.

SRPM_WHITELIST SRPMs to build. If this parameter is set, any SRPM which does not match a pattern in the whitelist is skipped. Similar to `SRPM_BLACKLIST`, this must be a whitespace-separated list of bash glob patterns.

As an example, imagine you have built the latest GCC version 99.0, and you want to try rebuilding all architecture-specific packages in Fedora 21 using the new compiler to see if it introduces any build failures:

```
<task name="/distribution/rebuild" role="STANDALONE">
    <params>
        <param name="SOURCE_REPO"
               value="http://dl.fedoraproject.org/pub/fedora/linux/releases/21/Everything/source/SRPM
        <param name="MOCK_REPOS"
               value="http://dl.fedoraproject.org/pub/fedora/linux/releases/21/Everything/x86_64/os/
                      http://example.com/my-gcc99-test-repo/" />
        <param name="SKIP_NOARCH" value="1" />
    </params>
</task>
```

3.8.11 Task source code

The source code for the above tasks can be found in the Beaker core tasks git repo. The tasks for testing Beaker itself are in the Beaker meta tasks git repo.

3.9 Troubleshooting

3.9.1 Why is my job aborted earlier than the specified reserve time?

A reason can be the hard limit of **99 hours/4 days** in `/distribution/reservesys`. Check your job XML if you're trying to reserve the system for longer. You can also check the result for a warning: `watchdog_exceeds_limit`. *Reserving a system after testing* describes system reservation in detail.

3.9.2 How to archive XML file for a job from within a Beaker job?

You can use the *harness API*:

```
wget ${BEAKER_LAB_CONTROLLER_URL}recipes/$BEAKER_RECIPE_ID
```

3.9.3 Troubleshooting checklist for an aborted job

1. Has the first task in the recipe passed?

   Yes. Installation has passed in this case, beaker performed provisioning and test harness deployment well. Please check the particular task with failure if it is working properly.

   No, Recide ID does not match any systems. There isn't any lab containing both requested distribution and requested machine. Try to change the requirements of your job and/or order suitable hardware.

   No, Command XYZ failed . Power management isn't working (machine can't be rebooted/switched on/switched off). This error is handled automatically by Beaker - it marks the machine as broken and notifies its administrator.

   No, External Watchdog Expired. Something went wrong during deployment. Check for status of other tasks in queue.

2. Are all jobs/recipes aborted?

   Yes. There is very likely something wrong with Beaker. Report it to your Beaker administrator.

   No. Go back to your failed job/recipe and start troubleshooting, check the console output first.

3. Is the console output missing/empty?

   Note: An empty console.log does not have to be necessarily blank. It can be considered *blank* if it contains a few time stamps.

   Yes. Either the system is stuck and cannot power on, or its serial console is not configured properly. No further diagnostics are possible, report this problem to the system owner.

   No. Great! Look at the bottom of the console output.

4. Does the bottom of the console output contain errors?

Note: Look carefully for possible causes. Typically, the bottom of an aborted job contains a few lines with time stamps and some useful information above these lines. This is the last information printed. This information can be hard to read as it was part of an ncurses dialog utilizing ANSI escape codes. Beaker removes escape characters in the console output, but the rest of the sequence stays there.

Yes, there is some problem with:

- the *disk space allocation*
- the *network*
- Python traceback from anaconda. Please report a bug to anaconda

No. Check if machine initiated installation at all.

5. Is ksdevice=eth[0-9] set for the machine?

   New RHEL and Fedora distributions use different naming scheme for network interfaces based on physical placement. Typically these names look like 'em1' or 'p3p1'. So far it may happen that beaker inventory defines ksdevice option with some traditional ethX name. To check this:

   - look at the NIC name at the bottom of the console output and
   - search for ksdevice= in anaconda.log

   If you see new names in the console output and traditional name as ksdevice, you're affected by this issue.

6. Did the machine boot into an existing installation, instead of running the installer?

   The boot order might have changed on the machine to boot from disk. In this case, it will never start the installation. In this case:

   - console.log is the only available log file,
   - the console output contains exclusively output of a boot loader (typically grub) and messages of boot from disk, there is no message related to boot from network, no message from Anaconda installer.

   In case the installation has started, check result of *installation of harness*.

7. Did the Beaker harness successfully install?

   To check this, open the console output and search for string `beah`. In case of successful installation you will find yum output with successful installation of several packages, beah is one of this.

   Some occurrences 'beah' substring were found. Check repositories added via kickstart.

