# openQA Documentation

openQA Team

# Table of Contents

# openQA starter guide

# Introduction

openQA is an automated test tool that makes it possible to test the whole installation process of an operating system. It uses virtual machines to reproduce the process, check the output (both serial console and screen) in every step and send the necessary keystrokes and commands to proceed to the next. openQA can check whether the system can be installed, whether it works properly in 'live' mode, whether applications work or whether the system responds as expected to different installation options and commands.

Even more importantly, openQA can run several combinations of tests for every revision of the operating system, reporting the errors detected for each combination of hardware configuration, installation options and variant of the operating system.

openQA is free software released under the GPLv2 license. The source code and documentation are hosted in the os-autoinst organization on GitHub.

This document describes the general operation and usage of openQA. The main goal is to provide a general overview of the tool, with all the information needed to become a happy user. More advanced topics like installation, administration or development of new tests are covered by further documents available in the official repository.

# Architecture

Although the project as a whole is referred to as openQA, there are in fact several components that are hosted in separate repositories as shown in the following figure.

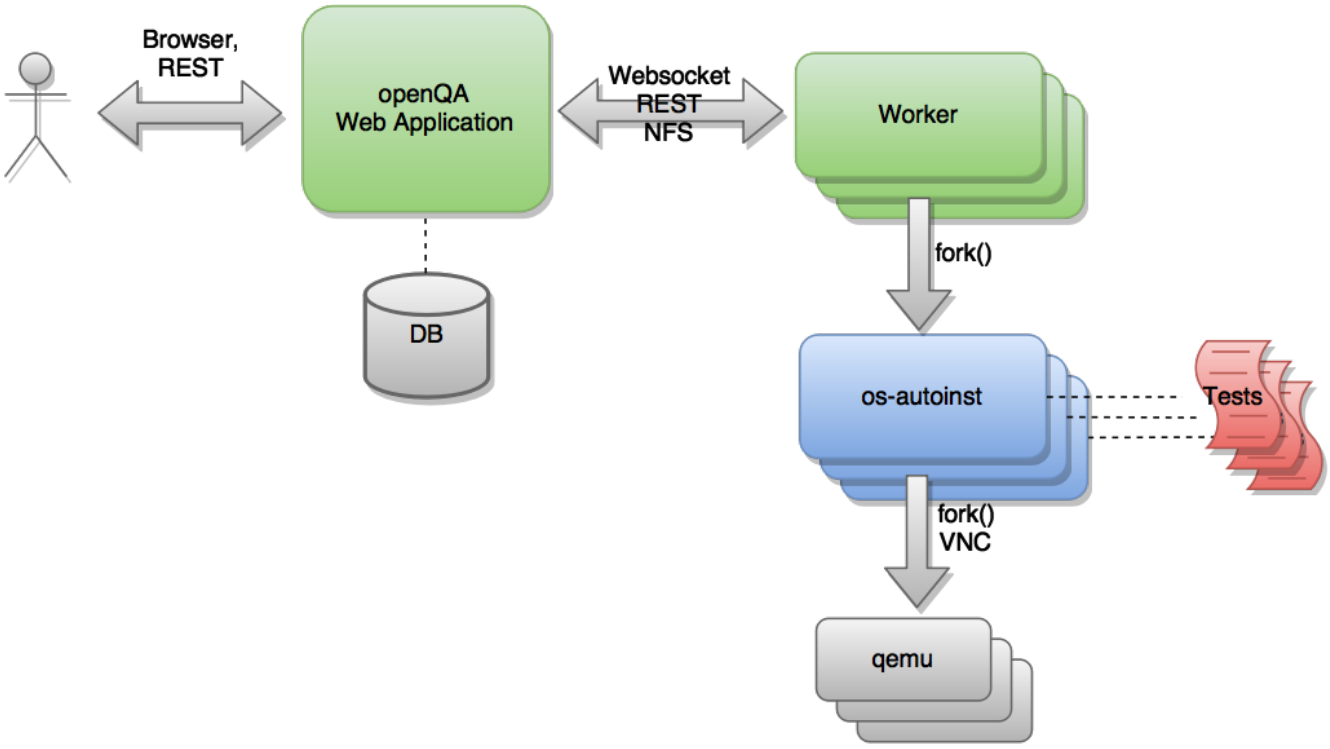



*Figure 1. openQA architecture*

The heart of the test engine is a standalone application called 'os-autoinst' (blue). In each execution, this application creates a virtual machine and uses it to run a set of test scripts (red). 'os-autoinst' generates a video, screenshots and a JSON file with detailed results.

'openQA' (green) on the other hand provides a web based user interface and infrastructure to run 'os-autoinst' in a distributed way. The web interface also provides a JSON based REST-like API for external scripting and for use by the worker program. Workers fetch data and input files from openQA for os-autoinst to run the tests. A host system can run several workers. The openQA web application takes care of distributing test jobs among workers. Web application and workers don't have to run on the same machine but can be connected via network instead.

# Basic concepts

## Glossary

*The following terms are used within the context of openQA*

*test modules*

an individual test case in a single perl module file, e.g. "sshxterm". If not further specified a test module is denoted with its "short name" equivalent to the filename including the test definition. The "full name" is composed of the *test group* (TBC), which itself is formed by the top-folder of the test module file, and the short name, e.g. "x11-sshxterm" (for x11/sshxterm.pm)

*test suite*

a collection of *test modules*, e.g. "textmode". All *test modules* within one *test suite* are run serially

*job*

one run of individual test cases in a row denoted by a unique number for one instance of openQA, e.g. one installation with subsequent testing of applications within gnome

*test run*

equivalent to *job*

*test result*

the result of one job, e.g. "passed" with the details of each individual *test module*

*test step*

the execution of one *test module* within a *job*

*distri*

a test distribution but also sometimes referring to a *product* (CAUTION: ambiguous, historically a "GNU/Linux distribution"), composed of multiple *test modules* in a folder structure that compose *test suites*, e.g. "opensuse" (test distribution, short for "os-autoinst-distri-opensuse")

*product*

the main "system under test" (SUT), e.g. "openSUSE"

*job group*

equivalent to *product*, used in context of the webUI

*version*

one version of a *product*, don't confuse with *builds*, e.g. "Tumbleweed"

*flavor*

a specific variant of a *product* to distinguish differing variants, e.g. "DVD"

*arch*

an architecture variant of a *product*, e.g. "x86_64"

*machine*

additional variant of machine, e.g. used for "64bit", "uefi", etc.

*scenario*

A composition of <distri>-<version>-<flavor>-<arch>-<test_suite>@<machine>, e.g. "openSUSE-Tumbleweed-DVD-x86_64-gnome@64bit", nicknamed *koala*

*build*

Different versions of a product as tested, can be considered a "sub-version" of *version*, e.g. "Build1234"; **CAUTION:** ambiguity: either with the prefix "Build" included or not

# Jobs

One of the most important features of openQA is that it can be used to test several combinations of actions and configurations. For every one of those combinations, the system creates a virtual machine, performs certain steps and returns an overall result. Every one of those executions is called a 'job'. Every job is labeled with a numeric identifier and has several associated 'settings' that will drive its behavior.

A job goes through several states:

- **scheduled** Initial state for recently created jobs. Queued for future execution.
- **running** In progress.
- **cancelled** The job was explicitly cancelled by the user or was replaced by a clone (see below).
- **waiting** The job is in 'interactive mode' (see below) and waiting for input.
- **done** Execution finished.

Jobs in state 'done' have typically gone through a whole sequence of steps (called 'testmodules') each one with its own result. But in addition to those partial results, a finished job also provides an overall result from the following list.

- **none** For jobs that have not reached the 'done' state.
- **passed** No critical check failed during the process. It doesn’t necessarily mean that all testmodules were successful or that no single assertion failed.
- **failed** At least one assertion considered to be critical was not satisfied at some point.
- **softfailed** At least one non-critical assertion was not satisfied at some point (eg. a softfailure has been recorded explicitly via record_soft_failure) or workaround needles are in place.
- **incomplete** The job is no longer running but no result was provided. Either it was cancelled while running or it crashed.

Sometimes, the reason of a failure is not an error in the tested operating system itself, but an outdated test or a problem in the execution of the job for some external reason. In those situations, it makes sense to re-run a given job from the beginning once the problem is fixed or the tests have been updated. This is done by means of 'cloning'. Every job can be superseded by a clone which is scheduled to run with exactly the same settings as the original job. If the original job is still not in

'done' state, it's cancelled immediately. From that point in time, the clone becomes the current version and the original job is considered outdated (and can be filtered in the listing) but its information and results (if any) are kept for future reference.

# Needles

One of the main mechanisms for openQA to know the state of the virtual machine is checking the presence of some elements in the machine's 'screen'. This is performed using fuzzy image matching between the screen and the so called 'needles'. A needle specifies both the elements to search for and a list of tags used to decide which needles should be used at any moment.

A needle consists of a full screenshot in PNG format and a json file with the same name (e.g. foo.png and foo.json) containing the associated data, like which areas inside the full screenshot are relevant or the mentioned list of tags.

```
{
   "area" : [
      {
         "xpos" : INTEGER,
         "ypos" : INTEGER,
         "width" : INTEGER,
         "height" : INTEGER,
         "type" : ( "match" | "ocr" | "exclude" ),
         "match" : INTEGER, // 0-100. similarity percentage
      },
      ...
   ],
   "tags" : [
      STRING, ...
   ]
}
```

# Interactive mode

There are several points in time during the execution of a job at which openQA tries to match the screen with the available needles, reacting to the result of that check. If the job is running in interactive mode it will stop the execution at that point, freezing the virtual machine and waiting for user input before proceeding. At that moment, the user can modify the existing needles or can create a new one using as a starting point either the current screen of the virtual machine or one of the existing needles. Once the needles are adjusted, the user can command the job to reload the list of needles and continue with the execution.

- **enable interactive mode** Get into waiting for input automatically (ie. waitforneedle) in case it can not find the matched needle and timeout.
- **stop waiting for needle** Stop the waitforneedle call immediately without timeout.
- **continue waiting for needle** Continue testing but will get into waitforneedle in case it can not find the matched needle and timeout.

- **reload needles and retry** Retries after 5 seconds and reloads needles. It helps if a new needle is created before retry.
- **open needle editor** Opens needle editor so the user can create a new needle or modify the existing ones.

The interactive mode is especially useful when creating needles for a new operating system or when the look & feel have changed and several needles need to be adjusted accordingly.

# Access management

Some actions in openQA require special privileges. openQA provides authentication through openID. By default, openQA is configured to use the openSUSE openID provider, but it can very easily be configured to use any other valid provider. Every time a new user logs into an instance, a new user profile is created. That profile only contains the openID identity and two flags used for access control:

- **operator** Means that the user is able to manage jobs, performing actions like creating new jobs, cancelling them, etc.
- **admin** Means that the user is able to manage users (granting or revoking operator and admin rights) as well as job templates and other related information (see the the corresponding section).

Many of the operations in an openQA instance are not performed through the web interface but using the REST-like API. The most obvious examples are the workers and the scripts that fetch new versions of the operating system and schedule the corresponding tests. Those clients must be authorized by an operator using an API key with an associated shared secret.

For that purpose, users with the operator flag have access in the web interface to a page that allows them to manage as many API keys as they may need. For every key, a secret is automatically generated. The user can then configure the workers or any other client application to use whatever pair of API key and secret owned by him. Any client to the REST-like API using one of those API keys will be considered to be acting on behalf of the associated user. So the API key not only has to be correct and valid (not expired), it also has to belong to a user with operator rights.

