

# openQA Documentation

openQA Team

# Table of Contents

openQA starter guide .....	1
Introduction .....	2
Architecture .....	3
Basic concepts .....	4
Glossary .....	4
Jobs .....	5
Needles .....	6
Areas .....	6
Access management .....	7
Job groups .....	7
Cleanup .....	7
Using the client script .....	9
Using job templates to automate jobs creation .....	10
The problem .....	10
Medium Types (products) .....	11
Test Suites .....	11
Machines .....	12
Variable expansion .....	13
Variable precedence .....	13
Testing openSUSE or Fedora .....	14
Getting tests .....	14
Getting openQA configuration .....	14
Adding a new ISO to test .....	15
Pitfalls .....	16
openQA installation guide .....	17
Introduction .....	18
Repositories and installation .....	19
Official repositories .....	19
Development version repository .....	19
Installation .....	19
Basic configuration .....	20
Apache proxy .....	20
TLS/SSL .....	20
Database .....	21
Example for connecting to local PostgreSQL database .....	21
Example for connecting to remote PostgreSQL database .....	21
User authentication .....	21
OpenID .....	21

iChain .....	22
Fake .....	22
Run the web UI .....	23
Run workers .....	24
Where to now? .....	25
Advanced configuration .....	26
Setting up git support .....	26
Referer settings to auto-mark important jobs .....	26
Worker settings .....	27
Configuring remote workers .....	27
Configuring AMQP message emission .....	27
Configuring worker to use more than one openQA server .....	28
Asset Caching .....	29
Auditing - tracking openQA changes .....	31
List of events tracked by the auditing plugin: .....	31
Filesystem Layout .....	33
Troubleshooting .....	35
Tests fail quickly .....	35
KVM doesn't work .....	35
openid login times out .....	35
openQA users guide .....	36
Introduction .....	37
Use of the web interface .....	38
/tests/overview - Customizable test overview page .....	38
Description of test suites .....	39
Review badges .....	40
Meaning of the different colors .....	41
Show bug or label icon on overview if labeled gh#550 .....	41
Build tagging .....	42
Tag builds with special comments on group overview .....	42
Keeping important builds .....	43
Filtering test results and builds .....	43
Highlighting job dependencies in 'All tests' table .....	44
Asset cleanup .....	45
Cleanup strategy .....	45
Configure limit for assets within groups .....	45
Configure limit for groupless assets .....	45
Use of the REST API .....	46
Triggering tests .....	46
Cloning existing jobs - clone_job.pl .....	46
Spawning single new jobs - jobs post .....	46

Spawning multiple jobs based on templates - isos post .....	46
Asset handling .....	47
Where to now? .....	48
openQA tests developer guide .....	49
Introduction .....	50
Basic .....	51
API .....	52
How to write tests .....	53
Test Case Examples .....	54
Variables .....	57
Advanced test features .....	58
Capturing kernel exceptions and/or any other exceptions from the serial console .....	58
Assigning jobs to workers .....	59
Writing multi-machine tests .....	59
Job dependencies .....	60
Inter-machine dependencies .....	60
Example: best match with correct dependency and machine placed .....	61
Example: wrong dependency or wrong machine placed .....	61
OpenQA worker requirements .....	62
Examples .....	62
Test synchronization and locking API .....	63
Support Server based tests .....	67
Preparing the supportserver: .....	67
Using the supportserver: .....	68
Using text consoles and the serial terminal .....	70
Using a serial terminal .....	72
Sending new lines and continuation characters .....	75
Sending signals - ctrl-c and ctrl-d .....	75
The virtio serial terminal implementation .....	76
Test Development tricks .....	78
Modifying setting of an existing test .....	78
Backend variables for faster test execution .....	78
Using snapshots to speed up development of tests .....	78
Enable snapshots for each module .....	79
Storing only the last successful snapshot .....	79
openQA test harness result processing .....	80
Introduction .....	81
Usage .....	82
openQA test distribution .....	82
Available parser formats .....	83
Extending the parser .....	84

OOP Interface .....	84
Structured data .....	84
openQA internal test result storage .....	85
openQA pitfalls .....	86
Needle editing .....	87
403 messages when using scripts .....	88
Mixed production and development environment .....	89
Performance impact .....	90
DB migration from SQLite to postgresQL .....	91
Networking in OpenQA .....	92
Qemu User Networking .....	93
TAP Based Network .....	93
VDE Based Network .....	93
Basic, Single Machine Tests .....	93
Multi Machine Tests Setup .....	95
Debugging Open vSwitch Configuration .....	99
GRE Tunnels .....	101
openQA developer guide .....	102
Introduction .....	103
Development guidelines .....	104
Rules for commits .....	104
Getting involved into development .....	106
Technologies .....	107
Starting the webserver from local Git checkout .....	108
Managing the database .....	109
When is it required to update the database schema? .....	109
How to update the database schema .....	109
How to add fixtures to the database .....	110
How to overwrite config files .....	111
How to setup PostgreSQL to test locally with production data .....	112
Adding new authentication module .....	113
Customize base directory .....	114
Running tests of openQA itself .....	115
How to run tests with docker .....	115
tips .....	116
How to run tests without docker .....	117
openQA branding .....	118
Web UI template .....	119

