

Documentation 0.1

Beaker guide

short description



Shikha Nansi

Documentation 0.1 Beaker guide

short description

Edition 0

Author

Shikha Nansi

snansi@redhat.com

Copyright © 2010 | You need to change the HOLDER entity in the en-US/Beaker_guide.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

All other trademarks are the property of their respective owners.

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

A short overview and summary of the book's subject and purpose, traditionally no more than one paragraph long. Note: the abstract will appear in the front matter of your book and will also be placed in the description field of the book's RPM spec file.

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	v
1.2. Pull-quote Conventions	vi
1.3. Notes and Warnings	vii
2. Getting Help and Giving Feedback	vii
2.1. Do You Need Help?	vii
2.2. We Need Feedback!	viii
A. Revision History	1
1. Introduction	3
1.1. Abstract	3
1.2. Introduction	3
1.3. Background	3
1.4. Beaker Overview	4
1.4.1. Components	5
1.4.2. Topology	5
2. Installation	7
2.1. Install Beaker	7
2.1.1. Disabling Repos	7
2.1.2. Install DB	7
2.1.3. Start Beaker	8
2.2. Setup Lab Controller	9
2.2.1. Install Lab Controller	9
2.2.2. Configure Lab Controller	9
2.3. Beaker Client	11
3. User Guide	15
3.1. Introduction	15
3.2. Getting Started	15
3.2.1. Process	15
3.2.2. Checklist Discussed	62

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)¹ set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F1** to switch to the first virtual terminal. Press **Ctrl+Alt+F7** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

¹ <https://fedorahosted.org/liberation-fonts/>

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' won't cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, you can find help in the following ways:

Red Hat Knowledgebase

Visit the Red Hat Knowledgebase at <http://kbase.redhat.com> to search or browse through technical support articles about Red Hat products.

Red Hat Global Support Services

Your Red Hat subscription entitles you to support from Red Hat Global Support Services (GSS). Visit <http://support.redhat.com> for more information about obtaining help from GSS.

Other Red Hat documentation

Access other Red Hat documentation at <http://www.redhat.com/docs>

Red Hat electronic mailing lists

Red Hat hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any list to subscribe to that list or to access the list archives.

2.2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/> against the product **Documentation**.

When submitting a bug report, be sure to mention the manual's identifier: *Beaker_guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Appendix A. Revision History

Revision 0 Tue Aug 17 2010

Dude McPants Dude.McPants@example.com

Initial creation of book by publican

Introduction

1.1. Abstract

- This is a short introduction to Beaker, designed for quick-learning the essential skills for automated test case running.
- It will provide you with all the necessary steps to create and manage customized 'tests', while keeping the instructions as brief as possible.
- By the end you will probably realize that creating a new Beaker test is much easier than you might expect.

1.2. Introduction

Purpose of Beaker

There are many reasons why you might want to use Beaker Perhaps you:

- are a developer and want to create regression tests for packages you maintain
- want to automatically run a particular test on different architectures and versions of RHEL and see if the results are the same
- are a quality engineer with testing responsibility for a particular package, RHEL/Fedora release.
- are a partner with special hardware that you would like to test RHEL on.
- want a simple way to reproduce and test customer issues and help to make sure they do not happen again.

Bug reporting

Bug reporting does the following:

- Report bugs in Bugzilla component.

1.3. Background

Beaker is a new-project written in python which aims to achieve separation of concerns between inventory management and test execution.

- Beaker provides an automated software testing system that should appeal to a variety of audiences for a variety of purposes. It provides a programming interface for developing automated unit tests, bug reproducers, hardware enablement, and regression tests. Beaker is composed of several components, but the most primary are the Beaker framework for writing the tests and the tests themselves. Everyone is encouraged to submit new tests to the Table Cloth repository and/or download stored tests for their use.
- Most developers create new tests in a standalone environments (such as a workstation) first and later deploy them in the automated test lab where they are run on machines dedicated for testing. Many types of automated environments are possible:
 - running on a farm of real machines in some kind of lab situation

- running as part of a continuous checkout/build/test "tinderbox"-style environment
- running inside a fake "chroot" tree on your development machine
- When tests are developed in accordance with the Beaker framework, they can be run either standalone or in a lab test environment without modification. The Beaker framework defines both an API and the format of required files that contain meta data. When run locally from a command line, the results are reported to **stdout**. When deployed in the automated test lab, the results can be stored in a repository and viewed via a UI.
- This section will first describe how to write tests within the Beaker framework. Later, once a new test is written and tested, it can be packaged, as a conventional RPM package from its source files, and submitted to the Table Cloth test repository, using Subversion source control system commands. This allows the test to be downloaded by others, but in addition, it also allows the test to be run in a Fedora lab environment. The Fedora project has a dedicated test lab environment where tests stored in the test repository can be run and the results of the tests made public. The name of this lab is Beaker. The reporting of the test results will be done with the Testify UI.
- A test lab has the advantage of being able to automatically schedule the test and collect test results on a variety of architectures and releases. The lab environment is also able to detect certain types of failure. For example, if a dedicated test machine goes into an infinite loop, the test is automatically killed and a failure is reported.

1.4. Beaker Overview

Beaker is the primary test harness and test deck used to test RHEL. It can be divided into two parts: a test scheduler and individual tests.

Tests

At its most basic level, a Beaker test is a program that attempts to perform a task or a series of tasks, and upon completion, determines success or failure and reports the results provided by hooks in the API. Care should be taken when creating the test to make sure all test outcomes are reported properly. A test can consist of code, data and meta data, as well as defining dependencies on other packages (if necessary). The Beaker framework is provided by a series of packages where tools, API libraries, and template files will be installed to the local workstation.

Test scheduler

The test scheduler manages the complex job of coordinating the farm machines set aside to run individual tests. It handles all aspects of test execution, ranging from machine selection, distribution installation, fencing (rebooting and reinstalling hosts that have exceeded their scheduled time), and coordinating tests which require multiple hosts to participate with each other. The scheduler also coordinates how tests are launched.

Individual tests

Tests make up one of the most important components of s/RHTS/Beaker. Individual tests are written in a format understood by the scheduler so that they can be automatically run by the scheduler on a variety of distributions, for example, RHEL3, RHEL4, RHEL5, and Fedora; and architectures, for example, i386, ia64, ppc, s390, s390x, and x86_64.

Tests written in the Beaker format can be launched from a lab controller or from a command line if the `rhts-devel` packages are installed on a local workstation.

Beaker has two major sub-divisions:

- Components
- Topology

1.4.1. Components

Beaker is an Open Source automated testing framework, consisting of the following core parts:

- **Lab Controller:** The Lab Controller maintains inventory data about distros available to install and machines to install on. It can be used by itself or in conjunction with the Beaker server. The Lab Controller is the only conduit of communication between the Lab Machines and the Beaker Server. The Lab Controller is built on top of several existing tools:
 - **Cobbler:** Does the actual interactions with the test systems (install distro etc).
 - **Conserver:** Provides console logging
 - **Fence-agents:** Power cycles machines to start PXE installs and to recover.
 - **Smolt:** Provides the inventory data. That is, the hardware data of the test systems.
- **Beaker server** The Beaker server is the central point at which all Job related activity occurs. System inventory as well as the ability to provision Systems is also controlled from here. It also holds the repository of Tasks.
- **Beaker Client** The shell based client (CLI) provides users with a subset of functions available in the Beaker web app, plus a few functions that the Beaker web app does not provide.
- **Beah test harness** Beaker needs a test harness to be responsible for executing the tasks on the system, currently it uses Beah, although theoretically any test harness could be used. It runs locally on the provisioned Systems.

1.4.2. Topology

Beaker's topology is relatively simple. The Beaker server acts as the interface through which all user actions are performed. Some of the requests are performed local to the Beaker server (such as scheduling Jobs and creating reports), but other requests are forwarded to the provisioned System (i.e. running of tasks). This communication between the Beaker server and the System occurs via the Lab Controller.

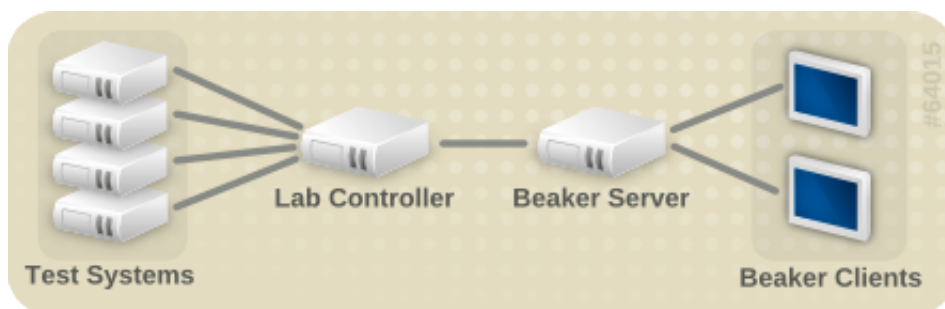


Figure 1.1. High level network topology of Beaker's components

Installation

The installation guide will teach you how to install Beaker into your system.

2.1. Install Beaker

Add the beaker.repo file that Red Hat provided for you on the machine(s) that will be running your Scheduler and Lab Controller. The following should be done as root.

2.1.1. Disabling Repos

Disable other repos to ensure that packages are installed from the beaker repo.

Install the dependencies

```
$ yum -y install rpm-build python-devel TurboGears python-TurboMail  
$ yum -y install mod_wsgi python-decorator python-tgexpandingformwidget python-xmltramp
```

Install Beaker server.

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

2.1.2. Install DB

Beaker supports MySQL, MSSQL, Oracle, MaxDB, PostgreSQL, and SQLite. For this tutorial, we will use MySQL. First, make sure MySQL server is installed, and configure the daemon to run at startup.

```
$ yum install -y mysql-server MySQL-python  
$ chkconfig mysqld on  
$ service mysqld start
```

Create a database, and grant access to beaker user. You can put the database on the local machine, or on a remote machine. In the example below, the database is hosted on the local machine.

```
$ echo "create database beaker;" | mysql  
$ echo "grant all on beaker.* to 'beaker'@'localhost' IDENTIFIED BY 'beaker';" | mysql
```

Now let's initialise our DB with tables. We'll also create an admin account called *admin* with password *testing*.

Table type

Beaker requires a transactional DB. If using a Database that default to something else (i.e MySQL defaults to MyISAM), either the default table type for the database needs to be changed (to InnoDB in

MySQL's case) or the user will have to convert the tables to a transactional DB after running beaker-init.

You can change the default storage engine for mysql by editing the file `/etc/my.cnf` and adding the following line in the `[mysqld]` section.

```
default-storage-engine=INNODB
$ beaker-init -u admin -p testing -e admin_email_address
```

2.1.3. Start Beaker

We are now ready to start the Beaker service. Make sure you have the following line in your `/etc/httpd/conf.d/wsgi.conf` and that it is uncommented.

```
LoadModule wsgi_module modules/mod_wsgi.so
```

First make sure apache is on and configured to run on startup.

```
$ sudo chkconfig httpd on
$ sudo /sbin/service/httpd start
```

We need to switch SELinux off.

```
$ setenforce 0
```

Due to permission issues, we need to delete the log file before we start Beaker for the first time. Otherwise Beaker will not run properly.

```
$ sudo rm /var/log/beaker/server*.log
```

Start Beaker and configure it to run on startup.

```
$ sudo chkconfig beakerd on
$ sudo /sbin/service beakerd start
```

To make sure Beaker is running go to <http://BeakerServer.example.com/bkr/> in your browser

Add Lab Controller details

One more step that we need to do is add the Lab Controller that we are yet to configure [Section 2.2, "Setup Lab Controller"](#). Login in at <http://BeakerServer.example.com/bkr/labcontrollers/new>. Use the username and password above from the beaker-init command. The new lab controller form requires 3 fields

- **FQDN:** This is the fully qualified domain name of the lab controller.
- **Username:** This is the login name we will use for xmlrpc, for the purposes of this document will use the login name *testing*
- **Password:** This is the password that goes along with the username, again we will use : *testing*

The new lab controller also needs a user account. Login at:

```
http://BeakerServer.example.com/bkr/users/new
```

- **Login:** Should be the host/FQDN (FQDN is the fully qualified domain of the lab controller).
- **Display Name:** Should be just FQDN.
- **Email address:** The email address should be root@FQDN.
- **Password:** If you are not using Kerberos authentication, you will need a password here.

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

2.2. Setup Lab Controller

Beaker uses Lab Controllers to manage the Systems in it's inventory. Open a terminal window on the system you will be running the Lab Controller on. This can be a separate system than the one running the Beaker server.