For more insights about authentication, authorization and the technical details of the openQA security model, refer to the detailed blog post about the subject by the openQA development team.

# Job groups

A job can belong to a job group. Those job groups are displayed on the index page and in the Job Groups menu on the navigation bar. From there the job group overview pages can be accessed. Besides the test results the job group overview pages provide a description about the job group and allow commenting.

Job groups have properties. These properties are mostly cleanup related. The configuration can be done in the operators menu for job groups.

It is also possible to put job groups into categories. The nested groups will then inherit properties

from the category. The categories are meant to combine job groups with common builds so test results for the same build can be shown together on the index page.

# Cleanup

| | |
|---|---|
| **IMPORTANT** | openQA automatically deletes data that it considers "old" based on different settings. For example job data is deleted from old jobs by the gru task. |

The following cleanup settings can be done on job-group-level:

*size limit*
: Limits size of assets

*keep logs for*
: Specifies how long logs of a non-important job are retained after it finished

*keep important logs for*
: How long logs of an important job are retained after it finished

*keep results for*
: specifies How long results of a non-important job are retained after it finished

*keep important results for*
: How long results of an important job are retained after it finished

The defaults for those values are defined in lib/OpenQA/Schema/JobGroupDefaults.pm.

**NOTE** Deletion of job results includes deletion of logs and will cause the job to be completely removed from the database.

**NOTE** Jobs which do not belong to a job group are currently not affected by the mentioned cleanup properties.

# Using the client script

Just as the worker uses an API key+secret every user of the client script must do the same. The same API key+secret as previously created can be used or a new one created over the webUI.

The personal configuration should be stored in a file `~/.config/openqa/client.conf` in the same format as previously described for the client.conf, i.e. sections for each machine, e.g. `localhost`.

# Using job templates to automate jobs creation

## The problem

When testing an operating system, especially when doing continuous testing, there is always a certain combination of jobs, each one with its own settings, that needs to be run for every revision. Those combinations can be different for different 'flavors' of the same revision, like running a different set of jobs for each architecture or for the Full and the Lite versions. This combinational problem can go one step further if openQA is being used for different kinds of tests, like running some simple pre-integration tests for some snapshots combined with more comprehensive post-integration tests for release candidates.

This section describes how an instance of openQA can be configured using the options in the admin area to automatically create all the required jobs for each revision of your operating system that needs to be tested. If you are starting from scratch, you should probably go through the following order:

1. Define machines in 'Machines' menu
2. Define medium types (products) you have in 'Medium Types' menu
3. Specify various collections of tests you want to run in the 'Test suites' menu
4. Go to the template matrix in 'Job templates' menu and decide what combinations do make sense and need to be tested

Machines, mediums and test suites can all set various configuration variables. Job templates define how the test suites, mediums and machines should be combined in various ways to produce individual 'jobs'. All the variables from the test suite, medium and machine for the 'job' are combined and made available to the actual test code run by the 'job', along with variables specified as part of the job creation request. Certain variables also influence openQA's and/or os-autoinst's own behavior in terms of how it configures the environment for the job. Variables that influence os-autoinst's behavior are documented in the file doc/backend_vars.asciidoc in the os-autoinst repository.

In openQA we can parametrize a test to describe for what product it will run and for what kind of machines it will be executed. For example, a test like KDE can be run for any product that has KDE installed, and can be tested in x86-64 and i586 machines. If we write this as a triples, we can create a list like this to characterize KDE tests:

```
(Product,             Test Suite, Machine)
(openSUSE-DVD-x86_64, KDE,        64bit)
(openSUSE-DVD-x86_64, KDE,        Laptop-64bit)
(openSUSE-DVD-x86_64, KDE,        USBBoot-64bit)
(openSUSE-DVD-i586,   KDE,        32bit)
(openSUSE-DVD-i586,   KDE,        Laptop-32bit)
(openSUSE-DVD-x86_64, KDE,        USBBoot-32bit)
(openSUSE-DVD-i586,   KDE,        64bit)
(openSUSE-DVD-i586,   KDE,        Laptop-64bit)
(openSUSE-DVD-x86_64, KDE,        USBBoot-64bit)
```

For every triplet, we need to configure a different instance of os-autoinst with a different set of parameters.

# Medium Types (products)

A medium type (product) in openQA is a simple description without any concrete meaning. It basically consists of a name and a set of variables that define or characterize this product in os-autoinst.

Some example variables used by openSUSE are:

- ISO_MAXSIZE contains the maximum size of the product. There is a test that checks that the current size of the product is less or equal than this variable.
- DVD if it is set to 1, this indicates that the medium is a DVD.
- LIVECD if it is set to 1, this indicates that the medium is a live image (can be a CD or USB)
- GNOME this variable, if it is set to 1, indicates that it is a GNOME only distribution.
- PROMO marks the promotional product.
- RESCUECD is set to 1 for rescue CD images.

# Test Suites

This is the form where we define the different tests that we created for openQA. A test consists of a name, a priority (used in the scheduler to choose the next job) and a set of variables that are used inside this particular test.

Some sample variables used by openSUSE are:

- BTRFS if set, the file system will be BtrFS.
- DESKTOP possible values are 'kde' 'gnome' 'lxde' 'xfce' or 'textmode'. Used to indicate the desktop selected by the user during the test.
- DOCRUN used for documentation tests.
- DUALBOOT dual boot testing, needs HDD_1 and HDDVERSION.
- ENCRYPT encrypt the home directory via YaST.

- HDDVERSION used together with HDD_1 to set the operating system previously installed on the hard disk.
- INSTALLONLY only basic installation.
- INSTLANG installation language. Actually used only in documentation tests.
- LIVETEST the test is on a live medium, do not install the distribution.
- LVM select LVM volume manager.
- NICEVIDEO used for rendering a result video for use in show rooms, skipping ugly and boring tests.
- NOAUTOLOGIN unmark autologin in YaST
- NUMDISKS total number of disks in QEMU.
- REBOOTAFTERINSTALL if set to 1, will reboot after the installation.
- SCREENSHOTINTERVAL used with NICEVIDEO to improve the video quality.
- SPLITUSR a YaST configuration option.
- TOGGLEHOME a YaST configuration option.
- UPGRADE upgrade testing, need HDD_1 and HDDVERSION.
- VIDEOMODE if the value is 'text', the installation will be done in text mode.

Some of the variables usually set in test suites that influence openQA and/or os-autoinst's own behavior are:

- HDDMODEL variable to set the HDD hardware model
- HDDSIZEGB hard disk size in GB. Used together with BtrFS variable
- HDD_1 path for the pre-created hard disk
- RAIDLEVEL RAID configuration variable
- QEMUVGA parameter to declare the video hardware configuration in QEMU

# Machines

You need to have at least one machine set up to be able to run any tests. Those machines represent virtual machine types that you want to test. To make tests actually happen, you have to have an 'openQA worker' connected that can fulfill those specifications.

- **Name.** User defined string - only needed for operator to identify the machine configuration.
- **Backend.** What backend should be used for this machine. Recommended value is qemu as it is the most tested one, but other options (such as kvm2usb or vbox) are also possible.
- **Variables** Most machine variables influence os-autoinst's behavior in terms of how the test machine is set up. A few important examples:
  - QEMUCPU can be 'qemu32' or 'qemu64' and specifies the architecture of the virtual CPU.
  - QEMUCPUS is an integer that specifies the number of cores you wish for.
  - LAPTOP if set to 1, QEMU will create a laptop profile.

- USBBOOT when set to 1, the image will be loaded through an emulated USB stick.

# Variable expansion

Any variable defined in Test Suite, Machine or Product table can refer to another variable using this syntax: %NAME%. When the test job is created, the string will be substituted with the value of the specified variable at that time.

For example this variable defined for Test Suite:

```
PUBLISH_HDD_1 = %DISTRI%-%VERSION%-%ARCH%-%DESKTOP%.qcow2
```

may be expanded to this job variable:

```
PUBLISH_HDD_1 = opensuse-13.1-i586-kde.qcow2
```

# Testing openSUSE or Fedora

An easy way to start using openQA is to start testing openSUSE or Fedora as they have everything setup and prepared to ease the initial deployment. If you want to play deeper, you can configure the whole openQA manually from scratch, but this document should help you to get started faster.

## Getting tests

First you need to get actual tests. You can get openSUSE tests and needles (the expected results) from GitHub. It belongs into the /var/lib/openqa/tests/opensuse directory. To make it easier, you can just run

```
/usr/share/openqa/script/fetchneedles
```

Which will download the tests to the correct location and will set the correct rights as well.

Fedora's tests are also in git. To use them, you may do:

```
cd /var/lib/openqa/share/tests
mkdir fedora
cd fedora
git clone https://bitbucket.org/rajcze/openqa_fedora_tools.git
cd ..
chown -R geekotest fedora/
```

## Getting openQA configuration

To get everything configured to actually run the tests, there are plenty of options to set in the admin interface. If you plan to test openSUSE Factory, using tests mentioned in the previous section, the easiest way to get started is the following command:

```
/var/lib/openqa/share/tests/opensuse/products/opensuse/templates [--apikey API_KEY]
[--apisecret API_SECRET]
```

This will load some default settings that were used at some point of time in openSUSE production openQA. Therefore those should work reasonably well with openSUSE tests and needles. This script uses /usr/share/openqa/script/load_templates, consider reading its help page (--help) for documentation on possible extra arguments.

For Fedora, similarly, you can call:

```
/var/lib/openqa/share/tests/fedora/templates [--apikey API_KEY] [--apisecret
API_SECRET]
```

Some Fedora tests require special hard disk images to be present in /var/lib/openqa/share/factory/hdd/fixed. The createhdds.py script in the openqa_fedora_tools repository can be used to create these. See the documentation in that repo for more information.

# Adding a new ISO to test

To start testing a new ISO put it in /var/lib/openqa/share/factory/iso and call the following commands:

```
# Run the first test
/usr/share/openqa/script/client isos post \
         ISO=openSUSE-Factory-NET-x86_64-Build0053-Media.iso \
         DISTRI=opensuse \
         VERSION=Factory \
         FLAVOR=NET \
         ARCH=x86_64 \
         BUILD=0053
```

If your openQA is not running on port 80 on 'localhost', you can add option --host=http://otherhost:9526 to specify a different port or host.

| | |
|---|---|
| **WARNING** | Use only the ISO filename in the 'client' command. You must place the file in /var/lib/openqa/share/factory/iso. You cannot place the file elsewhere and specify its path in the command. |

For Fedora, a sample run might be:

```
# Run the first test
/usr/share/openqa/script/client isos post \
         ISO=Fedora-Everything-boot-x86_64-Rawhide-20160308.n.0.iso \
         DISTRI=fedora \
         VERSION=Rawhide \
         FLAVOR=Everything-boot-iso \
         ARCH=x86_64 \
         BUILD=Rawhide-20160308.n.0
```

# Pitfalls

Take a look at Documented Pitfalls.

# openQA installation guide

# Introduction

openQA is an automated test tool that makes it possible to test the whole installation process of an operating system. It's free software released under the GPLv2 license. The source code and documentation are hosted in the os-autoinst organization on GitHub.

This document provides the information needed to install and setup the tool, as well as information useful for everyday administration of the system. It's assumed that the reader is already familiar with openQA and has already read the Starter Guide, available at the official repository.