# 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).
- **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, ...
  ]
}
```

## Areas

There are three kinds of areas:

- **Regular areas** define relevant parts of the screenshot. Those must match with at least the specified similarity percentage. Regular areas are displayed as green boxes in the needle editor and as green or red frames in the needle view (green for matching areas, red for non-matching ones).
- **OCR areas** also define relevant parts of the screenshot. However, an OCR algorithm is used for matching. In the needle editor OCR areas are displayed as orange boxes. To turn a regular area into an OCR area within the needle editor, double click the concerning area twice. Note that such needles are only rarely used.
- **Exclude areas** can be used to ignore parts of the reference picture. In the needle editor exclude areas are displayed as red boxes. To turn a regular area into an exclude area within the needle editor, double click the concerning area. In the needle view exclude areas are displayed as gray boxes.

# 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-i586,	KDE,	USBBoot-32bit)
(openSUSE-DVD-i586,	KDE,	64bit)
(openSUSE-DVD-i586,	KDE,	Laptop-64bit)
(openSUSE-DVD-i586,	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 and a set of variables that are used inside this particular test. The priority is used in the scheduler to choose the next job. If multiple jobs are scheduled and their requirements for running them are fulfilled the ones with a lower value for the priority are triggered. The id is the second sorting key: Of two jobs with equal requirements and same priority the one with lower id is triggered first.

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 = %DISTRIB%-%VERSION%-%ARCH%-%DESKTOP%.qcow2
```

may be expanded to this job variable:

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

## Variable precedence

It's possible to define the same variable in multiple places that would all be used for a single job - for instance, you may have a variable defined in both a test suite and a product that appear in the same job template. The precedence order for variables is as follows (from lowest to highest):

- Product
- Machine
- Test suite
- API POST query parameters

That is, variable values set as part of the API request that triggers the jobs will 'win' over values set at any of the other locations.

If you need to override this precedence - for example, you want the value set in one particular test suite to take precedence over a setting of the same value from the API request - you can add a leading + to the variable name. For instance, if you set +VARIABLE = foo in a test suite, and passed VARIABLE=bar in the API request, the test suite setting would 'win' and the value would be foo.

If the same variable is set with a + prefix in multiple places, the same precedence order described above will apply to those settings.

# 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://pagure.io/fedora-qa/os-autoinst-distri-fedora.git
./templates --clean
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 [createhdds](#) 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. However, openQA also supports a remote-download feature of assets from trusted domains.

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
```

More details on triggering tests can also be found in the [Users Guide](#).

# 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](#).

# Repositories and installation

Keep in mind that there can be disruptive changes between openQA versions. You need to be sure that the webui and the worker that you are using have the same version number or, at least, are compatible.

For example, the package distributed with openSUSE Leap 42.3 is not compatible with the version on Tumbleweed. And the package distributed with Tumbleweed may not be compatible with the version in the development package.

## Official repositories

The easiest way to install openQA is from distribution packages.

- For openSUSE, packages are available for Leap 42.3 and later.
- For Fedora, packages are available in the official repositories for Fedora 23 and later.

## Development version repository

You can find the development version of openQA in OBS in the [openQA:devel](#) repository.

To add the development repository to your system, you can use these commands.

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

LEAP_VERSION=15.0
zypper ar -f obs://devel:openQA/openSUSE_Leap_${LEAP_VERSION} devel-openQA
zypper ar -f obs://devel:openQA:Leap:${LEAP_VERSION}/openSUSE_Leap_${LEAP_VERSION} devel-
openQA-perl-modules
```

As required change LEAP\_VERSION to the version of openSUSE Leap you have installed.

## Installation

You can install the packages using these commands.

```
# openSUSE Leap 42.3+
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', 'proxy\_wstunnel' and 'rewrite' modules using the command 'a2enmod'. This is not necessary on Fedora.

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

For a basic setup, you can copy **openqa.conf.template** to **openqa.conf** and modify the ServerName if required setting. This will direct all HTTP traffic to openQA.

```
cp /etc/apache2/vhosts.d/openqa.conf.template /etc/apache2/vhosts.d/openqa.conf
```

## 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). On openSUSE, you should also add **SSL** to the **APACHE\_SERVER\_FLAGS** so it looks like this in **/etc/sysconfig/apache2**:

```
APACHE_SERVER_FLAGS="SSL"
```

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
```

# Database

Since version 4.5.1512500474.437cc1c7 of openQA, PostgreSQL is used as the database.