2.2.1. Install Lab Controller

If you have not installed the beaker repo on the Lab Controller, see [Section 2.1, "Install Beaker"](#). Follow the instruction to install Beaker repo. Install the Lab Controller rpm. These dependencies are needed to make the rpm.

```
$ sudo yum -y install rpm-build python-devel TurboGears
```

To install the Lab Controller, enter the following.

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

2.2.2. Configure Lab Controller

Cobbler is one of the dependencies that is installed with the Lab Controller. You'll need to edit the `/etc/cobbler/settings` file.

- **server:** This needs to be set to the Lab Controllers Fully qualified domain name.
- **next_server:** If you use cobbler as your dhcp server this needs to be the ip address of the Lab Controller.
- **pxe_just_once:** 1

- **anamon_enabled:** 1
- **redhat_management_server:** <https://login:password@BeakerServer.example.com/bkr>. login would be *admin*, password would be *testing*, and BeakerServer would be the *HOSTNAME* of the Beaker/Server you installed earlier. If your Lab Controller is on the same machine as your Beaker server, the values should be <https://login:password@BeakerServer/bkr>

You will need to enable an auth method in `/etc/cobbler/modules.conf`

- Change `module = auth_denyall` to `module = authn_testing`
- `authn_testing` gives a login of testing password testing
- If you create proper accounts, make sure they match what you entered in <http://BeakerServer/labcontrollers/new>

If you are using SELinux, do the following.

```
$ sudo setsebool -P httpd_can_network_connect true
$ sudo semanage fcontext -a -t public_content_t "/var/lib/tftpboot/.*"
$ sudo semanage fcontext -a -t public_content_t "/var/www/cobbler/images/.*"
```

Turn on http

```
$ sudo chkconfig httpd on
$ sudo service httpd start
```

Turn on tftp

```
$ sudo chkconfig xinetd on
$ sudo chkconfig tftp on
$ sudo service xinetd start
```

Turn on cobbler

```
$ sudo chkconfig cobblerd on
$ sudo service cobblerd start
```

Enable and turn on beaker watchdog proxy

```
$ sudo chkconfig beaker-watchdog on
$ sudo chkconfig beaker-proxy on
$ sudo service beaker-watchdog start
$ sudo service beaker-proxy start
```

Cobbler should now be running.

You'll need to import some distros. You can use the following command (whilst replacing the variables).

```
$ cobbler import --path=/net/${NFSSERVER}/${NFSPATH} \ --name=$DISTRONAME \ --available-as=nfs://${NFSSERVER}:${NFSPATH}
```

Beaker/Server needs a little more info than cobbler normally stores about a distro in order to use it. That's why beaker-lab-controller provides a script in `/var/lib/cobbler/triggers/sync/post/osversion.trigger` which needs to be run after you import a new distro. It looks up the distro's full family, update and looks for any yum repos that may be in the distro path. It also adds the cobbler distros into the Beaker server.

```
$ /var/lib/cobbler/triggers/sync/post/osversion.trigger
```

Check that the Distro was added successfully by going to <https://BeakerServer.example.com/bkr/distros>. You'll need to configure the `/etc/beaker/proxy.conf` file with the following settings.

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

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

# Username and password
USERNAME = "host/lab.example.com" # This needs to match the account you created on the
Beaker Scheduler
PASSWORD = "testing" # Again, only if you are not using kerberos does this need to be set.

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

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

2.3. Beaker Client

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 add the following:

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

If instead kerberos authentication is preferred:

```
AUTH_METHOD = "krbv"  
KRB_REALM = "krb_realm"
```

To verify it is working properly:

```
$ bkr list-labcontrollers
```

It should return a list of labcontrollers configured in Beaker.

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 in a the shell prompt, and submit this to the Beaker server. To see all the options that can be passed to the **workflow-simple**, use the following command:

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

A common set of paramaters 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 succesful, 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
```


User Guide

This user guide contains both written guide and the associated images that is intended to give assistance to people using Beaker.

3.1. Introduction

Beaker is the name for an automated testing framework proposed for Fedora. It allows you to:

- Build tasks.
- Run tasks on remote systems.
- Run tasks on multiple systems and system variations (i.e arch, operating system, memory size etc).
- Store and display test results in a central location.
- Manage an inventory of systems for running tests on.

Beaker enables users to create any task they would like (from testing changes to kernel memory management to installing their favourite OS and game server) and run these tasks on any number of machines of any specification located anywhere in the world, and provide a simple interface from where they can review the outcome of these tasks.

3.2. Getting Started

This section gets you started using Beaker. There are three basic steps that a user needs to know before using Beaker.

- **Installation:** See [Chapter 2, Installation](#) for instructions on installation.
- **Process:** comprises of description of procedures, workflows, componenets, architecture, test cases, etc.
- **Checklist:** It's a list of steps in the workflow in order to have accurate test results.

3.2.1. Process

This section has a detailed description of all the components and procedures involved in Beaker Test Environment.

3.2.1.1. Submitting and Reviewing a Job Workflow

To submit a Job you must create Job Workflow. This is an XML file containing the tasks you want to run, as well as special environment variables and other options you want to setup for your Job.



Valid Job Specs

If this is the first time running this Job Workflow make sure that the Distro, System Arch and Tasks are all available to Beaker. To do this See [Section 3.2.1.3.1.1, "System Searching"](#), [Section 3.2.1.3.1.5.1, "Distro Searching"](#) and [Section 3.2.1.3.3.2, "Task Searching"](#) respectively

To submit the Job, use either the beaker-client [Section 2.3, “Beaker Client”](#) or submit the Job via the web app [Section 3.2.1.3.1.6.3.1, “Submitting a New Job”](#)

Once Submitted you can view the progress of the Job by going to the Job search page [Section 3.2.1.3.1.6.2, “Job Searching”](#). Once your Job is Completed, see the Job results page [Section 3.2.1.3.1.6.3.4, “Job Results”](#).

3.2.1.2. 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.

3.2.1.2.1. Provision by System

Go to the System details page (see [Section 5.1.3, “System Details Tabs”](#)) of a System that is free (see [Section 5.1.1, “System Searching”](#)) and click on Take in the Current User field. After successfully taking the System, click the Provision tab of the System details page to provision the System.



Returning a System

After provisioning a System, you can manually return it by going to the above mentioned System details page, clicking on the Return link in the Current User field.

The screenshot shows the Beaker web application interface. At the top, there's a navigation bar with 'File', 'Edit', 'View', 'History', 'Bookmarks', 'Tools', and 'Help'. Below that, the 'Beaker' logo is visible, followed by tabs for 'Systems', 'Devices', 'Distros', 'Scheduler', 'Reports', and 'Activity'. The main content area is titled 'dev-pe1950-01' and shows system details. On the left, there's a sidebar with a list of Fedora distributions. The main area has a 'Provision' tab selected, showing fields for 'KickStart MetaData', 'Kernel Options (Install)', and 'Kernel Options (Post)'. The 'Provision System' button is highlighted in red.

System Name	dev-
Date Created	2010-03-31 13:43:05
Last Checkin	2010-04-07 14:18:04
Lender	Dell
Serial Number	
Condition	Working
Shared	<input type="checkbox"/>
Secret (NDA)	<input type="checkbox"/>
Lab Controller	lab
Type	Machine
Last Modification	2010-04-07 14:18:05
Vendor	Dell
Model	PE1950
Location	BOS, Lab 3, A-0
Owner	
Current User	Return
Loaned To	
Mac Address	

Version - 0.5.25

Details | Arch(s) | Key/Values | Groups | Excluded Families | Power | Notes | Install Options | Provision | Lab Info | History | Tasks

Fedora-10-Beta-ftp-PAE-i386
Fedora-10-Beta-http-PAE-i386
Fedora-10-ftp-i386
Fedora-10-ftp-PAE-i386
Fedora-10-ftp-x86_64
Fedora-10-http-i386
Fedora-10-http-PAE-i386
Fedora-10-http-x86_64
Fedora-10-nfs-i386
Fedora-10-nfs-PAE-i386

KickStart MetaData
Kernel Options (Install) console=ttyS1,57600
Kernel Options (Post)
Reboot System?
[Provision System](#)

Report Bug

Find: Previous Next Highlight all Match case

Provision by System

3.2.1.2.2. Provision by Distro

Go to the Distro search page ([Section 5.2.1, “Distro Searching”](#)) 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 one else is using them and you can take them, 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, however it cannot be changed later.
- **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.

3.2.1.2.3. 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, starting 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** See [Section 3.2.1.2.2, “Provision by Distro”](#)
- **KickStart MetaData** See [Section 3.2.1.2.2, “Provision by Distro”](#)
- **Kernel Options (Install)** See [Section 3.2.1.2.2, “Provision by Distro”](#)
- **Kernel Options (Post)** See [Section 3.2.1.2.2, “Provision by Distro”](#)

Once you are ready you can provision your System with your selected Distro by pressing **Queue Job**.

3.2.1.3. Components

3.2.1.3.1. 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 amongst other things.

3.2.1.3.1.1. 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 Systems listed in Beaker.
 - **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.

Systems - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Systems

Beaker Systems Devices Distro Scheduler Reports Activity

Hello, [User] Logout

version - 0.5.25

Systems

Toggle Search

Search

Table Operation Value

CPU/Cores greater than 4 Remove (x)

System/Arch is *64* Remove (x) Search

System/Type is Machine Remove (x)

Add (+)

Toggle Result Columns

- ☐ CPU/Cores
- ☒ CPU/Family
- ☒ CPU/Type
- ☐ CPU/Model
- ☐ CPU/ModelName
- ☐ CPU/Processors
- ☐ CPU/Sockets
- ☐ CPU/Speed
- ☐ CPU/Stepping
- ☐ CPU/Vendor
- ☐ System/Arch
- ☐ System/Lender
- ☐ System/LoanedTo
- ☐ System/Location
- ☐ System/Memory
- ☐ System/Model
- ☐ System/Name
- ☐ System/Owner
- ☐ System/PowerType
- ☐ System/Status
- ☐ System/Type
- ☐ System/User
- ☐ System/Vendor

Select None Select All Select Default

Show all 1 2 >>>

Name	Status	Vendor	Model	Arch	User	Type
del [redacted] .com	Working	Dell Computer Corporation	PowerEdge 1850	i386, x86_64		Machine
del [redacted] .com	Removed	Dell		i386, x86_64		Machine
del [redacted] .com	Removed					Machine

Find: [input] Previous Next Highlight all Match case

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**

3.2.1.3.1.2. Adding a System

To add a System, go to any System search page, and click on the **Add(+)** link on the bottom left. You must be logged in to do this. After filling in the details, press the **Save Changes** button on the bottom left hand corner.

Version - devel-version

System Name	dell-pe1850-01.beaker.com		
Date Created			
Last Checkin			
Lender	IBM	Last Modification	
Serial Number	2341245	Vendor	IBM
Condition	Working	Model	Z series
Shared	<input type="checkbox"/>	Location	Brisbane
Secret (NDA)	<input type="checkbox"/>	Owner	
Lab Controller	lab-devel-@.com	Current User	
Type	Machine	Loaned To	
		Mac Address	99-A4-00-00-1B

Save Changes

Find: Previous Next Highlight all Match case

Adding a System

3.2.1.3.1.3. System Details Tabs

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 Details
 - **Details**
 - Shows the details of the System's CPU, as well as Devices attached to the System.
 - **Arch**
 - Shows the architects supported by the system.
 - **Key/Values**
 - Shows further hardware details.
 - **Groups**
 - Shows the groups of which this System is a part of.
 - **Excluded Families**
 - Are the list of Distros that this System does not support.
 - **Power**
 - Allows the powering off/on and rebooting of this System. These options are only available if you are the current user of this System, in the admin group or are part of a group that has been given admin rights over the machine. Also the machine must be **Taken**.
 - **Notes**
 - Any info about the system that you want others to see and doesn't fit in anywhere else.
 - **Install 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 [Section 3.2.1.3.1.1, "System Searching"](#) for details on searching.

Activity - Mozilla Firefox

File Edit View History Bookmarks Tools Help

Activity

Beaker Systems Devices Distro Scheduler Reports Activity

Version - 0.5.25

Activity

Toggle Search

Search

Table Operation Value

Action contains k Remove Search

Add (+)

User	Via	Date	Object	Property	Action	Old Value	New Value
root	WebUI	2010-04-20 23:22:11	System: dev	User	Reserved		root
root	Scheduler	2010-04-20 23:10:10	System: dev	User	Returned	root	root
root	Scheduler	2010-04-20 23:10:10	System: del	User	Returned	root	root
root	Scheduler	2010-04-20 22:01:16	System: del	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 22:01:16	System: dev	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 22:00:39	System: dev	User	Reserved		root
root	Scheduler	2010-04-20 22:00:38	System: del	User	Reserved		root
root	Scheduler	2010-04-20 22:00:09	System: dev	User	Returned	root	root
root	Scheduler	2010-04-20 22:00:09	System: del	User	Returned	root	root
root	Scheduler	2010-04-20 21:12:55	System: del	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 21:12:55	System: dev	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 21:12:23	System: dev	User	Reserved		root
root	Scheduler	2010-04-20 21:12:22	System: del	User	Reserved		root
root	Scheduler	2010-04-20 21:11:32	System: dev	User	Returned	root	root
root	Scheduler	2010-04-20 21:11:32	System: del	User	Returned	root	root
root	Scheduler	2010-04-20 20:52:00	System: del	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 20:52:00	System: dev	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 20:51:28	System: dev	User	Reserved		root
root	Scheduler	2010-04-20 20:51:27	System: del	User	Reserved		root
root	Scheduler	2010-04-20 20:51:00	System: dev	User	Returned	root	root
root	Scheduler	2010-04-20 20:51:00	System: del	User	Returned	root	root
root	Scheduler	2010-04-20 20:13:25	System: del	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 20:13:25	System: dev	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 20:12:53	System: dev	User	Reserved		root
root	Scheduler	2010-04-20 20:12:52	System: del	User	Reserved		root
root	Scheduler	2010-04-20 20:11:04	System: dev	User	Returned	root	root
root	Scheduler	2010-04-20 20:11:04	System: del	User	Returned	root	root
root	Scheduler	2010-04-20 19:44:30	System: dev	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 19:44:30	System: del	Distro	Provision		Fedora-12
root	Scheduler	2010-04-20 19:44:11	System: dev	User	Returned	root	root

Find: Previous Next Highlight all Match case

Searching through all System's activities

3.2.1.3.1.4. System Activity

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

Individual System history

To search the history of a specific System, see the **History** tab in Section 5.1.3, [Section 3.2.1.3.1.3, "System Details Tabs"](#)

3.2.1.3.1.5. Distro

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

3.2.1.3.1.5.1. Distro Searching

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

3.2.1.3.1.6. Jobs

The purpose of a Job is to provide an encapsulation of Tasks. It is to provide a single point of submission of these Tasks, and a single point of reviewing the output and results of these Tasks. 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 can not alter its contents, or pause it. You can however cancel it (Section 5.3.3.4, "Job Results"), and alter its 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.

3.2.1.3.1.6.1. Job Workflow

To create a simple Job workflow, see the **bkr workflow-simpl** command in Chapter 2, Beaker client

3.2.1.3.1.6.2. Job Searching

To search for a Job, navigate to **Scheduler>Jobs** at the top of the page. To look up the **Job ID**, enter a number in the search box and press the **Lookup ID button**. Please see Section 5.1.1, "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.

3.2.1.3.1.6.3. Job Submission

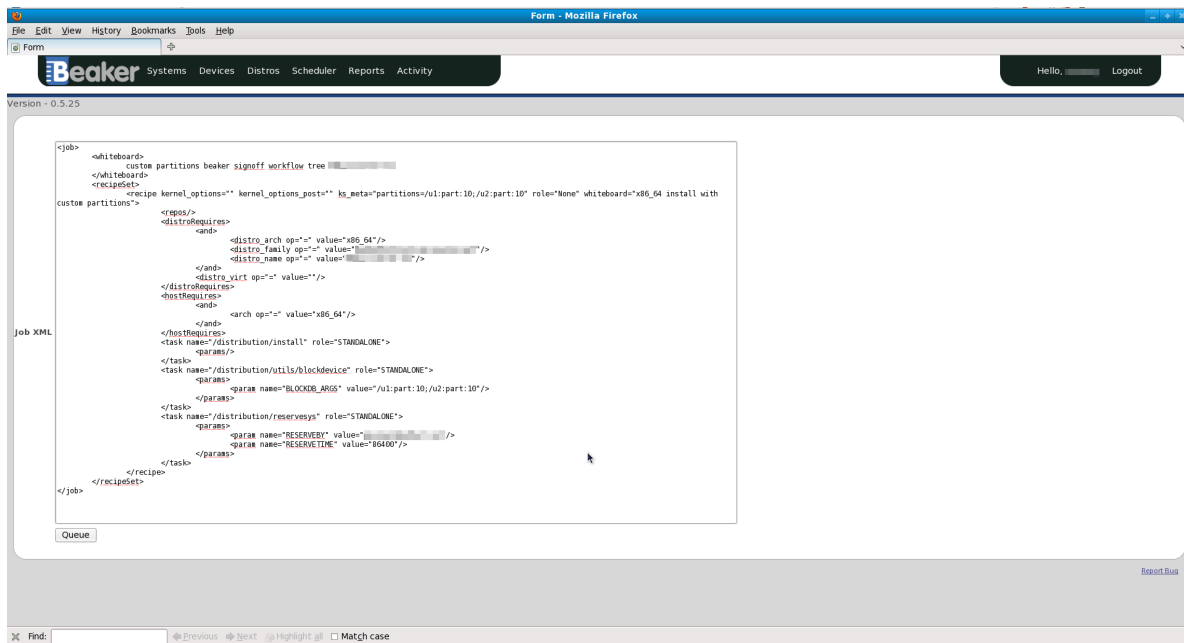
There are two ways of submitting a Job through the web app. They are outlined below.

3.2.1.3.1.6.3.1. 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 queueing it as a Job by pressing the **Queue** button.

3.2.1.3.1.6.3.2. 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. See Section 5.3.2, "Job Searching".



Cloning a Job

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!

3.2.1.3.1.6.3.3. Job workflow details

There are various XML entities in a Job workflow. You may wish to look at what some of these may be by looking at [Section 3.2.1.3.1.6.3.2, "Cloning an existing Job"](#) Each Job has a root node called the **job** element:

```
<job>
</job>
```

A direct child is the **whiteboard**. The content is normally a mnemonic piece of text describing the Job:

```
<job>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
</ob>
```

The next tag in the **recipeSet** tag (which describes a Recipe Set. See Section 5.4, "Recipes" for details). A Job workflow can have one or more **recipeSet**. All Recipes within a Recipe Set are run simultaneously, whereas multiple Recipe Sets are run in no predetermined order. This should help

you decide whether you wish to run tasks in one or many Recipe Set (i.e Multihost tests will require no more than one Recipe Set).

```
<job>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
  <recipeset>
    </recipeset>
</job>
```

Of course a **recipeSet** element needs one or more recipe children. As mentioned above, **Recipes** run simultaneously. The **recipeSet** element can have the following attributes

- **kernel_options**
- **kernel_options_post**
- **ks_meta**
- **role**In a Multihost environment, it could be either **SERVERS**, **CLIENT** or **STANDALONE**. If it is not important, it can be **None**.
- **whiteboard**Text that describes the Recipe

Here is an example:

```
<job>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
  <recipeset>
    <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
      </recipe>
    </recipeset>
</job>
```



Avoid having many Recipes in one Recipe Set

Because Recipes are run simultaneously, not one Recipe will commence until all other sibling Recipes are ready. This involves each Recipe reserving a machine, and waiting until every other Recipe has reserved a machine. This can tie up resources and keep them idle for long amounts of time. Try having many Recipe Sets containing few Recipes, rather than the opposite. Of course this only applies to Recipes that do not need to be run simultaneously (i.e not Multihost Jobs)

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

```
<job>
  <whiteboard>
```

```
        Apache 2.2 test
      </whiteboard>
      <recipeSet>
        <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
          <packages>
            <package name="emacs"/>
            <package name="vim-enhanced"/>
            <package name="unifdef"/>
            <package name="mysql-server"/>
            <package name="MySQL-python"/>
            <package name="python-twll"/>
          </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>
      <whiteboard>
        Apache 2.2 test
      </whiteboard>
      <recipeSet>
        <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
          <packages>
            <package name="emacs"/>
            <package name="vim-enhanced"/>
            <package name="unifdef"/>
            <package name="mysql-server"/>
            <package name="MySQL-python"/>
            <package name="python-twll"/>
          </packages>
          <repos>
            <repo name="myrepo_1" url="http://my-repo.com/tools/
beaker/devel/">
          </repos>
        </recipe>
      </recipeSet>
    </job>
```

To actually determine what distro will be installed, the **distroRequires** element needs to be populated. Within, we can specify such elements 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. The **distro_virt** element will determine whether we install on a virtual machine or not.

```
    <job>
      <whiteboard>
        Apache 2.2 test
      </whiteboard>
      <recipeSet>
        <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
```

```

        <packages>
        <package name="emacs"/>
        <package name="vim-enhanced"/>
        <package name="unifdef"/>
        <package name="mysql-server"/>
        <package name="MySQL-python"/>
        <package name="python-twll"/>
        </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>
        <distro_virt op="=" value=""/>
        </distroRequires>
        </recipe>
    </recipeSet>
</job>

```

hostRequires has similar attributes to **distroRequires**

```

    <job>
    <whiteboard>
    Apache 2.2 test
    </whiteboard>
    <recipeSet>
    <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
    <packages>
    <package name="emacs"/>
    <package name="vim-enhanced"/>
    <package name="unifdef"/>
    <package name="mysql-server"/>
    <package name="MySQL-python"/>
    <package name="python-twll"/>
    </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>
    <distro_virt op="=" value=""/>
    </distroRequires>
    <hostRequires>
    <and>
    <arch op="=" value="x86_64"/>
    </and>
    </hostRequires>
    </recipe>
    </recipeSet>
</job>

```

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**. Details of how to find which Task's are available, see Section 5.5.2, “Task Searching”. Also note that we've added in a **param** as a descendant of **task**. The **value** of this will be assigned to a new environment variable specified by **name**.