# Installation

The easiest way to install openQA is from packages. You can find openSUSE packages in OBS in the openQA:stable repository.

The latest development version can also be found in OBS in the openQA:devel repository.

For Fedora, packages are available in the official repositories for Fedora 23 and later. Installation on these distributions is therefore pretty simple:

```
# openSUSE Leap 42.2
zypper ar -f obs://devel:openQA/openSUSE_Leap_42.2 openQA
zypper ar -f obs://devel:openQA:Leap:42.2/openSUSE_Leap_42.2 openQA-perl-modules

# openSUSE Tumbleweed
zypper ar -f obs://devel:openQA/openSUSE_Tumbleweed openQA

# all openSUSE
zypper in openQA

# Fedora 23+
dnf install openqa openqa-httpd
```

# Basic configuration

## Apache proxy

It is required to run openQA behind an http proxy (apache, nginx, etc..). See the openqa.conf.template config file in /etc/apache2/vhosts.d (openSUSE) or /etc/httpd/conf.d (Fedora). To make everything work correctly on openSUSE, you need to enable the 'headers', 'proxy', 'proxy_http' and 'proxy_wstunnel' modules using the command 'a2enmod'. This is not necessary on Fedora. For a basic setup, you can copy openqa.conf.template to openqa.conf and modify the ServerName setting. This will direct all HTTP traffic to openQA.

```
# openSUSE Only
# You can check what modules are enabled by using 'a2enmod -l'
a2enmod headers
a2enmod proxy
a2enmod proxy_http
a2enmod proxy_wstunnel
```

## TLS/SSL

By default openQA expects to be run with HTTPS. The openqa-ssl.conf.template Apache config file is available as a base for creating the Apache config; you can copy it to openqa-ssl.conf and uncomment any lines you like, then ensure a key and certificate are installed to the appropriate location (depending on distribution and whether you uncommented the lines for key and cert location in the config file). If you don't have a TLS/SSL certificate for your host you must turn HTTPS off. You can do that in /etc/openqa/openqa.ini:

```
[openid]
httpsonly = 0
```

# Run the web UI

```
systemctl start openqa-scheduler
systemctl start openqa-gru
systemctl start openqa-websockets
systemctl start openqa-webui
# openSUSE
systemctl restart apache2
# Fedora
# for now this is necessary to allow Apache to connect to openQA
setsebool -P httpd_can_network_connect 1
systemctl restart httpd
```

The openQA web UI should be available on http://localhost/ now. To ensure openQA runs on each boot, you should also systemctl enable the same services.

```
systemctl enable openqa-scheduler
systemctl enable openqa-gru
systemctl enable openqa-websockets
systemctl enable openqa-webui
```

# Run workers

Workers are processes running virtual machines to perform the actual testing. They are distributed as a separate package and can be installed on multiple machines but still using only one WebUI.

```
# openSUSE
zypper in openQA-worker
# Fedora
dnf install openqa-worker
```

To allow workers to access your instance, you need to log into openQA as operator and create a pair of API key and secret. Once you are logged in, in the top right corner, is the user menu, follow the link 'manage API keys'. Click the 'create' button to generate key and secret. There is also a script available for creating an admin user and an API key+secret pair non-interactively, /usr/share/openqa/script/create_admin, which can be useful for scripted deployments of openQA. Copy and paste the key and secret into /etc/openqa/client.conf on the machine(s) where the worker is installed. Make sure to put in a section reflecting your webserver URL. In the simplest case, your client.conf may look like this:

```
[localhost]
key = 0123456789ABCDEF
secret = 0123456789ABCDEF
```

To start the workers you can use the provided systemd files via systemctl start openqa-worker@1. This will start worker number one. You can start as many workers as you dare, you just need to supply different 'worker id' (number after @).

You can also run workers manually from command line.

```
sudo -u _openqa-worker /usr/share/openqa/script/worker --instance X
```

This will run a worker manually showing you debug output. If you haven't installed 'os-autoinst' from packages make sure to pass --isotovideo option to point to the checkout dir where isotovideo is, not to /usr/lib! Otherwise it will have trouble finding its perl modules.

# Where to now?

From this point on, you can refer to the getting started guide to fetch the tests cases and possibly take a look at Test Developer Guide

# Advanced configuration

## User authentication

OpenQA supports three different authentication methods - OpenID (default), iChain and Fake. See auth section in /etc/openqa/openqa.ini.

```
[auth]
# method name is case sensitive!
method = OpenID|iChain|Fake
```

Independently of method used, the first user that logs in (if there is no admin yet) will automatically get administrator rights!

## OpenID

By default openQA uses OpenID with opensuse.org as OpenID provider. OpenID method has its own openid section in /etc/openqa/openqa.ini:

```
[openid]
## base url for openid provider
provider = https://www.opensuse.org/openid/user/
## enforce redirect back to https
httpsonly = 1
```

OpenQA supports only OpenID version up to 2.0. Newer OpenID-Connect and OAuth is not supported currently.

## iChain

Use only if you use iChain (NetIQ Access Manager) proxy on your hosting server.

## Fake

For development purposes only! Fake authentication bypass any authentication and automatically allow any login requests as 'Demo user' with administrator privileges and without password. To ease worker testing, API key and secret is created (or updated) with validity of one day during login. You can then use following as /etc/openqa/client.conf:

```
[localhost]
key = 1234567890ABCDEF
secret = 1234567890ABCDEF
```

If you switch authentication method from Fake to any other, review your API keys! You may be

vulnerable for up to a day until Fake API key expires.

## Setting up git support

Editing needles from web can optionally commit new or changed needles automatically to git. To do so, you need to enable git support by setting

```
[global]
scm = git
```

in /etc/openqa/openqa.ini. Once you do so and restart the web interface, openQA will automatically commit new needles to the git repository.

You may want to add some description to automatic commits coming from the web UI. You can do so by setting your configuration in the repository (/var/lib/os-autoinst/needles/.git/config) to some reasonable defaults such as:

```
[user]
    email = whatever@example.com
    name = openQA web UI
```

To enable automatic pushing of the repo as well, you need to add the following to your openqa.ini:

```
[scm git]
do_push = yes
```

Depending on your setup, you might need to generate and propagate ssh keys for user 'geekotest' to be able to push.

## Worker settings

Default behavior for all workers is to use the 'Qemu' backend and connect to 'http://localhost'. If you want to change some of those options, you can do so in /etc/openqa/workers.ini. For example to point the workers to the FQDN of your host (needed if test cases need to access files of the host) use the following setting:

```
[global]
HOST = http://openqa.example.com
```

Once you got workers running they should show up in the admin section of openQA in the workers section as 'idle'. When you get so far, you have your own instance of openQA up and running and all that is left is to set up some tests.

# Configuring remote workers

There are some additional requirements to get remote worker running. First is to ensure shared storage between openQA WebUI and workers. Directory /var/lib/openqa/share contains all required data and should be shared with read-write access across all nodes present in openQA cluster. This step is intentionally left on system administrator to choose proper shared storage for her specific needs.

Example of NFS configuration: NFS server is where openQA WebUI is running. Content of /etc/exports

```
/var/lib/openqa/share *(fsid=0,rw,no_root_squash,sync,no_subtree_check)
```

NFS clients are where openQA workers are running. Run following command:

```
mount -t nfs openQA-webUI-host:/var/lib/openqa/share /var/lib/openqa/share
```

# Needle Caching

If your network is slow or you experience long time to load needles you might want to consider needle caching. To use needle caching a directory /var/lib/openqa/cache must be created, and right permissions given to the 'geekotest' user. If you install openQA through the repositories, said directory will be created for you.

In the /etc/openqa/workers.ini

```
[global]
CACHEDIRECTORY = /var/lib/openqa/cache
```

# Auditing - tracking openQA changes

Auditing plugin enables openQA administrators to maintain overview about what is happening with the system. Plugin records what event was triggered by whom, when and what the request looked like. Actions done by openQA workers are tracked under user whose API keys are workers using.

Audit log is directly accessible from Admin menu.

Auditing, by default enabled, can be disabled by global configuration option in /etc/openqa/openqa.ini:

```
[global]
audit_enabled = 0
```

The audit section of /etc/openqa/openqa.ini allows to exclude some events from logging using a space separated blacklist:

```
[audit]
blacklist = job_grab job_done
```

## List of events tracked by the auditing plugin:

- Assets:
  - asset_register asset_delete
- Workers:
  - worker_register command_enqueue
- Jobs:
  - iso_create iso_delete iso_cancel
  - jobtemplate_create jobtemplate_delete
  - job_create job_grab job_delete job_update_result job_done jobs_restart job_restart job_cancel job_duplicate
  - jobgroup_create jobgroup_connect
- Tables:
  - table_create table_update table_delete
- Users:
  - user_new_comment user_update_comment user_delete_comment user_login
- Needles:
  - needle_delete needle_modify

Some of these events are very common and may clutter audit database. For this reason job_grab and job_done events are blacklisted by default.

| | |
|---|---|
| **NOTE** | Upgrading openQA does not automatically update /etc/openqa/openqa.ini. Review your configuration after upgrade. |

# Filesystem Layout

The openQA web interface can be started via MOJO_REVERSE_PROXY=1 morbo script/openqa in development mode.

/var/lib/openqa/ must be owned by root and contain several sub directories, most of which must be owned by the user that runs openQA (default 'geekotest'):

- db contains the sqlite database
- images is where the server stores test screenshots and thumbnails
- share contains shared directories for remote workers, can be owned by root
- share/factory contains test assets and temp directory, can be owned by root but sysadmin must create subdirs
- share/factory/iso and share/factory/iso/fixed contain ISOs for tests
- share/factory/hdd and share/factory/hdd/fixed contain hard disk images for tests
- share/factory/repo and share/factory/repo/fixed contain repositories for tests
- share/factory/other and share/factory/other/fixed contain miscellaneous test assets (e.g. kernels and initrds)
- share/factory/tmp is used as a temporary directory (openQA will create it if it owns share/factory)
- share/tests contains the tests themselves
- testresults is where the server stores test logs and test-generated assets

Each of the asset directories (factory/iso, factory/hdd, factory/repo and factory/other) may contain a fixed/ subdirectory, and assets of the same type may be placed in that directory. Placing an asset in the fixed/ subdirectory indicates that it should not be deleted to save space: the GRU task which removes old assets when the size of all assets for a given job group is above a specified size will ignore assets in the fixed/ subdirectories.

It also contains several symlinks which are necessary due to various things moving around over the course of openQA's development. All the symlinks can of course be owned by root:

- script (symlink to /usr/share/openqa/script/)
- tests (symlink to share/tests)
- factory (symlink to share/factory)

It is always best to use the canonical locations, not the compatibility symlinks - so run scripts from /usr/share/openqa/script, not /var/lib/openqa/script.

You only need the asset directories for the asset types you will actually use, e.g. if none of your tests refer to openQA-stored repositories, you will need no factory/repo directory. The distribution packages may not create all asset directories, so make sure the ones you need are created if necessary. Packages will likewise usually not contain any tests; you must create your own tests, or use existing tests for some distribution or other piece of software.

The worker needs to own /var/lib/openqa/pool/$INSTANCE, e.g. * /var/lib/openqa/pool/1 * /var/lib/openqa/pool/2 * …. - add more if you have more CPUs/disks

You can also give the whole pool directory to the _openqa-worker user and let the workers create their own instance directories.