   Were all repositories setup properly? It may happen that repositories are unavailable. To recognize it open the console output and search for string 'Trying other mirror'. If you find message informing you about some repomd.xml: [Errno 14] HTTP Error 404: Not Found then you found it.

   There are some inaccessible repositories. Somebody added to a kickstart of your job incorrect path to the repository. If you did it (you should know that), please fix.

3.9.4 Network issue during installation

If there was an error to configure the network interface, check if the `ksdevice` *value is correct*.

3.9.5 Error: `Could not allocate requested partitions`

1. First check your job XML. Is `<hostRequires />` requesting a system with a large enough disk to satisfy the `<partition />` you have requested?
2. Check the kernel output to ensure all expected disks and storage controllers are present. If any are missing, report a bug against the distro.

   Some storage controllers are unsupported in some distros. For example, the `cciss` driver was dropped from newer kernels and so any systems using an older SmartArray controller cannot be provisioned with RHEL7 or Fedora. In this case, the system owner should update the *Excluded Families* settings to exclude unsupported distros from their system. Report this problem to the system owner.

3.9.6 Error: `Cannot access /var/run/beah/rhts-compat/launchers: No such file or directory`

1. Check the network connection, it is most likely offline.
2. Some kind of installation error occurred, which is *fixed* by rebooting the host.
3. The system is incompatible with rhts-compat, so it must be disabled in the job XML.

/

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

–any
    rhts-sync-block command line option, 58
–timeout <timeout>
    rhts-sync-block command line option, 58
-s <state>
    rhts-sync-block command line option, 58
-t <type>
    rhts-abort command line option, 56
-u <user>
    rhts-run-simple-test command line option, 58

A

absolute_url() (built-in function), 52
arch() (built-in function), 52

C

chr() (built-in function), 52
CLIENTS, 68, 72

D

dictsplit() (built-in function), 51
distro (built-in variable), 52
distro_tree (built-in variable), 53
DRIVER, 68

E

environment variable
    ARCH, 59
    BEAKER, 60
    BEAKER_JOB_WHITEBOARD, 60
    BEAKER_RECIPE_WHITEBOARD, 60
    CLIENTS, 68, 72
    DISTRO, 59
    DRIVER, 68
    FAMILY, 59
    GUESTS, 60
    HOSTNAME, 68, 70
    HYPERVISOR_HOSTNAME, 60
    JOBID, 59, 70, 72
    KILLTIME, 59
    LAB_CONTROLLER, 60
    MYCLIENTS, 68, 70
    MYDRIVER, 70
    MYSERVERS, 68, 70
    REBOOTCOUNT, 57, 60
    RECIPE_MEMBERS, 60
    RECIPE_ROLE, 60
    RECIPEID, 59
    RECIPESETID, 59
    RECIPETESTID, 60
    RECIPETYPE, 60
    RESERVE_IF_FAIL, 25, 26
    RESERVETIME, 25
    ROLE, 60
    SERVERS, 68, 72
    SUBMITTER, 59
    TASKID, 59, 60
    TASKNAME, 59
    TEST, 59, 61
    TESTID, 60
    TESTORDER, 60
    TESTPATH, 59, 61
    TESTRPMNAME, 59
    VARIANT, 59

H

HOSTNAME, 68, 70

J

job_whiteboard (built-in variable), 53
JOBID, 70, 72

K

kernel_options_post (built-in variable), 53
ks_appends (built-in variable), 53

L

lab_controller (built-in variable), 53

M

MYCLIENTS, 68, 70
MYDRIVER, 70
MYSERVERS, 68, 70

N

netaddr (built-in variable), 53

O

ord() (built-in function), 53
osmajor() (built-in function), 52
osversion() (built-in function), 52

P

parsed_url() (built-in function), 51

R

re (built-in variable), 53
REBOOTCOUNT, 57
recipe (built-in variable), 53
recipe_whiteboard (built-in variable), 53
RECIPETESTID, 60
RESERVE_IF_FAIL, 25, 26
RESERVETIME, 25
rhts-abort command line option
    -t <type>, 56
rhts-run-simple-test command line option
    -u <user>, 58
rhts-sync-block command line option
    –any, 58
    –timeout <timeout>, 58
    -s <state>, 58

S

SERVERS, 68, 72
shell_quoted() (built-in function), 52
snippet() (built-in function), 53
split() (built-in function), 52

T

TASKID, 60
TASKNAME, 59
TEST, 61
TESTID, 60
TESTPATH, 61

U

urljoin() (built-in function), 52

V

var() (built-in function), 54