```
<job>
  <whiteboard>
    Apache 2.2 test
  </whiteboard>
  <recipeSet>
    <recipe kernel_options="" kernel_options_post=""
ks_meta="" role="None" whiteboard="Lab Controller">
      <packages>
        <package name="emacs"/>
        <package name="vim-enhanced"/>
        <package name="unifdef"/>
        <package name="mysql-server"/>
        <package name="MySQL-python"/>
        <package name="python-twll"/>
      </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>
        <distro_virt op="=" value=""/>
      </distroRequires>
      <task name="/distribution/install" role="STANDALONE">
        <params>
          <param name="My_ENV_VAR" value="foo"/>
        </params>
      </task>
    </recipe>
  </recipeSet>
</job>
```


3.2.1.3.1.6.3.4. 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 (Section 5.3.2, “Job Searching”). 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 Set**. 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**.

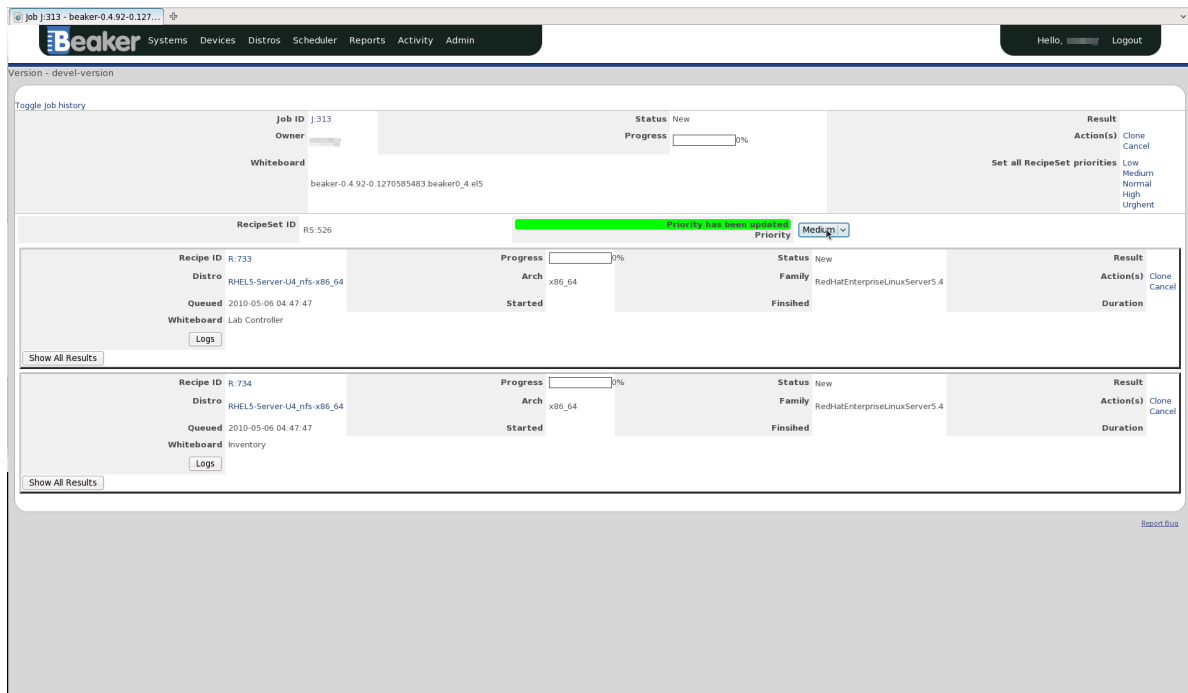
While your Job is still **Queued** it's possible to change the priority. You can change the priority of individual Recipe Sets by changing the value of **Priority**, or you can change all the Job's Recipe Sets at once by clicking an option beside the text **Set all RecipeSet priorities**, which is at 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



The screenshot shows the Beaker web interface. At the top, there's a navigation bar with links: Systems, Devices, Distros, Scheduler, Reports, Activity, Admin. The main content area displays job details for Job ID J-313. It includes a progress bar (0%), status (New), and a dropdown menu for priority (Medium). Below this, there's a table of Recipe Sets. The first row shows Recipe ID R-733, Distro RHEL5-Server-U4_nfs-x86_64, Arch x86_64, Status New, Family RedHatEnterpriseLinuxServer5.4, and Action(s) Clone, Cancel. The second row shows Recipe ID R-734, Distro RHEL5-Server-U4_nfs-x86_64, Arch x86_64, Status New, Family RedHatEnterpriseLinuxServer5.4, and Action(s) Clone, Cancel. Both rows have a 'Show All Results' button and a 'Logs' button.

Changing the priority of a Job's Recipe Set

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**
 - 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 Section 5.3.3.2, “Cloning an existing Job” to learn about Cloning.



Viewing Job results at a glance

If you would be able to look at the Result of all Tasks within a particular Job, try the Matrix Report, See Section 5.6.1, “Matrix Report”.

3.2.1.3.2. 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.

3.2.1.3.2.1. Recipe Searching

The Recipe search is accessed through the **Scheduler** at the top of the page, and clicking on the **Recipe** link.

ID	Whiteboard	Arch	Syst	Distro	Progress	Status	Result	Action
R-902		i386	intel	RHEL6.0-20100421.n.0_nfs-Server-i386	75%	Running	Pass	Cancel
R-901		i386	dev	RHEL6.0-20100421.n.0_nfs-Server-i386	75%	Running	Pass	Cancel
R-900		i386	dev	RHEL6.0-20100421.n.0_nfs-Server-i386	0%	Running	Pass	Cancel
R-899		i386	dev	RHEL6.0-20100421.n.0_nfs-Server-i386	75%	Running	Warn	Cancel

Searching for a Recipe

To look up the **Recipe ID** enter a number into the search box and press the **Lookup ID** button. See Section 5.1.1, “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.

3.2.1.3.2.2. Recipe Actions

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

3.2.1.3.3. Tasks

Tasks are the lowest unit in the Job hierarchy, and running a Task is the primary purpose of running a Job. Their purpose is to run one or more commands, and then collect the results of those commands in a way that other entities can access them. You can run as many or as few Tasks in a Job as you like.

3.2.1.3.3.1. Creating a Task

To create Tasks the **beaker-devel** package will need to be installed. From your terminal, type:

```
$ yum install beaker-devel
```

Now make a new directory from where you will create the test. Then **cd** into the newly created folder and run the following:

```
$ beaker-create-new-test
$ ls -l
Makefile  PURPOSE  runtest.sh
```

Below is a rundown of the files created and how to use them

3.2.1.3.3.1.1. runtest.sh

The core of each Beaker test is a **runtest.sh** shell script. It performs the testing (or delegates the work by invoking another script or executable) and reports the results. Either write the code that performs the test in the **runtest.sh** shell script or have **runtest.sh** execute another program that does the bulk of the work in perhaps another language. Choose a language appropriate to the job (and with which you are familiar): testing of a library could be written in C, parsing of text streams could be done in Perl, and GUI scripting in Python. Languages can be mixed and matched as appropriate within a single test - the **runtest.sh** script can call other code as necessary. Aim for correctness and readability: remember that others may have to debug this code if a test is flawed. Here is the example **runtest.sh**:

```
#!/bin/sh

# Copyright (c) 2006 Red Hat, Inc. All rights reserved. This
# material is made available to anyone wishing to use,
# redistribute it subject to the terms and conditions of the
# Public License v.2.
#
# This program is distributed in the hope that it will be
# useful, but WITHOUT
# ANY WARRANTY; without even the implied warranty of
# FOR A PARTICULAR PURPOSE. See the GNU General Public License
# for more details.
```

```
License                                     #
                                           # You should have received a copy of the GNU General Public
Foundation, Inc.,                          # along with this program; if not, write to the Free Software
                                           # 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,USA.
                                           # Author: Your Name <Your Email>

                                           beaker-run-simple-test $TEST ./do-my-test.sh
```

The `do-my-test.sh` is a rather simple test, seen below, just enough to illustrate the required components of a task.

```
#!/bin/sh
# simple test example

ls -al /root/.ssh
```

The exit code from listing the contents of `/root/.ssh` will be saved to the `$?` system variable and based on this, **beaker-run-simple-test** will report the failure or success of the Task. So the most basic Beaker coding requirements are that a shell script must be sourced (**beaker-environment.sh**) and the `report_result` API must be called (this is called in **beaker-run-simple-text**). The output was sent to **\$OUTPUTFILE** (in **beaker-environment.sh**). The Beaker API provides the **\$OUTPUTFILE** variable so each test can create a log file to record its activities. When run from the command line, **\$OUTPUTFILE** points to a temporary file in `/tmp`. When run in a test lab environment, the logs are stored in a central location. The `report_result` method is always called to inform the Beaker API of the success or failure of the test.

3.2.1.3.3.1.2. Makefile

A standard Beaker Makefile coordinates many aspects of developing and running a Beaker test:

- compiling test executables.
- packaging test files into a single RPM.
- downloading external source files for use in the test.
- collect test files into a known location for execution.
- running tests

A sample Makefile is copied into the local directory when `beaker-create-new-test` tool is invoked (or can be found at `/usr/share/doc/beaker-devel-2.6/Makefile.template`). The Makefile sets up certain targets and defines variables necessary for test execution and reporting. It is best to use this example when writing a new test, copying and modifying it as necessary. So for example, in order to have an executable compiled by the Makefile, the following two lines were changed.

```
BUILT_FILES=do-my-test
FILES=$(METADATA) runtest.sh Makefile PURPOSE do-my-test.c
```


When **make run** is typed, the Makefile will compile **do-my-test.cintodo-my-test** and run `runtest.sh`, which will then execute **do-my-test**. You do not have to enter anything in the **BUILT_FILES** directive if you are not compiling an executable. Also be sure to fill in the following section:

```
# The toplevel namespace within which the test lives.
# FIXME: You will need to change this:
TOPLEVEL_NAMESPACE=

# The name of the package under test:
# FIXME: you will need to change this:
PACKAGE_NAME=

# The path of the test below the package:
# FIXME: you will need to change this:
RELATIVE_PATH=
```

Those three place holders will be used to determine how your Task is named in Beaker, and also how it's mounted on a test System. It will be called **/TOPLEVEL_NAMESPACE/PACKAGE_NAME/RELATIVE_PATH** and it will be mounted the same in the **/mnt/tests** directory on a test System.

3.2.1.3.3.1.3. PURPOSE

The test code directory contains a plain text file called **PURPOSE** which explains what the test addresses along with any other information useful for troubleshooting or understanding it better. The **PURPOSE** file has no minimum or maximum length but should provide useful information. For example:

```
$ cat PURPOSE
This trivial test compiles a .c file, runs the resulting code,
and checks that the output is as expected.

It is intended as a simple example of how to write a test that
compiles source code to a binary and reports a single result using beaker-run-simple-test

It can also be used as a primitive smoketest for the compiler.
```

3.2.1.3.3.1.4. Packaging

Before we can add a new Task to Beaker we need to package it. Firstly, you're able to run the task locally (if it makes sense) to ensure that the task operates as expected. If you would like to run the task locally:

```
$ make run
chmod a+x ./runtest.sh
./runtest.sh
/tmp/simple_test/do-my-test.sh
Running ./do-my-test.sh As root:
total 48
drwx----- 2 root root 4096 Dec 15 08:41 .
drwxr-x--- 24 root root 12288 Jun 2 14:57 ..
-rw----- 1 root root 415 Dec 13 19:12 authorized_keys
-rw-r--r-- 1 root root 8630 May 25 09:42 known_hosts
...finished running ./do-my-test.sh, exit code=0
```

```
/// result: PASS
Log: /tmp/tmp.c24401
```

Once you're happy it works as expected, you can run the following:



Tagging

If you wish to, and your Task is revisioned with git or CVS, you can increase the version of the package with the following command:

```
$ make tag
```

3.2.1.3.3.2. Task Searching

To search for a Task, go to **Scheduler>Task** Library at the top of the page. The default search is on the **Name** property, with the **contains** operator. See Section 5.1.1, "System Searching" for search details.

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

It's also possible to search on the history of the running of Tasks. This is made possible by the **Executed Tasks** search, which is accessed by clicking on a task.

3.2.1.3.3.3. Adding a New Task

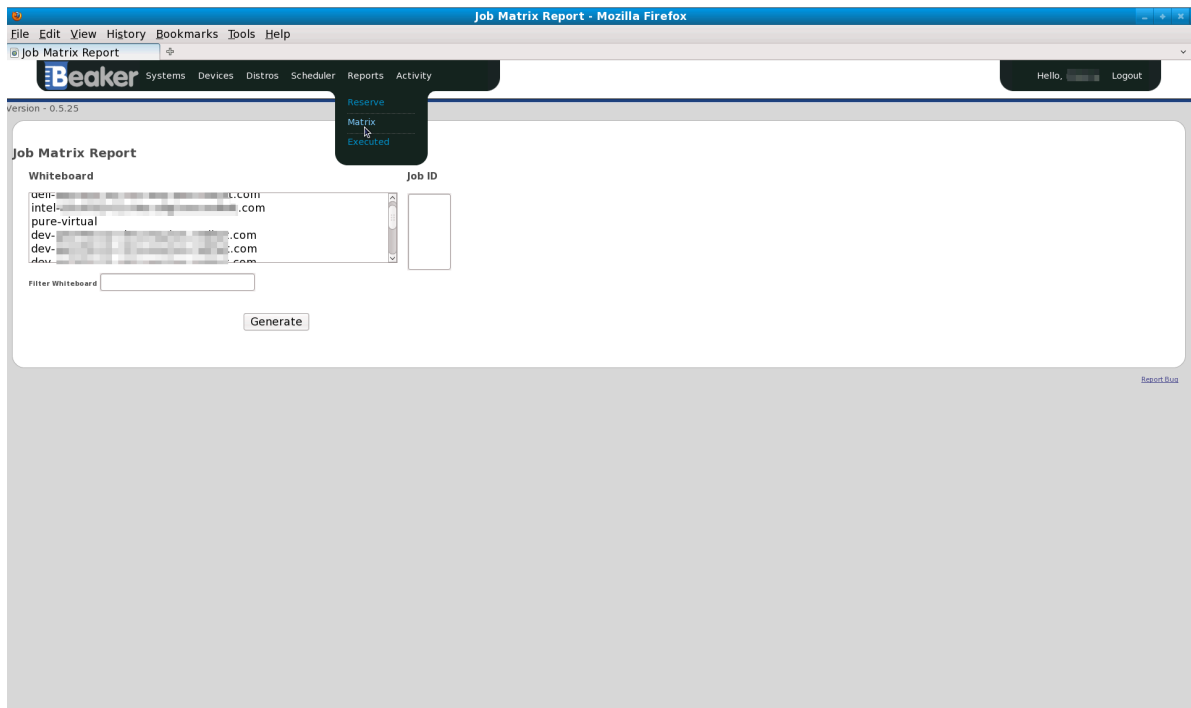
To add a Task which has already been packaged, click **Scheduler>New Task**. You will need to click on **Browse** to locate your Task, and then add it with the **Submit Data** button. See also Chapter 2, Beaker client for adding a task via the beaker client.

3.2.1.3.4. Reports

Beaker offers a few different reports. They can be accessed from the Reports menu at the top of the page. The Reserve report will give reservation details of Systems that are currently in use. The other report offered is the Matrix report.

3.2.1.3.4.1. 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 accesable from the top menu.



Generating a Matrix report from the Job's Whiteboard

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. 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.



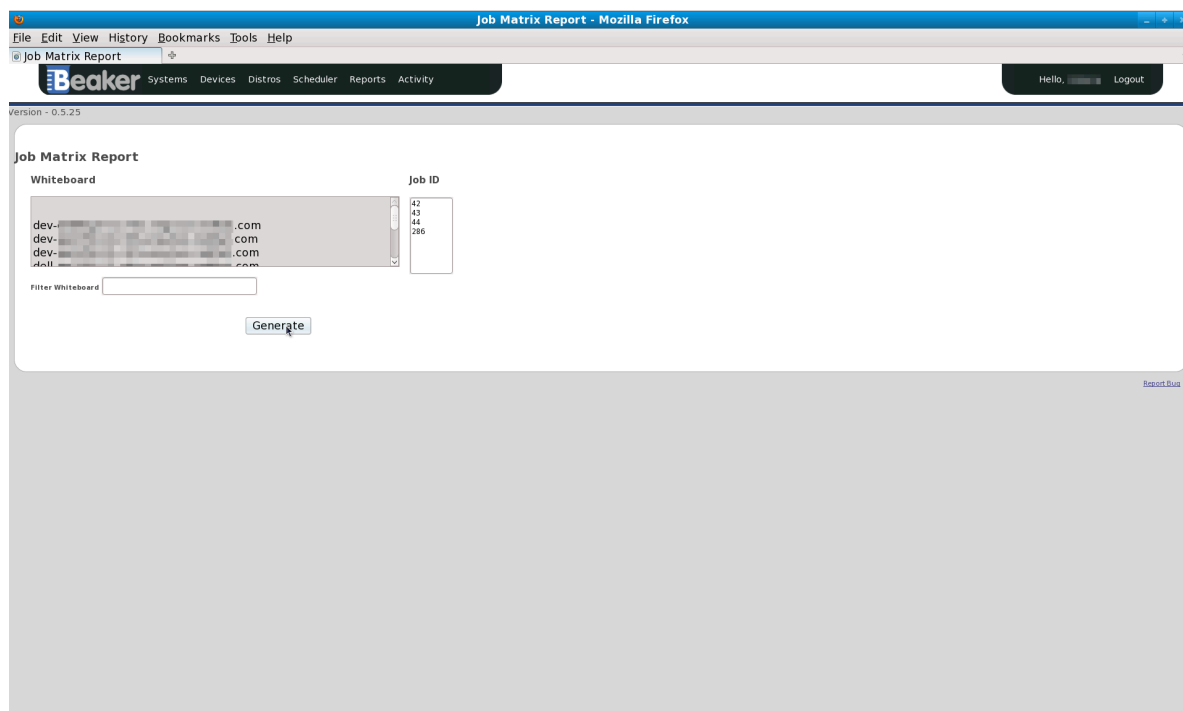
Filtering Whiteboards

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



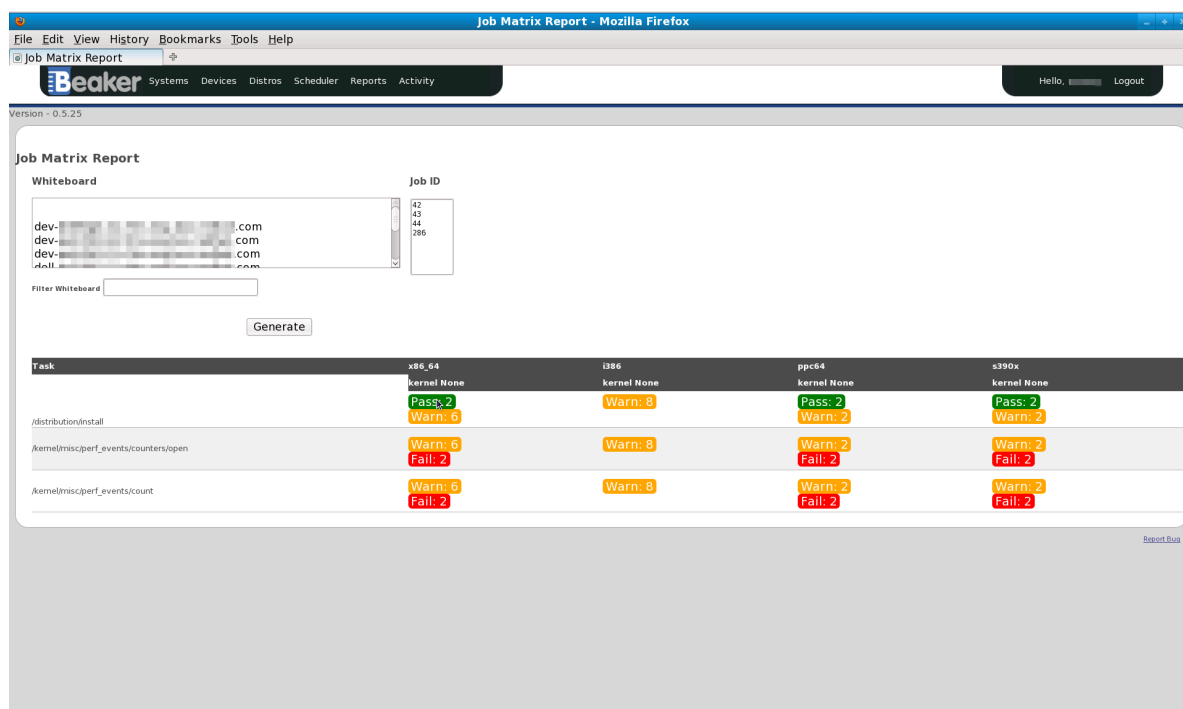
Displaying reports of any combination of Jobs

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



Generating a Matrix report from the Job ID

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 Section 5.5.2, "Task Searching".



Viewing the result of one or more Jobs via the Matrix report

3.2.1.3.5. Groups

To have one or more **Users** grouped together, Beaker uses **Groups**. Systems can belong to one or more Groups.

3.2.1.3.5.1. Adding a Group

Groups can only be created/edited by a User in the **Admin Group**. To add a new Group go to **Admin->Groups** and 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.

3.2.1.4. Test Architecture Considerations

If you want your test to be smart, that intelligence must be in the test; the Beaker API can help. A test running in an automated environment does not have intelligence, hunches, or the ability to notice unusual activity. This intelligence must be programmed into the test. Naturally the return on investment for time required to add this intelligence should be considered, however the more intelligence a test has to handle false failures and false passes, the more valuable the automation is to the entity running it. Contrasted with manual testing where tests are run on a local workstation and suspicious results can be investigated easily, many organizations find that well written tests which can be trusted save time that can be used for any number of other activities.

Questions to consider

- What is needed for a test run to return PASS?
- What is needed for a test run to return FAIL?
- How will PASS and FAIL conditions be determined pragmatically?
- If it is not possible for a test to ever FAIL, does it make sense to automate it?

Things to Keep in Mind

- Assume that nothing works:
 - The test could be running in an unstable test environment.
 - The package under test might be broken.
 - An apparently-unrelated component might cause your test to fail in an unexpected way.
 - The system might not be configured in the manner in which you expect.
 - The test may be buggy, reporting false positives or false negatives.
- Identifying potential problem sections in a test can save someone, possibly you, hours of debugging time.

Writing Good Test Code

- Check everything: all exit statuses, return values from function calls, etc. Unfortunately there are plenty of programs which return success codes even when a failure occurs.
- Capture all debug output that might indicate an error; it may give clues as to what is going wrong when a test fails.

- Comment your tests; good comments should describe the intent of what you are doing, along with caveats being followed, rather than simply parroting the code back as pseudo code.
- In most (ideally all) situations a test should report true PASS and FAIL results, but test code is still code, and will invariably contain bugs.
- Program defensively so that errors in test code report false FAIL results rather than false PASSes. For example, initialize a result variable to FAIL and only set it to PASS if no errors are detected.
- Do not initialize a variable to PASS which fails only on a specific error —what if you missed another error? What if the shell function you called failed to execute?
- It is easier to investigate and fix a failed test than a test that always passes (which it should not be).

3.2.1.5. Reporting Results

The philosophy of Beaker is that the engineers operating the system will want to quickly survey large numbers of tests, and thus the report should be as simple and clear as possible. "PASS" indicates that everything completed as expected. "FAIL" indicates that something unexpected occurred.

In general, a test will perform some setup (perhaps compiling code or configuring services), attempt to perform some actions, and then report on how well those actions were carried out. Some of these actions are your responsibility to capture or generate in your script:

- a PASS or FAIL and optionally a value indicating a test-specific metric, such as a performance figure.
- a debug log of information —invaluable when troubleshooting an unexpected test result. A test can have a single log file and report it into the root node of your results tree, or gather multiple logs, reporting each within the appropriate child node.

Other components of the result can be provided automatically by the framework when in a test lab environment:

- the content of the kernel ring buffer (from dmesg). Each report clears the ring buffer, so that if your test reports multiple results, each will contain any messages logged by the kernel since the last report was made.
- a list of all packages installed on the machine under test (at the time immediately before testing began), including name, version/release, and architecture.
- a separate report of the packages listed in the RunFor of the metadata including name, version/release, and architecture (since these versions are most pertinent to the test run).
- if a kernel panic occurs on the machine under test, this is detected for you from the console log output, and will cause an Abort Panic result in place of a PASS or FAIL for that test.

In addition, the Beaker framework provides a hierarchical namespace of results, and each test is responsible for a subtree of this namespace. Many simple tests will only return one result (the node they own), but a complex test can return an entire subtree of results as desired. The location in the namespace is determined by the value of variables defined in the Makefile. These variables will be discussed in the Packaging section.

A test may be testing a number of related things with a common setup (e.g. a setup phase of a server package onto localhost, followed by a collection of tests as a client). Some of these things may not

work across every version/architecture combination. This will produce a list of "subresults", each of which could be classified as one of:

- expected success: will lead to a PASS if nothing else fails
- expected failure: should be as a PASS (as you were expecting it).
- unexpected success: can be treated as a PASS (since it's a success), or a FAIL (since you were not expecting it).
- unexpected failure: should always be a FAIL

Given that there may be more than one result, the question arises as to how to determine if the whole test passes or fails. One way to verify regression tests is to write a script that compares a set of outputs to an expected "gold" set of outputs which grants PASS or FAIL based on the comparison.

It is possible to write a script that silently handles unexpected successes, but it is equally valid for a script to report a FAIL on an unexpected success, since this warrants further investigation (and possible updating of the script).

To complicate matters further, expected success/failure may vary between versions of the package under test, and architecture of the test machine.

If the test is checking multiple bugs, some of which are known to work, and some of which are due to be fixed in various successive (Fedora) updates, ensure that the test checks everything that ought to work, reporting PASS and FAIL accordingly. If the whole test is reporting a single result, it will typically report this by ensuring that all expected things work; as bugs are fixed, more and more of the test is expected to work and can cause an overall FAIL.

If it is reporting the test using a hierarchy of results, the test can have similar logic for the root node, and can avoid reporting a result for a subtree node for a known failure until the bug is fixed in the underlying packages, and avoid affecting the overall result until the bug(s) is fixed.

As a general Beaker rule of thumb, a FAIL anywhere within the result subtree of the test will lead to the result for the overall test being a FAIL.

3.2.1.5.1. Logging Tips

Indicate failure-causing conditions in the log clearly, with "FAIL" in upper case to make it easier to grep for.

Good log messages should contain three things: # what it is that you are attempting to do (e.g. checking to see what ls reports for the permission bits that are set on file foo) # what it is that you expect to happen (e.g. an expectation of seeing "-rw-r--r--") # what the actual result was an example of a test log showing success might look like:

```
Checking ls output: "foo" ought to have permissions "-rw-r--r--"
Success: "foo" has permissions: "-rw-r--r--"
```

An example of a failure might look like:

```
Checking ls output: "foo" ought to have permissions "-rw-r--r--"
FAIL: ls exit status 2
```

For multihost tests, time stamp all your logs, so you can interleave them.

Use of tee is also helpful to ensure that the output at the terminal as you debug a test is comparable to that logged from OUTPUTFILE in the lab environment.

Past experiences has shown problems where people confuse overwriting versus appending when writing out each line of a log file. Use tee -a \$OUTPUT rather than tee > \$OUTPUT or tee >> \$OUTPUT.

Include a final message in the log, stating that this is the last line, and (for a single-result test) whether the result is a success or failure; for example:

```
echo "----- Test complete: result=$RESULT -----" | tee -a
$OUTPUTFILE
```

Finish your runtest.sh: (after the report_result) to indicate that the final line was reached; for example:

```
echo "***** End of runtest.sh *****"
```

3.2.1.5.2. Passing Parameters to Tests

When you need a test to perform different steps in some specific situations there is an option available through Single package workflow command line interface called --test-params which allows you to pass the supplied parameter to runtest.sh where you can access it by TEST_PARAM_NAME=value.

For example you can launch the single workflow with a commandline like this:

```
single_package.py ... -t /some/test/name --test-params="PAR1=val1" --test-params="PAR2=val2"
```

And then make use of the passed parameter inside the runtest.sh script:

```
if [[ TEST_PARAM_PAR1 == 1 ]] ; then do something; fi
```

3.2.1.6. Unassimilated or Unfinished content

3.2.1.6.1. Using the startup_test function

The startup_test function can be used to provide a primitive smoketest of a program, by setting a shell variable named result. You will need to use report_result if you use it. The syntax is:

```
startup_test program [arg1] [arg2] [arg3]
```

The function takes the name of a program, along with up to three arguments. It fakes an X server for the test by ensuring that Xvfb is running (and setting DISPLAY accordingly), then enables core-

dumping, and runs the program with the arguments provided, piping standard output and error into OUTPUTFILE (overwriting, not appending).

The function then checks various things:

- any Gtk-CRITICAL warnings found in the resulting OUTPUTFILE cause result to be WARN.
- that the program can be found in the PATH, using the which command; if it is not found it causes result to be FAIL, appending the problem to OUTPUTFILE
- for binaries, it uses ldd to detect missing libraries; if any are missing it causes result to be FAIL, appending the problems to OUTPUTFILE
- if any coredumps are detected it causes result to be FAIL

Finally, it kills the fake X server. You then need to report the result.