# Other database engines

By default, openQA will use an SQLite database: /var/lib/openqa/db/db.sqlite. This will be automatically created on first access to the openQA web UI, if it does not exist.

It is possible to use PostgreSQL or MariaDB / MySQL instead of SQLite, and indeed this is recommended for production deployments of openQA. You should create a database and a dedicated user account with full access to it. To configure access to the chosen database in openQA, edit /etc/openqa/database.ini and change the settings in the [production] section.

## Example for connecting to local PostgreSQL database

```
[production]
dsn = dbi:Pg:dbname=openqa
```

## Example for connecting to remote PostgreSQL database

```
[production]
dsn = dbi:Pg:dbname=openqa;host=db.example.org
user = openqa
password = somepassword
```

## General notes

The dsn value format technically depends on the database type (though at time of writing it's in fact identical for both supported databases). For PostgreSQL it's documented at DBD::Pg, for MySQL / MariaDB it's documented at DBD::mysql

If you intend to use a different database, it is best to create the database and configuration file before starting the services and connecting to the web UI for the first time, otherwise openQA will set itself up with an SQLite database and may get confused when you try to switch to a different one. See the following section if you want to migrate an existing openQA-on-SQLite deployment to a different database.

## Changing the database engine

openQA is compatible with several database engines and comes with all the needed tools to initialize a clean database in any of them. But openQA does not include tools to migrate the existing data from a database to another. If you are planning, for example, to leave behind SQLite and switch to PostgreSQL in your openQA installation, you will need to start with a clean database or perform the data conversion by yourself.

Converting databases from one engine to another is far from trivial. There are plenty of tools, both

commercial and free, that try to address the problem for different databases and in different ways. The following example SQL scritps are provided just as a starting point for those willing to migrate from SQLite (the default engine) to PostgreSQL (successfully backing the biggest openQA installations at the time of writing). Keep in mind that the scripts will probably need some previous work, since they are based on the version 22 of the database schema (likely outdated at the time of reading).

First, run this in the SQLite database to dump the database content into a bunch of CSV files.

```
.mode csv
.header ON
.output assets.csv
SELECT * FROM assets;
.output job_settings.csv
SELECT * FROM job_settings;
.output machine_settings.csv
SELECT * FROM machine_settings;
.output machines.csv
SELECT * FROM machines;
.output product_settings.csv
SELECT * FROM product_settings;
.output products.csv
SELECT * FROM products;
.output secrets.csv
SELECT * FROM secrets;
.output test_suite_settings.csv
SELECT * FROM test_suite_settings;
.output test_suites.csv
SELECT * FROM test_suites;
.output users.csv
SELECT * FROM users;
.output worker_properties.csv
SELECT * FROM worker_properties;
.output workers.csv
SELECT * FROM workers WHERE id > 0;
.output api_keys.csv
SELECT * FROM api_keys;
.output job_modules.csv
SELECT * FROM job_modules;
.output job_templates.csv
SELECT * FROM job_templates;
.output jobs.csv
SELECT * FROM jobs;
.output job_dependencies.csv
SELECT * FROM job_dependencies;
.output jobs_assets.csv
SELECT * FROM jobs_assets;
```

Then, initialize the PostgreSQL database using the standard procedure and afterwards run this

script from the directory containing the CSV files to import them into the new database.

```
\copy users FROM users.csv WITH csv header NULL AS ''
\copy api_keys FROM api_keys.csv WITH csv header NULL AS ''
\copy secrets FROM secrets.csv WITH csv header NULL AS ''
\copy assets FROM assets.csv WITH csv header NULL AS ''
\copy workers FROM workers.csv WITH csv header NULL AS ''
\copy worker_properties FROM worker_properties.csv WITH csv header NULL AS ''
\copy products FROM products.csv WITH csv header NULL AS ''
\copy product_settings FROM product_settings.csv WITH csv header NULL AS ''
\copy machines FROM machines.csv WITH csv header NULL AS ''
\copy machine_settings FROM machine_settings.csv WITH csv header NULL AS ''
\copy test_suites FROM test_suites.csv WITH csv header NULL AS ''
\copy test_suite_settings FROM test_suite_settings.csv WITH csv header NULL AS ''
\copy job_templates FROM job_templates.csv WITH csv header NULL AS ''
\copy jobs FROM jobs.csv WITH csv header NULL AS ''
\copy job_settings FROM job_settings.csv WITH csv header NULL AS ''
\copy job_modules FROM job_modules.csv WITH csv header NULL AS ''
\copy job_dependencies FROM job_dependencies.csv WITH csv header NULL AS ''
\copy jobs_assets FROM jobs_assets.csv WITH csv header NULL AS ''
SELECT SETVAL('users_id_seq', (SELECT MAX(id) FROM users));
SELECT SETVAL('api_keys_id_seq', (SELECT MAX(id) FROM api_keys));
SELECT SETVAL('secrets_id_seq', (SELECT MAX(id) FROM secrets));
SELECT SETVAL('assets_id_seq', (SELECT MAX(id) FROM assets));
SELECT SETVAL('workers_id_seq', (SELECT MAX(id) FROM workers));
SELECT SETVAL('worker_properties_id_seq', (SELECT MAX(id) FROM worker_properties));
SELECT SETVAL('products_id_seq', (SELECT MAX(id) FROM products));
SELECT SETVAL('product_settings_id_seq', (SELECT MAX(id) FROM product_settings));
SELECT SETVAL('machines_id_seq', (SELECT MAX(id) FROM machines));
SELECT SETVAL('machine_settings_id_seq', (SELECT MAX(id) FROM machine_settings));
SELECT SETVAL('test_suites_id_seq', (SELECT MAX(id) FROM test_suites));
SELECT SETVAL('test_suite_settings_id_seq', (SELECT MAX(id) FROM
test_suite_settings));
SELECT SETVAL('job_templates_id_seq', (SELECT MAX(id) FROM job_templates));
SELECT SETVAL('jobs_id_seq', (SELECT MAX(id) FROM jobs));
SELECT SETVAL('job_settings_id_seq', (SELECT MAX(id) FROM job_settings));
SELECT SETVAL('job_modules_id_seq', (SELECT MAX(id) FROM job_modules));
```

# Troubleshooting

## Tests fail quickly

Check the log files in /var/lib/openqa/testresults

## KVM doesn't work

- make sure you have a machine with kvm support
- make sure kvm_intel or kvm_amd modules are loaded
- make sure you do have virtualization enabled in BIOS
- make sure the '_openqa-worker' user can access /dev/kvm
- make sure you are not already running other hypervisors such as VirtualBox
- when running inside a vm make sure nested virtualization is enabled (pass nested=1 to your kvm module)

## openid login times out

www.opensuse.org's openid provider may have trouble with IPv6. openQA shows a message like this:

```
no_identity_server: Could not determine ID provider from URL.
```

To avoid that switch off IPv6 or add a special route that prevents the system from trying to use IPv6 with www.opensuse.org:

```
ip -6 r a to unreachable 2620:113:8044:66:130:57:66:6/128
```

# openQA users guide

# Introduction

This document provides additional information for use of the web interface or the REST API as well as administration information. For administrators it is recommend to have read the installation guide first to understand the structure of components as well as the configuration of an installed instance.

# Use of the web interface

In general the web UI should be intuitive or self-explanatory. Look out for the little blue help icons and click them for detailed help on specific sections.

Some pages use queries to select what should be shown. The query parameters are generated on clickable links, for example starting from the index page or the group overview page clicking on single builds. On the query pages there can be UI elements to control the parameters, for example to look for more older builds or only show failed jobs or other settings. Additionally, the query parameters can be tweaked by hand if you want to provide a link to specific views.

## /tests/overview - Customizable test overview page

The overview page is configurable by the filter box. Also, some additional query parameters can be provided which can be considered advanced or experimental. For example specifying no build will resolve the latest build which matches the other parameters specified. Specifying no group will show all jobs from all matching job groups. Also specifying multiple groups works, see the following example.

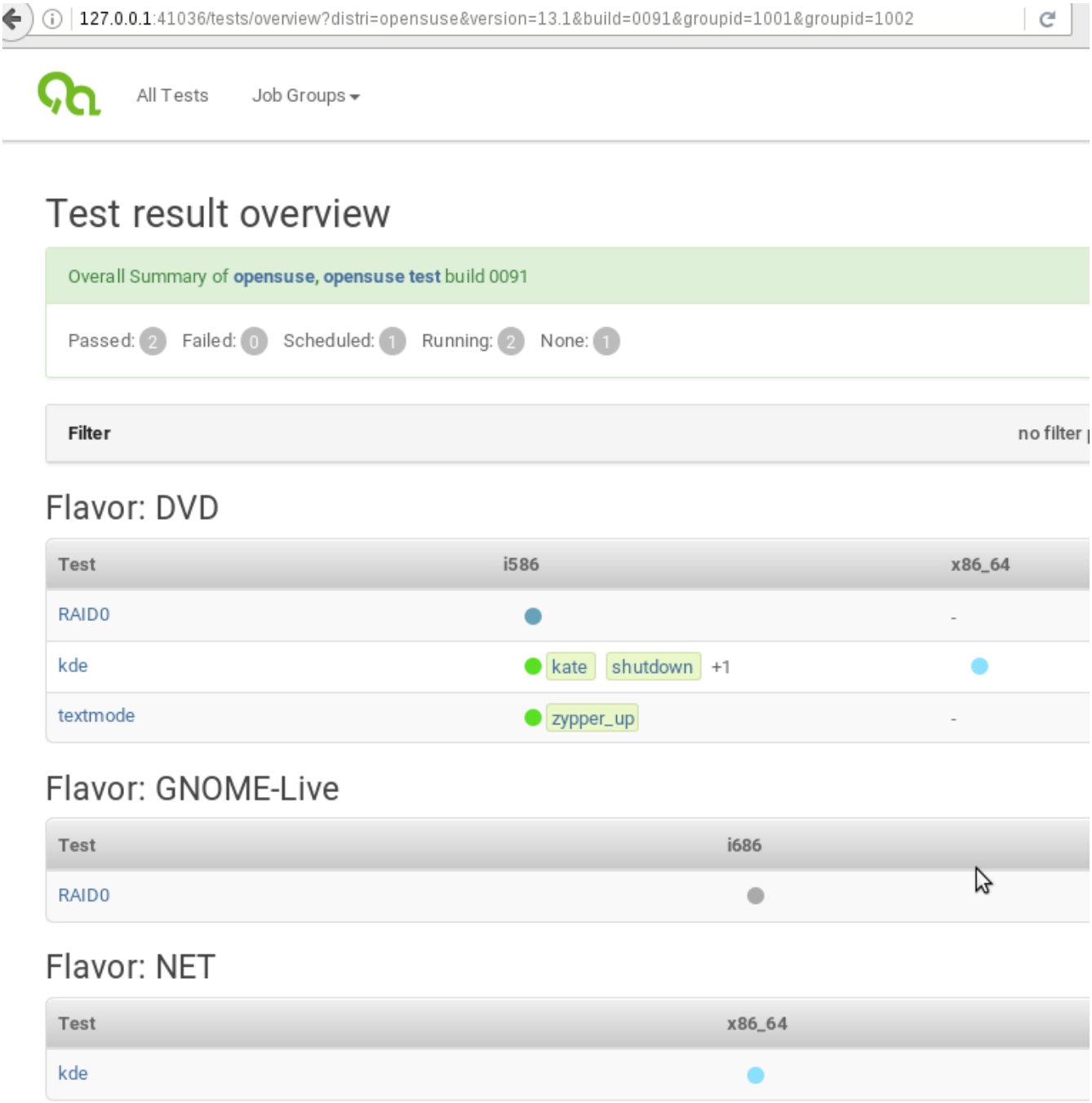



*Figure 2. The openQA test overview page showing multiple groups at once. The URL query parameters specify the groupid parameter two times to resolve both the "opensuse" and "opensuse test" group.*

# Description of test suites

Test suites can be described using API commands or the admin table for any operator using the web UI.


*Figure 3. Entering a test suite description in the admin table using the web interface:*

If a description is defined, the name of the test suite on the tests overview page shows up as a link. Clicking the link will show the description in a popup. The same syntax as for comments can be used, that is Markdown with custom extensions such as shortened links to ticket systems.

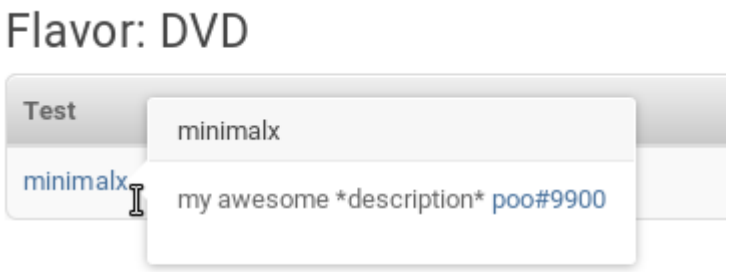
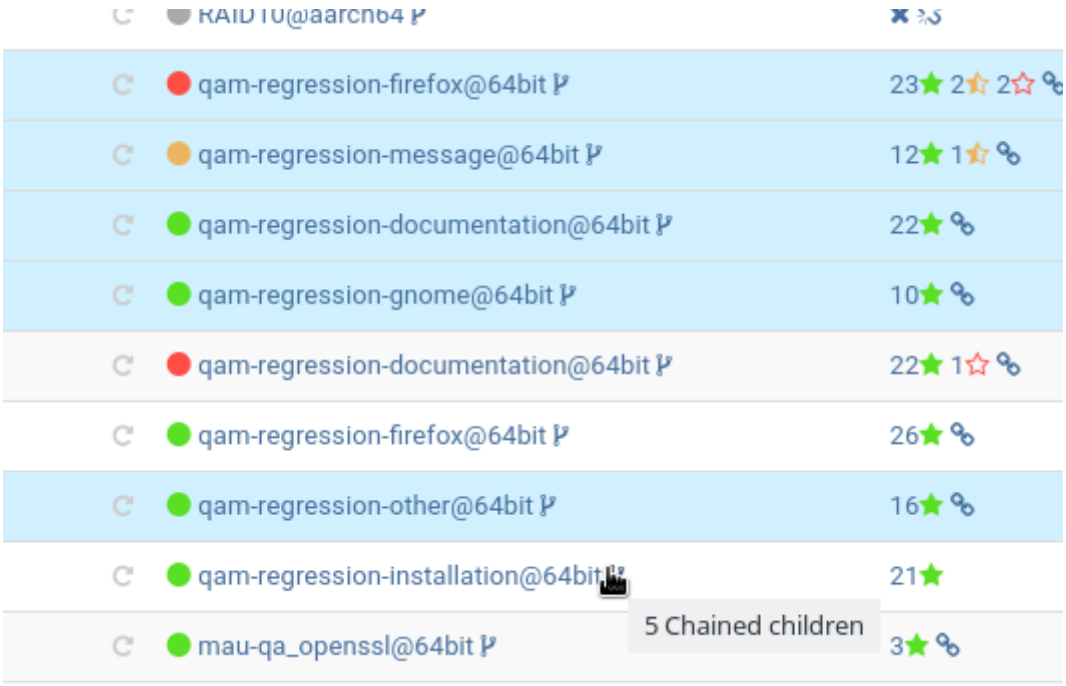



*Figure 4. popover in test overview with content as configured in the test suites database:*

# Highlighting job dependencies in 'All tests' table

When hovering over the branch icon after the test name children of the job will be highlighted blue and parents red. So far this only works for jobs displayed on the same page of the table.



# Use of the REST API

openQA includes a *client* script which - depending on the distribution - is packaged independantly if you just want to interface with an existing openQA instance without needing to install the full package. Call `<openqa-folder>/script/client --help` for help (openSUSE: `openqa-client --help`).

# Where to now?

For test developers it is recommended to continue with the Test Developer Guide.

# openQA tests developer guide

# Introduction

openQA is an automated test tool that makes it possible to test the whole installation process of an operating system. It's free software released under the GPLv2 license. The source code and documentation are hosted in the os-autoinst organization on GitHub.

This document provides the information needed to start developing new tests for openQA or to improve the existing ones. It's assumed that the reader is already familiar with openQA and has already read the Starter Guide, available at the official repository.

# Basic

This section explains the basic layout of openQA tests and the API available in tests. openQA tests are written in the **Perl** programming language. Some basic but no in-depth knowledge of Perl is needed. This document assumes that the reader is already familiar with Perl.

# API

os-autoinst provides the API for the tests using the os-autoinst backend, you can take a look to the published documentation at http://open.qa/api/testapi/.

# How to write tests

openQA tests need to implement at least the **run** subroutine to contain the actual test code and the test needs to be loaded in the distribution's main.pm.

The **test_flags** subroutine specifies what happens when the test fails.

There are several callbacks defined:

- **post_fail_hook** is called to upload log files or determine the state of the machine
- **pre_run_hook** is called before the run function - mainly useful for a whole group of tests
- **post_run_hook** is run after successful run function - mainly useful for a whole group of tests

The following example is a basic test that assumes some live image that boots into the desktop when pressing enter at the boot loader:

```
use base "basetest";
use strict;
use testapi;

sub run {
    # wait for bootloader to appear
    # with a timeout explicitly lower than the default because
    # the bootloader screen will timeout itself
    assert_screen "bootloader", 15;

    # press enter to boot right away
    send_key "ret";

    # wait for the desktop to appear
    assert_screen "desktop", 300;
}

sub test_flags {
    # without anything - rollback to 'lastgood' snapshot if failed
    # 'fatal' - abort whole test suite if this fails (and set overall state)
    # 'milestone' - after this test succeeds, update 'lastgood'
    # 'important' - if this fails, set the overall state to 'failed'
    return { important => 1 };
}

1;
```

# Test Case Examples

*Example: Console test that installs software from remote repository via zypper command*

```
sub run() {
    # change to root
    become_root;

    # output zypper repos to the serial
    script_run "zypper lr -d > /dev/$serialdev";

    # install xdelta and check that the installation was successful
    assert_script_run 'zypper --gpg-auto-import-keys -n in xdelta';

    # additionally write a custom string to serial port for later checking
    script_run "echo 'xdelta_installed' > /dev/$serialdev";

    # detecting whether 'xdelta_installed' appears in the serial within 200 seconds
    die "we could not see expected output" unless wait_serial "xdelta_installed", 200;

    # capture a screenshot and compare with needle 'test-zypper_in'
    assert_screen 'test-zypper_in';
}
```

*Example: Typical X11 test testing kate*

```
sub run() {
    # make sure kate was installed
    # if not ensure_installed will try to install it
    ensure_installed 'kate';

    # start kate
    x11_start_program 'kate';

    # check that kate execution succeeded
    assert_screen 'kate-welcome_window';

    # close kate's welcome window and wait for the window to disappear before
    # continuing
    wait_screen_change { send_key 'alt-c' };

    # typing a string in the editor window of kate
    type_string "If you can see this text kate is working.\n";

    # check the result
    assert_screen 'kate-text_shown';

    # quit kate
    send_key 'ctrl-q';

    # make sure kate was closed
    assert_screen 'desktop';
}
```

# Variables

Test case behavior can be controlled via variables. Some basic variables like DISTRI, VERSION, ARCH are always set. Others like DESKTOP are defined by the 'Test suites' in the openQA web UI. Check the existing tests at os-autoinst-distri-opensuse on GitHub for examples.

Variables are accessible via the **get_var** and **check_var** functions.

# Test Development tricks

## Modifying setting of an existing test

There is no interface to modify existing tests but the clone_job.pl script can be used to create a new job that adds, removes or changes settings. This script is located at /usr/share/openqa/script/.

```
/usr/share/openqa/script/clone_job.pl --from localhost --host localhost 42 FOO=bar
BAZ=
```

If you do not want a cloned job to start up in the same job group as the job you cloned from, e.g. to not pollute build results you the job group can be overwritten, too, using the special variable _GROUP. Add the quoted group name, e.g.:

```
clone_job.pl --from localhost 42 _GROUP="openSUSE Tumbleweed"
```

The special group value 0 means that the group connection will be separated and the job will not appear as a job in any job group, e.g.:

```
clone_job.pl --from localhost 42 _GROUP=0
```

## Using snapshots to speed up development of tests

Sometimes it's annoying to run the full installation to adjust some test. It would be nice to have the VM jump to a certain point. QEMU backend provides feature that allows a job to start from a snapshot that might help in this situation.

Depending on the use case, there are two options to help:

- create and **preserve** snapshots for **every test** module run (MAKETESTSNAPSHOTS)
  - offers more flexibility as test can be resumed almost at any point, however disk space requirements are high (expect more than 30GB for one job)
  - this mode is useful for fixing non fatal issues in tests and debugging SUT
- create snapshot **after every successful** test module, **always overwrite** to preserve only latest (TESTDEBUG)
  - allows to skip just before the start of first failed test module, which can be limiting, but there are no additional hardware requirements.
  - this mode is useful for iterative test development

In both modes there is no need to modify tests (i.e. adding milestone test flag, it is implied). In later mode, every test module is also considered fatal. This means the job is aborted after first failed test module.

### Enable snapshots for each module

- Run the worker with --no-cleanup parameter. This will preserve the hard disks after test runs.
- Set MAKETESTSNAPSHOTS=1 on a job. This will make openQA save a snapshot for every test module run. One way to do that is by cloning an existing job and adding the setting:

```
clone_job.pl --from https://openqa.opensuse.org  --host localhost 24
MAKETESTSNAPSHOTS=1
```

- Create a job again, this time setting the SKIPTO variable to the snapshot you need. Again, clone_job.pl comes handy here:

```
clone_job.pl --from https://openqa.opensuse.org  --host localhost 24
SKIPTO=consoletest-yast2_i
```

- Use qemu-img snapshot -l something.img to find out what snapshots are in the image. Snapshots are named `"test module category"-"test module name"` (e.g. `installation-start_install`)

### Storing only the last sucessfull snapshot

- Run the worker with --no-cleanup parameter. This will preserve the hard disks after test runs.
- Set TESTDEBUG=1 on a job. This will make openQA save a snapshot after each successful test module run. Snapshots are overwritten.

```
clone_job.pl --from https://openqa.opensuse.org  --host localhost 24 TESTDEBUG=1
```

- Create a job again, this time setting the SKIPTO variable to the snapshot which failed on previous run. If clone_job script is not used, TESTDEBUG=1 variable must be also included:

```
clone_job.pl --from https://openqa.opensuse.org  --host localhost 24 TESTDEBUG=1
SKIPTO=consoletest-yast2_i
```

## Assigning jobs to workers

By default, any worker can get any job with the matching architecture.