```
#!/bin/sh

# source the test script helpers
. /usr/bin/beaker-environment.sh

# ---- do the actual testing ----
result=PASS 1
startup_test /usr/bin/evolution
report_result $TEST $result 2
```

Normally it's a bad idea to start with a PASS and try to detect a FAIL, since an unexpected error that prevents further setting of the value will lead to a false PASS rather than a false FAIL. Unfortunately in this case the startup_test function requires it.

```
report_result $TEST $result
```

We report the result, using the special result shell variable set by startup_test

3.2.1.7. Writing and Running Multihosts Tests

All of the examples so far have run on a single host. Beaker has support for tests that run on multiple hosts, e.g. for testing the interactions of a client and a server.

When a multihost test is run in the lab, a machine will be allocated to each role in the test. Each machine has its own recipe. Multihost testing requires the concept of a recipe set: all of the per-machine recipes within a particular multihost test that go together to form the test as a whole.

Each machine under test will need to synchronize to ensure that they start the test together, and may need to synchronize at various stages within the test. Beaker has three notional roles: 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 whilst the servers get ready. Once all servers are ready, the clients perform whatever testing they need, using the services provided by the server machines, and eventually report results back to Red Hat Test System Whilst this is happening the server tests block; the services running on these machines are

carrying out work for the clients in parallel. Once all clients have finished testing, the server tests finish, and report their results.

Each participant in a test will be reporting results within the same job, and so must 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. Talk to Red Hat Quality Engineering if you need to do this.

All of the participants in a multihost test share a single **runtest.sh**, which must perform 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 these variables, and perform the necessary role accordingly. These variables are shared by all instances of the **runtest.sh** within a recipe set:

- **CLIENTS** contains a space-separated list of hostnames of clients within this recipe set.
- **SERVERS** contains a space-separated list of hostnames of servers within this recipe set.
- **DRIVER** is the hostname of the driver of this recipe set, if any.

The variable **HOSTNAME** can be used by **runtest.sh** to determine its identity. It is set by **beaker-environment.sh**, and will be unique for each host within a recipe set.

Your test can thus decide whether it is a client, server or driver by investigating these variables: see the example below.

When you are developing your test outside the lab environment, only **HOSTNAME** is set for you (when sourcing the **beaker-environment.sh** script). Typically you will copy your 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.

A multihost test needs to be marked as such in the *Type: Multihost*.

3.2.1.7.1. Synchronization Commands

Synchronization of machines within a multihost test is performed using per-host state strings managed on the Red Hat Test System server. Each machine's starting state is the empty string.

```
beaker-sync-set -s state
```

The **beaker-sync-set** command sets the state of this machine within the test to the given value.

```
beaker-sync-block -s state [hostnames...]
```

The **beaker-sync-block** command blocks further execution of this instance of the script until all of the listed hosts are in the given state.

Unfortunately, there is currently no good way to run these commands in the standalone helper environment.

3.2.1.7.1.1. Example of a runtest.sh for a multihost test

```
#!/bin/sh
# Source the common test script helpers
. /usr/bin/beaker_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."
    beaker_sync_block -s "DONE" ${SERVERS}

    user="finger1"
    for i in ${SERVERS}
    do
        echo "-- finger user \"${user}\" from server
        \"${i}\".\"

        ./finger_client "${i}" "${user}"
        # It returns non-zero for failure.
        if [ $? -ne 0 ]; then
            beaker_sync_set -s "DONE"
            report_result "${TEST}" "FAIL" 0
            exit 1
        fi
    done

    echo "-- client finishes."
    beaker_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
        beaker_sync_set -s "DONE"
        report_result "${TEST}" "FAIL" 0
        exit 1
    fi
    useradd finger1
    echo "-- server finishes."
    beaker_sync_set -s "DONE"
    beaker_sync_block -s "DONE" ${CLIENTS}
    result="PASS"
}

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

```
exit 0
```

3.2.1.7.1.2. Tuning up multihost tests

Multihost tests can be easily tuned up outside Beaker using following code snippet based on \$JOBID variable (which is set when running in Beaker environment). Just log in to two machines (let's say: client.redhat.com and server.redhat.com) and add following lines at the beginning of your runtest.sh script.

```
# decide if we're running on Beaker or in developer mode
if test -z $JOBID ; then
    echo "Variable JOBID not set, assuming developer mode"
    CLIENTS="client.redhat.com"
    SERVERS="server.redhat.com"
else
    echo "Variable JOBID set, we're running on Beaker"
fi
echo "Clients: $CLIENTS"
echo "Servers: $SERVERS"
```

Then you just run the script on both client and server. When scripts reach one of the synchronization commands (beaker-sync-set or beaker-sync-block) you will be asked for supplying actual state of the client/server by keyboard (usually just confirm readiness by hitting Enter). That's it! :-)

3.2.1.8. Beaker Makefile

This guide provides in depth information for the required and optional Makefile variables in an Beaker test. A sample Makefile is copied into the local directory when beaker-create-new-test tool is invoked (or can be found at `/usr/share/doc/beaker-devel-x.y/Makefile.template`).

3.2.1.8.1. PACKAGE NAME

```
# The name of the package under test:
PACKAGE_NAME=gcc
```

The package under test is the common command or executable being tested. This must be the name of an installable RPM in the distribution. If the focus of your test is a third party application, set PACKAGE_NAME equal to the primary package used by the third party application. For example, if you are writing a Beaker test to test a web application based on CGI, set PACKAGE_NAME to perl.

3.2.1.8.2. TOPLEVEL NAMESPACE

```
# The toplevel namespace within which the test lives.
TOPLEVEL_NAMESPACE=CoreOS
```

The **Makefile** contains three hierarchies resembling file systems, each with their own collections of paths. In order to ensure consistency between test creators and tests, the provided **Makefile** should be used to manage these hierarchies. Thus, the scheme in the **Makefile** template does all the work.

For clarity, it is worth noting the hierarchies at this time:

- *installation* tests are built as packages for clean deployment on test machines. More than one test can be run on a given machine, so there is a hierarchy below `/mnt/tests` which keeps the files of the individual tests separate from each other. This is set in each test's Makefile.
- *result namespace* a hierarchical namespace for results. For example, tests relating to the kernel report their results somewhere within the `/kernel` subtree, and tests relating to the NFS file system (as a part of the kernel) report their results inside `/kernel/filesystems/nfs/`. Each test "owns" a subtree of the namespace, specified by the `Name:` field of the metadata. Many tests report only a single result, but it is possible for a test to write out a complex hierarchy of results below its subtree.

The following top level reporting namespaces are predefined and should be used to ensure consistent reporting of test results. These are the only valid accepted namespaces.

- **distribution** contains tests that involve the distribution as a whole, or a large number of packages, for example `/distribution/standards/posixtestsuite`
- **kernel** contains tests results relating to the kernel, for example `/kernel/xen/xm/dmesg`
 - The **kernel** namespace is unique in that it is also the name of a package. In this case it is usually best to define the **TOPLEVEL_NAMESPACE** like this:

```
# The toplevel namespace definition for kernel tests
TOPLEVEL_NAMESPACE=$(PACKAGE_NAME)
```

- **Desktop** contains tests results relating to desktop packages, for example `/desktop/evolution/first-time-wizard-password-settings`, which is a specific test relating to evolution
- **Tools** contains tests results relating to the tool chain, for example `/tools/gcc/testsuite/3.4`
- **CoreOS** all test results relating to user-space packages not covered by any of the above namespaces
- **Examples** example tests that illustrate usage and functionality and are not actively maintained. This is a good place to experiment when you are getting hang of Beaker or to place simple examples to help others.

3.2.1.8.3. RELATIVE PATH

```
# The path of the test below the package:
RELATIVE_PATH=example-compilation
```

An implementation of Beaker should run the test from the directory containing the `runtest.sh`, as listed in the **RELATIVE_PATH** file of the **Makefile**. If the test needs to move around, store this somewhere with `DIR=`pwd`` or use `pushd` and `popd`.

3.2.1.8.4. TESTVERSION

```
# Version of the Test. Used with make tag.
export TESTVERSION=1.0
```

This is used when building a package of a test, and provides the "version" component of the RPM name-version-release triplet.

- The value must be valid as an RPM version string.
- It may consist of numbers, letters, and the dot symbol.
- It may not include a dash symbol this is used by RPM to delimit the version string within the name-version-release triplet.

When writing a test from scratch, use 0.1 and increment gradually until the test has reached a level of robustness to merit a "1.0" release. When wrapping a test from an upstream location, use the upstream version string here, as closely as possible given the restrictions on valid characters. The version should be incremented each time a change is made to the **Makefile** or test files and a RPM is created from these files to be publicly consumed in a test review or submission to a lab scheduler.

3.2.1.8.5. TEST

```
# The compiled namespace of the test.
export TEST=$(TOPLEVEL_NAMESPACE)/$(PACKAGE_NAME)/$(RELATIVE_PATH)
```

This variable defines the path to a test. This path should also be the same in source control.

3.2.1.8.6. BUILT_FILES

```
BUILT_FILES=hello-world
```

List the files that need to be compiled to be used in the test.

3.2.1.8.7. FILES=\$(METADATA)

```
FILES=$(METADATA) runtest.sh Makefile PURPOSE hello-world.c \
    verify-hello-world-output.sh
```

List all of the files needed to run the test to insure that there are no packaging errors when the test is built binaries should be pulled in via BUILT_FILES.

3.2.1.8.8. Targets

Each test must supply a run target which allows an implementation of the framework to invoke **make run**. It is usually best to have this as the first target defined in the **Makefile** so that a simple invocation of **make** will use it as the default, and run the test. Note how the **build** target is set up as a dependency of run to ensure that this happens if necessary.

Additional targets and variables supplied by the include for **/usr/share/beaker/lib/beaker-make.include**. This file is supplied with **beaker-devel** as seen below.

```
[root@dhcp83-5 example-compilation]# cat /usr/share/beaker/lib/
beaker-make.include
```

```

copyrighted material 1
General
but WITHOUT AN Y
or FITNESS FOR A
details.
License
02110-1301, USA.

# Copyright (c) 2006 Red Hat, Inc. All rights reserved. This
# is made available to anyone wishing to use, modify, copy, or
# redistribute it subject to the terms and conditions of the GNU
# Public License v.2.
#
# This program is distributed in the hope that it will be useful,
# WARRANTY; without even the implied warranty of MERCHANTABILITY
# PARTICULAR PURPOSE. See the GNU General Public License for more
#
# You should have received a copy of the GNU General Public
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
#
# Author: Greg Nichols <gnichols@redhat.com>
#
# beaker-make.include
#
# default rules and settings for beaker makefiles
#

# Common Variables.
TEST_DIR=/mnt/tests$(TEST)
INSTALL_DIR=$(DEST)$(TEST_DIR)
METADATA=testinfo.desc

# tag: mark the test source as a release

tag:
    beaker-mk-tag-release

release: tag

# prep: prepare the test(s) for packaging

install: $(FILES) runtest.sh testinfo.desc
    mkdir -p $(INSTALL_DIR)
    cp -a $(FILES) Makefile $(INSTALL_DIR)
    install -m 0755 runtest.sh $(INSTALL_DIR)

# package: build the test package

package:
    beaker-mk-build-package

# submit: submit the test package to Beaker

submit:
    beaker-mk-build-package -s $(TESTSERVER)

#####
# example makefile
#
# include ~/devel/beaker/greg/beaker_nb/make.include
#
# FILES=prog1.c prog2.c

```

```
#
# ARENA=$(DEST)/mnt/tests/glibc/double-free-exploit
#
# install:
#     mkdir -p $(ARENA)
#     cp -a runtest.sh $(FILES) $(ARENA)
#
# run: tests
#     runtest.sh
#
# tests: prog2 prog2
```

The **tag** target is used to tag a package in anticipation of submitting it to a test lab.