This behavior can be changed by setting job variable WORKER_CLASS. Jobs with this variable set (typically via machines or test suites configuration) are assigned only to workers, which have the same variable in the configuration file.

For example, the following configuration ensures, that jobs with WORKER_CLASS=desktop can be assigned *only* to worker instances 1 and 2.

*File: workers.ini*

```
[1]
WORKER_CLASS = desktop

[2]
WORKER_CLASS = desktop

[3]
# WORKER_CLASS is not set
```

# Writing multi-machine tests

Scenarios requiring more than one system under test (SUT), like High Availability testing, are covered as multi-machine tests (MM tests) in this section.

OpenQA approaches multi-machine testing by assigning dependencies between individual jobs. This means the following:

- *everything needed for MM tests must be running as a test job* (or you are on your own), even support infrastructure (custom DHCP, NFS, etc. if required), which in principle is not part of the actual testing, must have a defined test suite so a test job can be created
- OpenQA scheduler makes sure *tests are started as a group* and in right order, *cancelled as a group* if some dependencies are violated and *cloned as a group* if requested.
- OpenQA *does not synchronize* individual steps of the tests.
- OpenQA provides *locking server for basic synchronization* of tests (e.g. wait until services are ready for failover), but the *correct usage of locks is test designer job* (beware deadlocks).

In short, writing multi-machine tests adds a few more layers of complexity:

1. documenting the dependencies and order between individual tests
2. synchronization between individual tests
3. actual technical realization (i.e. custom networking)

## Job dependencies

There are 2 types of dependencies: CHAINED and PARALLEL:

- CHAINED describes when one test case depends on another and both are run sequentially, i.e. KDE test suite is run after and only after Installation test suite is successfully finished and cancelled if fail.

To define CHAINED dependency add variable START_AFTER_TEST with the name(s) of test suite(s) after which selected test suite is supposed to run. Use comma separated list for multiple test suite dependency. E.g. START_AFTER_TEST="kde,dhcp-server"

- PARALLEL describes MM test, test suites are scheduled to run at the same time and managed as

a group. On top of that, PARALLEL also describes test suites dependencies, where some test suites (children) run parallel with other test suites (parents) only when parents are running.

To define PARALLEL dependency, use PARALLEL_WITH variable with the name(s) of test suite(s) which acts as a parent suite(s) to selected test suite. In other words, PARALLEL_WITH describes "I need this test suite to be running during my run". Use comma separated list for multiple test suite dependency. E.g. PARALLEL_WITH="web-server,dhcp-server" Keep in mind that parent job *must be running until all children finish*, else scheduler will cancel child jobs once parent is done.

Job dependencies are only resolved when using the iso controller to create new jobs from job templates. Posting individual jobs manually won't work.

Job dependencies are currently only possible between tests that are scheduled for the same machine.

## OpenQA worker requirements

CHAINED dependency requires only one worker, since dependent jobs will run only after the first one finish. On the other hand PARALLEL dependency requires at *least 2 workers* for simple scenarios.

**Examples:**

*Listing 1. CHAINED - i.e. test basic functionality before going advanced - requires 1 worker*

```
A <- B <- C

Define test suite A,
then define B with variable START_AFTER_TEST=A and then define C with
START_AFTER_TEST=B

-or-

Define test suite A, B
and then define C with START_AFTER_TEST=A,B
In this case however the start order of A and B is not specified.
But C will start only after A, B are successfully done.
```

*Listing 2. PARALLEL basic High-Availability*

```
A
^
B

Define test suite A
and then define B with variable PARALLEL_WITH=A.
A in this case is parent test suite to B and must be running throughout B run.
```

*Listing 3. PARALLEL with multiple parents - i.e. complex support requirements for one test - requires 4 workers*

```
A B C
\ | /
  ^
  D

Define test suites A,B,C
and then define D with PARALLEL_WITH=A,B,C.
A,B,C run in parallel and are parent test suites for D and all must run until D
finish.
```

*Listing 4. PARALLEL with one parent - i.e. running independent tests against one server - requires at least 2 workers*

```
   A
   ^
  /|\
 B C D

Define test suite A
and then define B,C,D with PARALLEL_WITH=A
A is parent test suite for B, C, D (all can run in parallel).
Children B, C, D can run and finish anytime, but A must run until all B, C, D
finishes.
```

# Test synchronization and locking API

OpenQA provides locking server through lock API. To use lock API import lockapi package (*use lockapi;*) in your test file. Lock API provides three functions: mutex_create, mutex_lock, mutex_unlock. Each of these functions take one parameter: name of the lock. Locks are associated with caller`s job - locks can't be unlocked by different job then the one who locked the lock.

mutex_lock tries to lock the mutex lock for caller`s job. If lock is unavailable or locked by someone else, mutex_lock call blocks.

mutex_unlock tries to unlock the mutex lock. If lock is locked by different job, mutex_unlock call blocks. When lock become available or if lock does not exist, call returns without doing anything.

mutex_create create new mutex lock. When lock is created by mutex_create, lock is automatically unlocked. When mutex lock already exists call returns without doing anything.

Locks are addressed by *their name*. This name is *valid in test group* defined by their dependencies. If there are more groups running at the same time and the same lock name is used, these locks are independent of each other.

The mmapi package provides wait_for_children, which the parent can use to wait for the children to complete.

*Example of mmapi: Parent JobWait until login prompt appear, assume services are started*

```
use base "basetest";
use strict;
use testapi;
use lockapi;
use mmapi;

sub run {
    assert_screen 'bootloader';
    assert_screen 'login', 300;

    # services start automatically
    # unlock by creating the lock
    mutex_create('services_ready');

    # wait until all children finish
    wait_for_children;
}
```

*Example of mmapi: Child jobCheck until parent is ready, then start testing services*

```
use base "basetest";
use strict;
use testapi;
use lockapi;

sub run {
    assert_screen 'bootloader';
    assert_screen 'login', 300;

    # this blocks until lock is created then locks and immediately unlocks
    mutex_lock('services_ready');
    mutex_unlock('services_ready');

    # login to continue
    type_string("root\n");
    sleep 1;
    type_string("secret\n");
}
```

Sometimes it is useful to let a parent wait for certain action on a child, for example to verify server state after completed request. In this scenario the child creates a mutex and the parent unlocks it.

The child can however die at any time. To prevent parent deadlock in this situation, parent has to pass child ID as a second parameter to mutex_lock(). If a child job with given ID already finished,

mutex_lock() calls die.

*Example of mmapi: Parent JobWait until the child reaches given point*

```
use base "basetest";
use strict;
use testapi;
use lockapi;
use mmapi;

sub run {
    my $children = get_children();

    # let's suppose there is only one child
    my $child_id = (keys %$children)[0];

    # this blocks until lock is available and then does nothing
    mutex_unlock('child_reached_given_point', $child_id);

    # continue with the test
}
```

Getting information about parents and children

*Example of mmapi: Getting info about parents / children*

```
use base "basetest";
use strict;
use testapi;
use mmapi;

sub run {
    # returns a hash ref containing (id => state) for all children
    my $children = get_children();

    for my $job_id (keys %$children) {
      print "$job_id is cancelled\n" if $children->{$job_id} eq 'cancelled';
    }

    # returns an array with parent ids, all parents are in running state (see Job
dependencies above)
    my $parents = get_parents();

    # let's suppose there is only one parent
    my $parent_id = $parents->[0];

    # any job id can be queried for details with get_job_info()
    # it returns a hash ref containing these keys:
    #   name priority state result worker_id
    #   retry_avbl t_started t_finished test
    #   group_id group settings
    my $parent_info = get_job_info($parent_id);

    # it is possible to query variables set by openqa frontend,
    # this does not work for variables set by backend or by the job at runtime
    my $parent_name = $parent_info->{settings}->{NAME}
    my $parent_desktop = $parent_info->{settings}->{DESKTOP}
    # !!! this does not work, VNC is set by backend !!!
    # my $parent_vnc = $parent_info->{settings}->{VNC}
}
```

# Support Server based tests

The idea is to have a dedicated "helper server" to allow advanced network based testing.

Support server takes advantage of the basic parallel setup as described in the previous section, with the support server being the parent test 'A' and the test needing it being the child test 'B'. This ensures that the test 'B' always have the support server available.

## Preparing the supportserver:

The support server image is created by calling a special test, based on the autoyast test:

```
/usr/share/openqa/script/client jobs post DISTRI=opensuse VERSION=13.2 \
    ISO=openSUSE-13.2-DVD-x86_64.iso  ARCH=x86_64 FLAVOR=Server-DVD \
    TEST=supportserver_generator MACHINE=64bit DESKTOP=textmode  INSTALLONLY=1 \
    AUTOYAST=supportserver/autoyast_supportserver.xml SUPPORT_SERVER_GENERATOR=1 \
    PUBLISH_HDD_1=supportserver.qcow2
```

This produces qemu image 'supportserver.qcow2' that contains the supportserver. The 'autoyast_supportserver.xml' should define correct user and password, as well as packages and the common configuration.

More specific role the supportserver should take is then selected when the server is run in the actual test scenario.

### Using the supportserver:

In the Test suites, the supportserver is defined by setting:

```
HDD_1=supportserver.qcow2
SUPPORT_SERVER=1
SUPPORT_SERVER_ROLES=pxe,qemuproxy
WORKER_CLASS=server,qemu_autoyast_tap_64
```

where the SUPPORT_SERVER_ROLES defines the specific role (see code in 'tests/support_server/setup.pm' for available roles and their definition), and HDD_1 variable must be the name of the supportserver image as defined via PUBLISH_HDD_1 variable during supportserver generation. If the support server is based on older SUSE versions (opensuse 11.x, SLE11SP4..) it may also be needed to add HDDMODEL=virtio-blk. In case of qemu backend, one can also use BOOTFROM=c, for faster boot directly from the HDD_1 image.

Then for the 'child' test using this supportserver, the following additional variable must be set: PARALLEL_WITH=supportserver-pxe-tftp where 'supportserver-pxe-tftp' is the name given to the supportserver in the test suites screen. Once the tests are defined, they can be added to openQA in the usual way:

```
/usr/share/openqa/script/client isos post DISTRI=opensuse VERSION=13.2 \
        ISO=openSUSE-13.2-DVD-x86_64.iso ARCH=x86_64 FLAVOR=Server-DVD
```

where the DISTRI, VERSION, FLAVOR and ARCH correspond to the job group containing the tests. Note that the networking is provided by tap devices, so both jobs should run on machines defined by (apart from others) having NICTYPE=tap, WORKER_CLASS=qemu_autoyast_tap_64.

*Example of Support Server: a simple tftp test*

Let's assume that we want to test tftp client operation. For this, we setup the supportserver as a tftp server:

```
HDD_1=supportserver.qcow2
SUPPORT_SERVER=1
SUPPORT_SERVER_ROLES=dhcp,tftp
WORKER_CLASS=server,qemu_autoyast_tap_64
```

With a test-suites name supportserver-opensuse-tftp.

The actual test 'child' job, will then have to set PARALLEL_WITH=supportserver-opensuse-tftp, and also other variables according to the test requirements. For convenience, we have also started a dhcp server on the supportserver, but even without it, network could be set up manually by assigning a free ip address (e.g. 10.0.2.15) on the system of the test job.

*Example of Support Server: The code in the *:pm module doing the actual tftp test could then look something like the example below*

```
use strict;
use base 'basetest';
use testapi;

sub run {
  my $script="set -e -x\n";
  $script.="echo test >test.txt\n";
  $script.="time tftp ".$server_ip." -c put test.txt test2.txt\n";
  $script.="time tftp ".$server_ip." -c get test2.txt\n";
  $script.="diff -u test.txt test2.txt\n";
  script_output($script);

}
```

assuming of course, that the tested machine was already set up with necessary infrastructure for tftp, e.g. network was set up, tftp rpm installed and tftp service started, etc. All of this could be conveniently achieved using the autoyast installation, as shown in the next section.

*Example of Support Server: autoyast based tftp test*

Here we will use autoyast to setup the system of the test job and the os-autoinst autoyast testing infrastructure. For supportserver, this means using proxy to access qemu provided data, for dowloading autoyast profile and tftp verify script:

```
HDD_1=supportserver.qcow2
SUPPORT_SERVER=1
SUPPORT_SERVER_ROLES=pxe,qemuproxy
WORKER_CLASS=server,qemu_autoyast_tap_64
```

The actual test 'child' job, will then be defined as :

```
AUTOYAST=autoyast_opensuse/opensuse_autoyast_tftp.xml
AUTOYAST_VERIFY=autoyast_opensuse/opensuse_autoyast_tftp.sh
DESKTOP=textmode
INSTALLONLY=1
PARALLEL_WITH=supportserver-opensuse-tftp
```

again assuming the support server's name being supportserver-opensuse-tftp. Note that the pxe role already contains tftp and dhcp server role, since they are needed for the pxe boot to work.

*Example of Support Server: The tftp test defined in the autoyast_opensuse/opensuse_autoyast_tftp.sh file could be something like:*

```
set -e -x
echo test >test.txt
time tftp #SERVER_URL# -c put test.txt test2.txt
time tftp #SERVER_URL# -c get test2.txt
diff -u test.txt test2.txt && echo "AUTOYAST OK"
```

and the rest is done automatically, using already prepared test modules in tests/autoyast subdirectory.

# Using text consoles and the serial terminal

Typically the OS you are testing will boot into a graphical shell e.g. The Gnome desktop environment. This is fine if you wish to test a program with a GUI, but lets say you want to run some shell scripts then it is not so convenient.

*To access a text based console or TTY, you can do something like the*

following.

```
use 5.018;
use warnings;
use base 'opensusebasetest';
use testapi;
use utils;

sub run {
    wait_boot;  # Utility function defined by the SUSE distribution
    select_console 'root-console';
}

1;
```

This will select a text TTY and login as the root user (you could use become_root instead in this case). Had select_console 'root-console' been used before then it would just select the TTY. Now that we are on a text console it is possible to run scripts and observe their output. Note that root-console is defined by the distribution, but also that calls to select_console can have far reaching consequences depending on what console is being selected and what backend/architecture the SUT is using.

*Running a script: Using the assert_script_run and script_output commands*

```
assert_script_run('cd /proc');
my $cpuinfo = script_output('cat cpuinfo');
if($cpuinfo =~ m/avx2/) {
    # Do something which needs avx2
}
else {
    # Do some workaround
}
```

Note that it is usually not necessary to return text from the SUT to the test module for processing and it is often faster to do the processing in a shell script on the SUT. However you may find it more convenient, readable or reliable to do it in the Perl test module.

The script_run and script_output commands are high level commands which use type_string and wait_serial underneath. Sometimes you may wish to use lower level commands which give you more control, but be warned that it may also make your code less portable.

## Using a serial terminal

| | |
|---|---|
| **IMPORTANT** | You need a QEMU version >= 2.6.1 and to set the VIRTIO_CONSOLE variable to 1 to use this with the QEMU backend. |

Usually OpenQA controls the system under test using VNC. This allows the use of both graphical and

text based consoles. Key presses are sent individually as VNC commands and output is returned in the form of screen images and text output from the SUT's serial port.

Sending key presses over VNC is very slow so for tests which send a lot of text commands it is much faster to use a serial port for both sending and receiving TTY commands.

```
select_console('root-virtio-terminal');  # Selects a virtio based serial terminal
```

Changing input and output to a serial terminal has the side effect of changing where wait_serial reads output from. This will cause some distribution specific utility functions to fail, however they can usually be fixed with the is_serial_terminal API function. To find out more look at the is_serial_terminal POD in testapi.pm.

Another consequence of moving to a serial terminal is that none of the needle based commands will be available because there is no screen image to match against.

# openQA pitfalls

# Needle editing

- If a new needle is created based on a failed test, the new needle will not be listed in old tests.
- If an existing needle is updated with a new image or different areas, the old test will display the new needle which might be confusing
- If a needle is deleted, old tests may display an error when viewing them in the web UI.

# Mixed production and development environment

There are few things to take into account when running a development version and a packaged version of openqa:

If the setup for the development scenario involves sharing /var/lib/openqa, it would be wise to have a shared group *openqa*, that will have write and execute permissions over said directory, so that *geekotest* user and the normal development user can share the environment without problems.

This approach will lead to a problem when the openqa package is updated, since the directory permissions will be changed again, nothing a `chmod -R g+rwx /var/lib/openqa/` and `chgrp -R openqa /var/lib/openqa` can not fix.

# Networking in OpenQA

| IMPORTANT | This overview is valid only when using QEMU backend! |
|---|---|

Which networking type to use is controlled by the `NICTYPE` variable. If unset or empty `NICTYPE` defaults to user, ie qemu user networking which requires no further configuration.

For more advanced setups or tests that require multiple jobs to be in the same networking the TAP or VDE based modes can be used.

# Qemu user networking

With Qemu user networking each jobs gets it's own isolated network with TCP and UDP routed to the outside. DHCP is provided by qemu. The MAC address of the machine can be controlled with the NICMAC variable. If not set, it's 52:54:00:12:34:56.

# TAP based network

os-autoinst can connect qemu to TAP devices of the host system to leverage advanced network setups provided by the host by setting NICTYPE=tap.

The TAP device to use can be configured with the TAPDEV variable. If not set defined, ist's automatically set to "tap" + ($worker_instance - 1), i.e. worker1 uses tap0, worker 2 uses tap1 and so on.

For multiple networks per job (see NETWORKS variable), the following numbering scheme is used:

```
worker1: tap0 tap64 tap128 ...
worker2: tap1 tap65 tap129 ...
worker3: tap2 tap66 tap130 ...
...
```

MAC address of virtual NIC is controlled by NICMAC variable or automatically computed from $worker_id if not set.

In TAP mode the system administrator is expected to configure the network, required internet access etc on the host manually.

TAP devices need be owned by the _openqa-worker user for openQA to be able to access them.

```
tunctl -u _openqa-worker -p -t tap0
```

or Wicked way:

*File: /etc/sysconfig/network/ifcfg-tap0*

```
BOOTPROTO='none'
IPADDR=''
NETMASK=''
PREFIXLEN=''
STARTMODE='auto'
TUNNEL='tap'
TUNNEL_SET_GROUP='nogroup'
TUNNEL_SET_OWNER='_openqa-worker'
```

If you want to use TAP device which doesn't exist on the system, you need to set CAP_NET_ADMIN capability on qemu binary file:

```
zypper in libcap-progs
setcap CAP_NET_ADMIN=ep /usr/bin/qemu-system-x86_64
```

Network setup can be changed after qemu is started using network configure script specified in

TAPSCRIPT variable.

Sample script to add TAP device to existing bridge br0:

```
sudo brctl addif br0 $1
sudo ip link set $1 up
```

# TAP with Open vSwitch

The recomended way to configure the network for TAP devices is using Open vSwitch. There is a support service os-autoinst-openvswitch.service which sets vlan number of Open vSwitch ports based on NICVLAN variable - this separates the groups of tests from each other.

NICVLAN variable is dynamically assigned by OpenQA scheduler.

Compared to VDE setup discussed later, Open vSwitch is more complicated to configure, but provides more robust and scalable network.

Start Open vSwitch and add TAP devices:

```
# start openvswitch.service
systemctl start openvswitch.service
systemctl enable openvswitch.service

# create bridge
ovs-vsctl add-br br0

# add tap devices, use vlan 999 by default, the vlan number is supposed to be changed
when the vm starts
ovs-vsctl add-port br0 tap0 tag=999
ovs-vsctl add-port br0 tap1 tag=999
ovs-vsctl add-port br0 tap2 tag=999
ovs-vsctl add-port br0 tap3 tag=999
ovs-vsctl add-port br0 tap4 tag=999
```

If the workers with TAP capability are spread across multiple hosts, the network must be connected. See Open vSwitch documentation for details.

Example configuration of GRE tunnel:

```
ovs-vsctl add-port br0 gre0 -- set interface gre0 type=gre options:remote_ip=<IP
address of other host>
```

The virtual machines need access to os-autoinst webserver acessible via IP 10.0.2.2. The IP addresses of VMs are controlled by tests and are likely to conflict if more independent tests runs in parallel.

The VMs have unique MAC that differs in the last 16 bits (see /usr/lib/os-autoinst/backend/qemu.pm).

`os-autoinst-openvswitch.service` sets up filtering rules for the following translation scheme which provide non-conflicting addresses visible from host:

```
MAC 52:54:00:12:XX:YY -> IP 10.1.XX.YY
```

That means that the local port of the bridge must be configured to IP 10.0.2.2 and netmask /15 that covers 10.0.0.0 and 10.1.0.0 ranges.

```
ip addr add 10.0.2.2/15 dev br0
ip route add 10.0.0.0/15 dev br0
ip link set br0 up
```

and permanently in /etc/sysconfig/network

```
# /etc/sysconfig/network/ifcfg-br0
BOOTPROTO='static'
IPADDR='10.0.2.2/15'
STARTMODE='auto'
```

wicked 0.6.23 and later has enhanced support for the creation and configuration of OpenvSwitch bridges.

| NOTE | In some cases (e.g. on Leap) can be needed to start the OpenvSwitch service before the Network service by modifying the OpenvSwitch service. For reference see this. |
|---|---|

The permanent configuration for wicked 0.6.23 and later should look like this:

```
# /etc/sysconfig/network/ifcfg-br0
BOOTPROTO='static'
IPADDR='10.0.2.2/15'
STARTMODE='auto'
OVS_BRIDGE='yes'
OVS_BRIDGE_PORT_DEVICE_1='tap0'
OVS_BRIDGE_PORT_DEVICE_2='tap1'
OVS_BRIDGE_PORT_DEVICE_3='tap2'
```

The IP 10.0.2.2 can also serve as a gateway to access outside network. For this, a NAT between br0 and eth0 must be configured with SuSEfirewall or iptables.

```
# configuration options for NAT with SuSEfirewall
# /etc/sysconfig/SuSEfirewall

FW_ROUTE="yes"
FW_MASQUERADE="yes"
FW_DEV_INT="br0"
```

Then it is possible to start the os-autoinst-openvswitch.service The service uses br0 by default. It can be configured for another bridge name by setting /etc/sysconfig/os-autoinst-openvswitch

```
OS_AUTOINST_USE_BRIDGE=bridge_name
```

Then, start the service:

```
systemctl start os-autoinst-openvswitch.service
systemctl enable os-autoinst-openvswitch.service
```

# Debugging Open vSwitch configuration

Boot sequence with wicked < 0.6.23:

1. wicked - creates tap devices
2. openvswitch - creates the bridge br0, adds tap devices to it
3. wicked handles br0 as hotplugged device, assignd the IP 10.0.2.2 to it, updates SuSEFirewall
4. os-autoinst-openvswitch - installs openflow rules, handles vlan assignment

Boot sequence with wicked 0.6.23 and newer:

1. openvswitch
2. wicked - creates the bridge br0 and tap devices, add tap devices to the bridge,
3. SuSEFirewall
4. os-autoinst-openvswitch - installs openflow rules, handles vlan assignment

The configuration and operation can be checked by the following commands:

```
ovs-vsctl show # shows the bridge br0, the tap devices are assigned to it
ovs-ofctl dump-flows br0 # shows the rules installed by os-autoinst-openvswitch in
table=0
```

- packets from tapX to br0 create additional rules in table=1
- packets from br0 to tapX increase packet counts in table=1
- empty output indicates a problem with os-autoinst-openvswitch service
- zero packet count or missing rules in table=1 indicate problem with tap devices

```
ipables -L -v
```

As long as the SUT has access to external network, there should be nonzero packet count in the forward chain between br0 and external interface.