The **submit** target is used to submit a package to a test lab and requires the **TESTSERVER** variable to be defined. It builds an **RPM** of the test (if necessary) and uploads the test package to a test lab controller where it can be used to schedule tests.

3.2.1.8.9. \$(METADATA)

Following is an example of the **METADATA** section needed to execute a basic test. Following subsections will comment briefly on the values that must be set manually (not set by variables) and optional values to enhance test reporting and execution.

```
$(METADATA): Makefile
    @touch $(METADATA)
    @echo "Owner:      David Malcolm"
<dmalcolm@redhat.com>" > $(METADATA)
    @echo "Name:      $(TEST)" >> $(METADATA)
    @echo "Path:      $(TEST_DIR)" >> $(METADATA)
    @echo "License:    GPLv2" >> $(METADATA)
    @echo "TestVersion: $(TESTVERSION)" >>
$(METADATA)
    @echo "Description: Ensure that compiling a
simple .c file works as expected" >> $(METADATA)
    @echo "TestTime:    1m" >> $(METADATA)
    @echo "RunFor:      $(PACKAGE_NAME)" >>
$(METADATA) # add any other packages for which your test ought to run here
    @echo "Requires:   $(PACKAGE_NAME)" >>
$(METADATA) # add any other requirements for the script to run here
```

3.2.1.8.10. Owner

Owner: (optional) is the person responsible for this test. Initially for Beaker, this will be whoever committed the test to Subversion. A naming policy may have to be introduced as the project develops. Acceptable values are a subset of the set of valid email addresses, requiring the form: **Owner: human readable name <username@domain>**.

3.2.1.8.11. 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 names (analogous to a path). This field specifies the namespace where the test will appear in the framework, and serves as a unique ID for the test. Tests should be grouped logically by the package under test. This name should be consistent

with the name used in source control too. Since some implementations will want to use the file system to store results, make sure to only use characters that are usable within a file system path.

3.2.1.8.12. Description

Description (required) must contain exactly one string.

For example:

```

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

3.2.1.8.13. TestTime

Every **Makefile** must contain exactly one **TestTime** value. It represent the upper limit of time that the **runtest.sh** script should execute before being terminated. That is, the API should automatically fail 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. It is recommended to provide a value in minutes, for readability.

The time should be the absolute longest a test is expected to take on the slowest platform supported, plus a 10% margin of error. It is usually meaningless to have a test time of less than a minute, since some implementations of the API may be attempting to communicate with a busy server such as writing back to an NFS share or performing an XML-RPC call.

For example:

```

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

3.2.1.8.14. Requires

Requires one or more. This field indicates the packages that are required to be installed on the test machine for the test to work. The package being tested is automatically included via the **PACKAGE_NAME** variable. 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:

```
$(METADATA)
@echo "Requires:      gdb" >> $(METADATA)
@echo "Requires:      @legacy-software-development" >>
@echo "Requires:      @kde-software-development" >> $(METADATA)
@echo "Requires:      -pdksh" >> $(METADATA)
```

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, the names of packages are used rather than Provides: dependencies. If one of the dependencies changes name between releases, one of these approaches below may be helpful:

- for major changes, split the test, so that each release is a separate test in a sub-directory, with the common files built from a shared directory in the **Makefile**.
- if only a dependency has changed name, specify the union of the names of dependencies in the Requires: field; an implementation should silently ignore unsolvable dependencies.
- it may be possible to work around the differences by logic in the section of the **Makefile** that generates the **testinfo.desc** file.

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.

3.2.1.8.15. BeakerRequires

BeakerRequires one or more. This field 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 be a space-separated list of Beaker Test RPM name without the version or .rpm. Each Test must exist on the Beaker Scheduler.

For example:

```
@echo "BeakerRequires:      rh-tests-distribution-hts-common" >> $(METADATA)
```

3.2.1.8.16. RunFor

RunFor allows for the specification of the packages which are relevant for the test. This field is the hook to be used for locating tests by package. For example, when running all tests relating to a particular package[1], an implementation should use this field. Similarly, when looking for results on a particular package, this is the field that should be used to locate the relevant test runs.

When testing a specific package, that package must be listed in this field. If the test might reasonably be affected by changes to another package, the other package should be listed here. If a package changes name in the various releases of the distribution, all its 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.

3.2.1.8.17. Releases

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 test 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.

- Valid **Releases** are:
 - RedHatEnterpriseLinux3
 - RedHatEnterpriseLinux4
 - RedHatEnterpriseLinuxServer5
 - RedHatEnterpriseLinuxClient5
 - FedoraCore6
 - Fedora7
 - Fedora8
- Releases can be used in two ways:
 - specifying releases you **want** run your test for : For example, if you want to run your test on RHEL3 and RHEL4 only, add "Releases: RedHatEnterpriseLinux3 RedHatEnterpriseLinux4" to your Makefile METADATA variable, i.e.:

```

...
$(METADATA)      @echo "Requires:      openldap-servers" >>
                  @echo "Releases:      RedHatEnterpriseLinux3
RedHatEnterpriseLinux4" >> $(METADATA)
                  @echo "Priority:      Normal" >> $(METADATA)
...

```

- specifying releases you **don't want** run your test for (using "-" sign before given releases): For example, if you don't want to run your test on RHEL3, but the other releases are valid for your test, add "Releases: -RedHatEnterpriseLinux3" to your Makefile METADATA variable, i.e.:

```

...
$(METADATA)      @echo "Requires:      openldap-servers" >>
                  @echo "Releases:      -RedHatEnterpriseLinux3" >>
$(METADATA)
                  @echo "Priority:      Normal" >> $(METADATA)
...

```

3.2.1.9. Virtualization Workflow

Virtualization workflow is designed to take advantage all Beaker offers to be used for virtualization testing. The audience of this tutorial is expected to have basic familiarity with Beaker.

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 virtualization

test framework is `guestrecipe`. Each virtual machine is defined in its own `guestrecipe` and `guestrecipes` are a part of the host's (`dom0`) 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>
...
(dom0 test recipe)
...
<guestrecipe guestname="guest1">
...
(guest1 test recipe)
...
</guestrecipe>
<guestrecipe guestname="guest2">
...
(guest2 test recipe)
...
</guestrecipe>
</recipe>
```

xml syntax workflows example can be found here: [Section 3.2.1.3.1.6.1, "Job Workflow"](#)

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

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.

3.2.1.9.1. 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.

3.2.1.9.2. Making More Sense of LVM and PARTITION based Guest Installations

Installing a file-based guest is the simplest of all. A specific file can be specified in guestarg with -file or -f arguments , or can just be omitted. However, for lvm and partition based guests, the virtinstall test will have to know where to install the guests exactly, because there is no way for it to know what the partition or volume name might be.

Obviously, if lvm or partition based guests are desired a custom partitioning will have to be done. Beaker allows the testers to submit a custom partition/lvm for their tests. The syntax for this below:

<partition> <type> type </type> <name> name </name> <size> size in GB </size> <fs> filesystem to format </fs> </partition>

- **type:** Can be either lvm or part. This is required.
- **name:** For partitions, this will be mount point, such as /mnt/guest1 , for lvm, it'll be the name of the volume, such as myguestlvm . In the case of lvm, it'll be named /TestVolumeGroup??/myguestlvm . This is required.
- **size:** Size of the blockdevice in Gigabytes, this is required.
- **fs:** filesystem you'd like to be formatted for this partition . This is optional, if omitted, ext3 will be used. * -file or -f guestargs, whatever the name is given inside the partition block should be passed on.

3.2.1.9.3. Other Uses of PARTITIONS and LVM Volumes

In virtualization testing, block devices can be used for other purposes than installing a guest on them. It's possible to have block devices attached to or detached from a guest. If you'd like to do this operation in your tests, you can either pass the information about the partitions you have created as environment variable to your test, or you can use the blockdevice utility, which is another test that lives in /distribution/utls/blockdevice . This testcase just creates a text file containing information about the block devices and manages them. Its commands are:

getdevice

- Usage: getdevice <lvm|partition> <minimum size in GB>
- Description: returns a free device that has enough space

- Returns: string with device name and return code 0 on success, error string and return code 1 on failure

freedevice

- Usage: freedevice <devicename>
- Descriptions: Marks the device can free so it can be reused.
- Returns: return code zero on success or error string and return code 1 on failure.

When running blockdevice test-util, you need to provide testing an environmental variable named BLOCKDB_ARGS, telling the test what partitions/lvms you have intended to be used in test script. The format of **\$BLOCKDB_ARGS** is name:type:size:fs;name:type:size:fs; for each block device. For example suppose you have these partitions in your job:

```
<partition>
  <type>part</type>
  <name>/mnt/block1</name>
  <size>1</size> <!-- 1 gig -->
</partition>
<partition>
  <type>lvm</type>
  <name>mylvm</name>
  <size>5</size>
</partition>
<partition>
  <type>part</type>
  <name>/mnt/block4ext4</name>
  <size>1</size> <!-- 1 gig -->
  <fs>ext4dev</fs>
</partition>
<partition>
  <type>lvm</type>
  <name>mylvm4ext4</name>
  <size>5</size>
  <fs>ext4dev</fs>
</partition>
```

then BLOCKDB_ARGS would be: /mnt/block1:part:1;mylvm:lvm:5;/mnt/block4ext4:part:1:ext4dev;mylvm4ext4:lvm:5:ext4dev

If you utilize blockdevices test, you should ensure that:

- blockdevice is called before any of your scripts
- freedevice must be called after the testing on the space is done.

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

```

<partition> is the xml element for the partitioning. You can have multiple partition elements in a recipe. 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/utills/blockdevice . This test unmounts the specified partitions/volumes and lets users manage custom partitions thru its own scripts.
- **getdevice**
 - Usage: getdevice <lvm|partition> <minimum size in GB>>
 - Description: returns a free device that has enough space
 - Returns: string with device name and return code 0 on success, error string
 - and return code 1 on failure
- **freedevice**
 - Usage: freedevice <devicename>
 - Descriptions: Marks the device can free so it can be reused.
 - Returns: return code zero on success or error string and return code 1 on failure.

When running blockdevice test-util, you need to provide testing an environmental variable named BLOCKDB_ARGS, telling the test what partitions/lvms you have intended to be used in test script. The format of \$BLOCKDB_ARGS is name,type,size;name,type,size; for each block device. For example suppose you have these partitions in your job:

```

<partition>
  <type>part</type>
  <name>/mnt/block1</name>
  <size>1</size> <!-- 1 gig ->
</partition>
<partition>

```

```
<type>lvm</type>
<name>mylvm</name>
<size>5</size>
</partition>
<partition>
  <type>part</type>
  <name>/mnt/block4ext4</name>
  <size>1</size> <!-- 1 gig -->
  <fs>ext4dev</fs>
</partition>
<partition>
  <type>lvm</type>
  <name>mylvm4ext4</name>
  <size>5</size>
  <fs>ext4dev</fs>
</partition>
</screen>
<para>then BLOCKDB_ARGS would be:
</para>
<screen>/mnt/block1:part:1;mylvm:lvm:5;/mnt/
block4ext4:part:1:ext4dev;mylvm4ext4:lvm:5:ext4dev
```

as in:

```
<test role='STANDALONE' name='/distribution/utils/blockdevice'>
  <params>
    <param name='BLOCKDB_ARGS' value='/mnt/
block1:part:1;mylvm:lvm:5;/mnt/block4ext4:part:1:ext4dev;mylvm4ext4:lvm:5:ext4dev' />
  </params>
</test>
```

If you utilize blockdevices test, you should ensure that:

- blockdevice is called before any of your scripts
- freedevice must be called after the testing on the space is done.

3.2.1.9.4.1. 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 `kickPart()` call to the recipe object. The string you have to pass is exactly same format as the `BLOCKDB_ARGS` argument mentioned above. An example can be :

```
part_str = "/mnt/block1:part:1;mylvm:lvm:5;/mnt/
block4ext4:part:1:ext4dev;mylvm4ext4:lvm:5:ext4dev"
rec = Recipe(scheduler=beaker_sched)
rec.kickPart(part_str)
```

3.2.1.9.4.2. Installing Package with Workflows

Workflow scripts are packaged in the `beaker-redhat` package.

- [Installing package with workflows](#)¹
- [Executing a Workflow](#)²

3.2.1.9.5. 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.