# VDE Based Network

Virtual Distributed Ethernet provides a software switch that runs in user space. It allows to connect several qemu instances without affecting the system's network configuration.

The openQA workers need a vde_switch instance running. The workers reconfigure the switch as needed by the job.

## Basic, single machine tests

To start with a basic configuration like qemu user mode networking, create a machine with the following settings:

- VDE_SOCKETDIR=/run/openqa
- NICTYPE=vde
- NICVLAN=0

Start switch and user mode networking:

```
systemctl start openqa-vde_switch
systemctl start openqa-slirpvde
```

With this setting all jobs on the same host would be in the same network share the same SLIRP instance though.

## Multi machine tests

Create a machine like above but don't set NICVLAN. openQA will dynamically allocate a VLAN number for all jobs that have dependencies between each other. By default this VLAN is private and has no internet access. To enable user mode networking set `VDE_USE_SLIRP=1` on one of the machines. The worker running the job on such a machine will start slirpvde and put it in the correct VLAN then.

# openQA developer guide

# Introduction

openQA is an automated test tool that makes it possible to test the whole installation process of an operating system. It's free software released under the GPLv2 license. The source code and documentation are hosted in the os-autoinst organization on GitHub.

This document provides the information needed to start contributing to the openQA development improving the tool, fixing bugs and implementing new features. For information about writing or improving openQA tests, refer to the Tests Developer Guide. In both documents it's assumed that the reader is already familiar with openQA and has already read the Starter Guide. All those documents are available at the official repository.

# Development guidelines

As mentioned, the central point of development is the os-autoinst organization on GitHub where several repositories can be found:

- openQA containing documentation, server, worker and other support scripts.
- os-autoinst with the standalone test tool.
- os-autoinst-distri-opensuse containing the tests used in http://openqa.opensuse.org
- os-autoinst-needles-opensuse with the needles associated to the tests in the former repository.
- os-autoinst-distri-example with an almost empty set of tests meant to be used to start writing tests (and creating the corresponding needles) from scratch for a new operating system.

As in most projects hosted on GitHub, pull request are always welcome and are the right way to contribute improvements and fixes.

## Rules for commits

- Every commit is checked by Travis CI as soon as you create a pull request but you **should** run the openQA tests locally, i.e. before every commit call:

```
./script/tidy
```

to ensure your perl code changes are consistent with the style rules

- You **may** also run local tests on your machine or in your own development environment to verify everything works as expected. Call:

```
make test
```

for unit and integration tests.

- For git commit messages use the rules stated on How to Write a Git Commit Message as a reference
- Every pull request is reviewed in a peer review to give feedback on possible implications and how we can help each other to improve

If this is too much hassle for you feel free to provide incomplete pull requests for consideration or create an issue with a code change proposal.

# Getting involved into development

But developers willing to get really involved into the development of openQA or people interested in following the always-changing roadmap should take a look at the openQAv3 project in openSUSE's project management tool. This Redmine instance is used to coordinate the main development effort organizing the existing issues (bugs and desired features) into 'target versions'.

Currently developers meet in IRC channel #opensuse-factory and in a daily jangouts call of the core developer team.

In addition to the ones representing development sprints, two other versions are always open. Easy hacks lists issues that are not specially urgent and that are considered to be easy to implement by newcomers. Developers looking for a place to start contributing are encouraged to simply go to that list and assign any open issue to themselves. Future improvements groups features that are in the developers' and users' wish list but that have little chances to be addressed in the short term, either because the return of investment is not worth it or because they are out of the current scope of the development.

openQA and os-autoinst repositories also include test suites aimed at preventing bugs and regressions in the software. codecov is configured in the repositories to encourage contributors to raise the tests coverage with every commit and pull request. New features and bug fixes are expected to be backed with the corresponding tests.

# Technologies

Everything in openQA, from os-autoinst to the web frontend and from the tests to the support scripts is written in Perl. So having some basic knowledge about that language is really desirable in order to understand and develop openQA. Of course, in addition to bare Perl, several libraries and additional tools are required. The easiest way to install all needed dependencies is using the available os-autoinst and openQA packages, as described in the Installation Guide.

In the case of os-autoinst, only a few CPAN modules are required. Basically Carp::Always, Data::Dump. JSON and YAML. On the other hand, several external tools are needed including QEMU, Tesseract and OptiPNG. Last but not least, the OpenCV library is the core of the openQA image matching mechanism, so it must be available on the system.

The openQA package is built on top of Mojolicious, an excellent Perl framework for web development that will be extremely familiar to developers coming from other modern web frameworks like Sinatra and that have nice and comprehensive documentation available at its home page.

In addition to Mojolicious and its dependencies, several other CPAN modules are required by the openQA package. For a full list of hard dependencies, see the file DEPENDENCIES.txt at the root of the openQA repository. Some additional modules could be required when using a database engine other than the default SQLite.

As already mentioned, openQA relies on a database engine to store the information. PostgreSQL, MySQL and SQLite3 are supported, with the latter being the default option.

Lastly, png2theora (part of the Theora project) is used to create a video from the screenshots generated by os-autoinst.

As stated in the previous section, every feature implemented in both packages should be backed by proper tests. Test::More is used to implement those tests. As usual, tests are located under the /t/ directory. In the openQA package, one of the tests consists of a call to Perltidy to ensure that the contributed code follows the most common Perl style conventions.

# Managing the database

## How to change the database schema

During the development process there are cases in which the database schema needs to be changed. After modifying files in lib/OpenQA/Schema/Result there are some steps that have to be followed so that new database instances and upgrades include those changes.

1. First, you need to increase the database version number in the `$VERSION` variable in the lib/OpenQA/Schema.pm file. Note that it's recommended to notify the other developers before doing so, to synchronize in case there are more developers wanting to increase the version number at the same time.
2. Then you need to generate the deployment files for new installations, this is done by running ./script/initdb --prepare_init.
3. Afterwards you need to generate the deployment files for existing installations, this is done by running ./script/upgradedb --prepare_upgrade. After doing so, the directories dbicdh/$ENGINE/deploy/<new version> and dbicdh/$ENGINE/upgrade/<prev version>-<new version> for SQLite and PosgreSQL should have been created with some SQL files inside containing the statements to initialize the schema and to upgrade from one version to the next in the corresponding database engine.
4. And finally, you need to create the fixtures files. Under dbicdh/_common/deploy, rename the directory of the (previous) latest version to the new version and do the necessary changes (if any). Then, under dbicdh/_common/upgrade create a <prev_version>-<new_version> directory and put some files there with SQL statements that upgrade the fixtures. Usually a diff from the previous version to the new one helps to see what has to be in the upgrade file.

The above steps are executed in the developer's system. Once openQA is installed in a production server, you should run either ./script/initdb --init_database or ./script/upgradedb --upgrade_database to actually create or upgrade a database.

## How to add fixtures to the database

Fixtures (initial data stored in tables at installation time) are stored in files into the dbicdh/_common/deploy/_any/<version> and dbicdh/_common/upgrade/<prev_version>-<next_version> directories.

You can create as many files as you want in each directory. These files contain SQL statements that will be executed when initializing or upgrading a database. Note that those files (and directories) have to be created manually and they shouldn't create a transaction, since each file is already automatically executed in its own transaction (so that changes are rolled back if there's any problem) and sqlite doesn't support nested transactions.

Executed SQL statements can be traced by setting the DBIC_TRACE environment variable.

```
export DBIC_TRACE=1
```

# How to overwrite config files

It can be necessary during development to change the config files in etc/. For example you have to edit etc/openqa/database.ini to use another database. Or to increase the log level it's useful to set the loglevel to debug in etc/openqa/openqa.ini.

To avoid these changes getting in your git workflow, copy them to a new directory and set OPENQA_CONFIG in your shell setup files.

```
cp -ar etc/openqa etc/mine
export OPENQA_CONFIG=$PWD/etc/mine
```

Note that OPENQA_CONFIG points to the directory containing openqa.ini, database.ini, client.conf and workers.ini.

# How to setup PostgreSQL to test locally with production data

1. Install PosgreSQL - under openSUSE the following package are required: postgresql-server postgresql-init
2. Start the server: systemctl start postgresql
3. The following steps need to be done by the user postgres: su - postgres
4. Create user: createuser your_username where your_username must be the same as the UNIX user you start your local openQA instance with.
5. Create database: createdb -O your_username openqa
6. The next steps must be done by the user you start your local openQA instance with.
7. Import dump: pg_restore -c -d openqa path/to/dump
8. Configure openQA to use PostgreSQL as described in the section Other database engines of the installation guide. User name and password are not required.

# Adding new authentication module

OpenQA comes with three authentication modules providing authentication methods: OpenID, iChain and Fake (see User authentication).

All authentication modules reside in lib/OpenQA/Auth directory. During OpenQA start, [auth]/method section of /etc/openqa/openqa.ini is read and according to its value (or default OpenID) OpenQA tries to require OpenQA::WebAPI::Auth::$method. If successful, module for given method is imported or the OpenQA ends with error.

Each authentication module is expected to export auth_config, auth_login and auth_logout functions. In case of request-response mechanism (as in OpenID), auth_response is imported on demand.

Currently there is no login page because all implemented methods use either 3rd party page or none.

Authentication module is expected to return HASH:

```
%res = (
    # error = 1 signals auth error
    error => 0|1
    # where to redirect the user
    redirect => ''
);
```

Authentication module is expected to create or update user entry in OpenQA database after user validation. See included modules for inspiration.

# Customize base directory

It is possible to customize the openQA base directory by setting the environment variable OPENQA_BASEDIR. The default value is /var/lib.

# openQA branding

You can alter the appearance of the openQA web UI to some extent through the 'branding' mechanism. The 'branding' configuration setting in the 'global' section of /etc/openqa/openqa.ini specifies the branding to use. It defaults to 'openSUSE', and openQA also includes the 'plain' branding, which is - as its name suggests - plain and generic.

To create your own branding for openQA, you can create a subdirectory of /usr/share/openqa/templates/branding (or wherever openQA is installed). The subdirectory's name will be the name of your branding. You can copy the files from branding/openSUSE or branding/plain to use as starting points, and adjust as necessary.

# Web UI template

openQA uses the Mojolicious framework's templating system; the branding files are included into the openQA templates at various points. To see where each branding file is actually included, you can search through the files in the templates tree for the text include_branding. Anywhere that helper is called, the branding file with the matching name is being included.

The branding files themselves are Mojolicious 'Embedded Perl' templates just like the main template files. You can read the Mojolicious Documentation for help with the format.