3.2.1.10. Multihost Testing

Running multihost tests on Beaker is done thru having different test roles in multihost tests amongst multiple recipes inside a recipeset. In it's simplest form, a job with multihost testing can look like:

```
<job>
  <RecipeSet>
    <recipe>
      <test role='STANDALONE' name='/distribution/install' />
      <test role='SERVERS' name='/my/multihost/test' />
    </recipe>
    <recipe>
      <test role='STANDALONE' name='/distribution/install' />
      <test role='CLIENTS' name='/my/multihost/test' />
    </recipe>
  </RecipeSet>
</job>
```



Note

For brevity some necessary parts are left out in the above job description



Tips

you can use [*multi_workflow.py*](#)³ to generate a XML template

Submitting the job above will export environmental variables SERVERS and CLIENTS set to their respective hostnames. This allows a tester to write tests for each machines. So 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"
SERVERS="test1.beaker.bos.redhat.com"
CLIENTS="test2.beaker.bos.redhat.com"
DEVMODE=true
fi

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

```

if $(echo $SERVERS | grep -q $HOSTNAME); then
    TEST="$TEST/Server"
    Server
fi

if $(echo $CLIENTS | grep -q $HOSTNAME); then
    TEST="$TEST/Client"
    Client
fi

```

Let's dissect the code. First of, we have `Server()` and `Client()` functions which will be executed on `SERVERS` and `CLIENTS` machines respectively. Then we have an `if` block to determine if this is running as an beaker test, or if it's being run on the test developer's machine(s) to test it out. The last couple `if` blocks determine what code to run on this particular machine. As mentioned before, `SERVERS` and `CLIENTS` environmental variables will be set to their respective machines' names and exported on both machines.

Obviously, there will have to be some sort of coordination and synchronization between the machines and the execution of the test code on both sides. Beaker offers two utilities for this purpose, `beaker-sync-set` and `beaker-sync-block`. `beaker-sync-set` is used to setting a state on a machine. `beaker-sync-block` is used to block the execution of the code until a certain state on certain machine(s) are reached. Those familiar with parallel programming can think of this as a barrier operation. The detailed usage information about both of these utilities is below:

- **beaker-sync-set:** It does set the state of the current machine. State can be anything. Syntax: `beaker-sync-set -s STATE`
- **beaker-sync-block:** It blocks the code and doesn't return until a desired STATE is set on desired machine(s). You can actually look for a certain state on multiple machines.. Syntax: `beaker-sync-block -s STATE [-s STATE1 -s STATE2] machine1 machine2 ...`

There are a couple of important points to pay attention to. First of, the multihost testing must be on the same chronological order on all machines. For example, the below will fail:

```

<recipe>
  <test role='STANDALONE' name='/distribution/install' />
  <test role='STANDALONE' name='/my/test/number1' />
  <test role='SERVERS'      name='/my/multihost/test' />
</recipe>
<recipe>
  <test role='STANDALONE' name='/distribution/install' />
  <test role='CLIENTS'    name='/my/multihost/test' />
</recipe>

```

This will fail, because the multihost test is the 3rd test on the server side and it's the 2nd test on the client side.. To fix this, you can pad in dummy testcases on the side that has fewer testcases. There is a dummy test that lives in `/distribution/utls/dummy` for this purpose. So, the above can be fixed as:

```

<recipe>
  <test role='STANDALONE' name='/distribution/install' />
  <test role='STANDALONE' name='/my/test/number1' />
  <test role='SERVERS'      name='/my/multihost/test' />
</recipe>

```

```
<recipe>
  <test role='STANDALONE' name='/distribution/install' />
  <test role='STANDALONE' name='/distribution/utls/dummy' />
  <test role='CLIENTS'      name='/my/multihost/test' />
</recipe>
```



Tips

*multi_workflow.py*⁴ automatically fix un-balanced recipes for you.

One shortcoming of the beaker-sync-block utility is that it blocks forever, so if there are multiple things being done in your test between the hosts, your test will timeout without possibly a lot of code being executed. There is a utility, `blockwrapper.exp` which can be used to put a limit on how many second it should block. The script lives in `/CoreOS/common test`, so be sure to add that test before your multihost tests in your recipes. The usage is exactly same as that of `beaker-sync-block` with the addition of a timeout value at the end, i.e.:

```
blockwrapper.exp -s STATE machine N
```

where N is the timeout value in seconds. If the desired state in the desired machine(s) haven't been set in N seconds, then the script will exit with a non-zero return code. In case of success it'll exit with code 0 .

3.2.1.10.1. A detailed example of multihost testing

For a detailed example of multihost testing, look at migration testing in `$TOP/virt/xen/migrate` . There, a machine is running as an origin machine with the role of "MIGRATE_FROM", which does export in `/var/lib/xen` directory as an NFS share, sets up passwordless-ssh between machines, copies over guest domain config files and then migrates the domains.

3.2.1.11. Workflow XML

A workflow is a model to represent real work for further assessment. More abstractly, a workflow is a pattern of activity enabled by a systematic organization information flows, into a work process that can be documented and learned. Workflows are designed to achieve processing intents of some sort, such as information processing.

3.2.1.11.1. XML Syntax

You can now specify Beaker jobs using an xml file that will be workflow agnostic. This allows users a more consistent, maintainable ways of storing and submitting jobs. You can now 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. Here is a barebone XML file:

```
<job>
  <workflow>Reserve ia64</workflow>
  <submitter>tester@redhat.com</submitter>
  <whiteboard>Reserving ia64 machine</whiteboard>
  <recipeSet>
```

```

<recipe testrepo='development'>
  <distroRequires>ARCH = ia64</distroRequires>
  <distroRequires>FAMILY = RedHatEnterpriseLinuxServer5</
distroRequires>
  <distroRequires>NAME = RHEL5-Server-U2-RC-1</distroRequires>
  <test role='STANDALONE' name='/distribution/install' />
  <test role='STANDALONE' name='/distribution/reservesys'>
    <params>
      <param name='RESERVEBY' value='gozen@redhat.com' />
      <param name='RESERVETIME' value='86400' />
    </params>
  </test>
</recipe>
</recipeSet>
</job>

```

All xml tags above are pretty self explanatory. One of the biggest advantages of using xml for your jobs is not having to use different workflows for different types of tests. There is only one workflow, `submit_job.py` namely, that's being used. It does not have all the functionality/binding for all the workflows out there, but we are working on it to add more workflow functionalities into our xml binding and `submit_job.py` script. Currently as of Nov 2008, you should be able to use `submit_job.py` for any workflows that was used previously but not all cases have been tested yet and we are working on this .

The easiest way to get started on the xml definition would be seeing if your workflow has a way to provide xml definition for the job. For example, `single_package` workflow has `-xml` argument which prints out an xml definition of the job.

Below is a list of XML elements for Beaker:

- `<job>` : It's the root element of any Beaker job definition. Has no attributes.
- `<workflow>` : The name of the workflow. Has no attributes.
- `<submitter>` : Name, email or other identifying string of the job submitter. Has no attributes.
- `<whiteboard>` : A whiteboard string of anything descriptive about the job. Has no attributes.
- `<recipeSet>` : Set of recipes, has multiple recipe child nodes. Has no attributes.
- `<recipe>` : Test recipe itself. Has attributes of `testrepo`, `whiteboard`, `bootargs` .
- `<installPackage>` : Package names to be installed. These will be appended to kickstart's `%packages` section.
- `<hostRequires>` : Set host properties here.
- `<distroProperties>` : Set properties of the distribution you'd like to install here.
- `<kickstart>` : Pass in your custom kickstart with this element.
- `<accesskey>` : Specify the accesskey for certain systems here, if you need to.
- `<partition>` : Root element of partition definition
- `<name>` : name of the partition. If you are intending the partition to be used as partition , then just give the mountpoint, such as `/mnt/myblockdevice` . If you'd rather have the spaces used for LVM , then give a logical volume name, for example, `mylogvol` .

- `<type>` : This is either `part`, for partition , or `lvm` , for logical volume.
- `<size>` : Size of the partition in Gigabytes.
- `<test>` : Specify the tests to run. Has attributes `testrole` and `name` . Can have the `params` child elements.
- `<params>` : Root element for parameters to be passed to the test.
- `<param>` : It has `name` and `value` attributes with for setting name and value of the environmental variables respectively.
- `<guestrecipe>` : This is a pseudo-recipe for virtual machine guests. Can have almost everything `<recipe>` can have. You can specify `distroProperties`, `installPackage`, etc. here. The attributes of this element are vitally important. First attribute is, `guestname`, which is the name you'd like to give to the guest. You can omit `guestname`, in which case Beaker will assign the hostname of the guest as the name of the guest. The second attribute is `guestargs` , which is identical to arguments that are given to `virtinstall` program, except for `-lvm` or `-part` argument which indicates if the guest is to be installed on blockdevice or lvm volume respectively.

For advanced xml examples, visit :<https://engineering.redhat.com/trac/beaker/wiki/WorkflowXmlExamples>⁵

3.2.2. Checklist Discussed

3.2.2.1. Quality of code

Check for the following:

- **Commenting**: Test code is commented and complex routines sufficiently documented.
- **PURPOSE file**: Test code directory contains a plain text file called `PURPOSE` which explains what the test addresses along with any other information useful for troubleshooting or understanding it better.
- **Language-Review**: Optional, but preferred: review by someone with language-of-implementation knowledge.
- **Functional-Review**: Optional, but preferred: functionality peer-reviewed (i.e. by someone else) with knowledge of the given domain.

3.2.2.2. Quality of Logs

Check the following attributes to ensure the quality of logs:

- **Detail of logging**
 - Test logs should be verbose logging activity for both successful and unsuccessful operations. At a minimum these conditions should be recorded:
 - Name of Test (or subtest; something unique)
 - Expected Result
 - Actual Result

⁵ <https://engineering.redhat.com/trac/beaker/wiki/WorkflowXmlExamples>

- Whether items 2 and 3 constitute a PASS or a FAIL.
- This should help with questions such as:
 - How many tests ran?
 - What went wrong on FAILED cases?
 - How many PASSES/FAILs were there?
- And, associating the Name+Result with prior runs:
 - How well are we doing?

3.2.2.3. Correctness

Correctness has following parameters:

- **True PASS and true FAIL results**
 - The test runs and generates true PASS and true FAIL results as appropriate. It is permissible for a test to FAIL even if the expected result is PASS if the software under test has a known defect that has been reported. The applicable bug number should be referenced in the error message so that it is easy to research the failure.
- **Watch for bogus success values**
 - The test verifies PASS and FAIL results (versus returning the success or failure from a particular shell command... many shell commands return success because they successfully ran, not that they returned expected data. This usually requires user verification)
- **Security review**
 - A cursory review of the code should be performed to make sure it does not contain obviously malicious or suspicious routines which appear more focused on damaging or casing the testing infrastructure versus performing a valid test.

3.2.2.4. Packaging

Check the following attributes to ensure the correctness of Packaging:

- **Makefile**
 - **make package** works correctly, generating an **RPM** with the expected payload. The **RPM** should successfully install correctly without any errors or dependency problems.
 - **make clean** should clean up all generated files that will not be stored in source control
 - All unneeded comments and unused variable should be removed from the **Makefile**. The Makefile template contains lots of **FIXME** comments indicating what to put where. These comments should be removed from the final Makefile
 - Metadata section of **Makefile** should have these fields filled properly:
 - Releases (only few tests can correctly run on everything from RHEL-2.1 to F8)
 - RunFor (some tests stresses a lot of RHEL components, so they could be all here)
 - Bug (lot of tests tests specific bug number, it is not enough to have it in test name)
 - **Permissions**: File permissions should be set appropriately on built packages and verified by running **rpm -qp1v** [package name]. For example:

File permissions should be set appropriately on built packages and verified by running `rpm -qplv [package name]`. For example:

- **runtest.sh** should be executable by all users
- any other executables should be executable by all users
- **PURPOSE** and generated **testinfo.desc** should be 644
- **Correct namespace** For Correct namespace, double check the following:
 - Confirm that the test is included in the correct namespace and has followed the proper naming conventions. Refer to the [TOPLEVEL_NAMESPACE] to make sure that the underlying package being tested is reporting results in the correct namespace.
 - The Makefile variables and testnames should also correspond to the correct path in source control. For example:

```
[grover@dhcp83-5 smoke-high-load]# pwd
/home/grover/rhts/tests/bind/smoke-high-load
```

Here are the applicable variables from the Makefile:

```
# The toplevel namespace within which the test lives.
TOPLEVEL_NAMESPACE=CoreOS

# The name of the package under test:
PACKAGE_NAME=bind

# The path of the test below the package:
RELATIVE_PATH=smoke-high-load
```