To configure access to the database in openQA, edit `/etc/openqa/database.ini` and change the settings in the `[production]` section.

The dsn value format technically depends on the database type and is documented for PostgreSQL at [DBD::Pg](#)

## 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
```

For older versions of openQA, you can migrate from SQLite to PostgreSQL according to [DB migration from SQLite to PostgreSQL](#)

# 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.

# Run the web UI

```
systemctl start postgresql
systemctl start openqa-gru
systemctl start openqa-webui
systemctl start openqa-scheduler
# 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 postgresql
systemctl enable openqa-gru
systemctl enable openqa-webui
systemctl enable openqa-scheduler
```

# 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 = 1234567890ABCDEF
secret = 1234567890ABCDEF
```

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.

```
install -d -m 0755 -o _openqa-worker /var/lib/openqa/pool/X
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

## 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.

## Referer settings to auto-mark important jobs

Automatic cleanup of old results (see GRU jobs) can sometimes render important tests useless. For example bug report with link to openQA job which no longer exists. Job can be manually marked as important to prevent quick cleanup or referer can be set so when job is accessed from particular web page (for example bugzilla), this job is automatically labeled as linked and treated as important.

List of recognized referers is space separated list configured in `/etc/openqa/openqa.ini`:

```
[global]
recognized_referers = bugzilla.suse.com bugzilla.opensuse.org
```

## 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
```

## Configuring AMQP message emission

You can configure openQA to send events (new comments, tests finished, ...) to an AMQP message bus. The messages consist of a topic and a body. The body contains json encoded info about the event. See [amqp\\_infra.md](#) for more info about the server and the message topic format. There you will find instructions how to configure the AMQP server as well.

To let openQA send messages to an AMQP message bus, first make sure that the perl-Mojo-RabbitMQ-Client RPM is installed. Then you will need to configure amqp in `/etc/openqa/openqa.ini`:



```
# Configuration for AMQP plugin
[amqp]
heartbeat_timeout = 60
reconnect_timeout = 5
# guest/guest is the default anonymous user/pass for RabbitMQ
url = amqp://guest:guest@localhost:5672/
exchange = pubsub
topic_prefix = suse
```

For a TLS connection use amqps:// and port 5671.

## Configuring worker to use more than one openQA server

When there are multiple openQA web interfaces (openQA instances) available a worker can be configured to register and accept jobs from all of them.

Requirements:

- /etc/openqa/client.conf must contain API keys and secrets to all instances
- Shared storage from all instances must be properly mounted

In the /etc/openqa/workers.ini enter space-separated instance hosts and optionally configure where the shared storage is mounted. Example:

```
[global]
HOSTS = openqa.opensuse.org openqa.fedora.fedoraproject.org

[openqa.opensuse.org]
SHARE_DIRECTORY = /var/lib/openqa/opensuse

[openqa.fedoraproject.org]
SHARE_DIRECTORY = /var/lib/openqa/fedora
```

Configuring SHARE\_DIRECTORY is not a hard requirement. Worker will try following directories prior registering with openQA instance:

1. SHARE\_DIRECTORY
2. /var/lib/openqa/\$instance\_host
3. /var/lib/openqa/share
4. /var/lib/openqa
5. fail if none of above is available

Once worker registers to openQA instance it checks for available job and starts accepting websockets commands. Worker accepts jobs as they will come in, there is no priority, or other

ordering, support at the moment. It is possible to mix local openQA instance with remote instances or use only remote instances.

## Asset Caching

If your network is slow or you experience long time to load needles you might want to consider to enable caching in your remote workers. To enable caching, `/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]
HOST=http://webui
CACHEDIRECTORY = $cache_location
CACHELIMIT = 50 # This is currently noop. Cache supports limiting, but is not enabled.

[http://webui]
TESTPOOLSERVER = rsync://yourlocation/tests
```

Setup and run rsync server daemon on HOST machine, in `/etc/rsyncd.conf` should be:

```
gid = users
read only = true
use chroot = true
transfer logging = true
log format = %h %o %f %l %b
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
slp refresh = 300
use slp = false

#[Example]
# path = /home/Example
# comment = An Example
# auth users = user
# secrets file = /etc/rsyncd.secrets

[tests]
path = /var/lib/openqa/share/tests
comment = OpenQA Test Distributions
```

and

```
systemctl start rsyncd.service
systemctl enable rsyncd.service
```

This will allow the workers to download the assets from the webUI and use them locally. If TESTPOOLSERVER is set tests and needles will also be cached by the worker.

# 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 database lockfile
- `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.

# 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](#).



All Tests

Job Groups ▾





# Test result overview

Overall Summary of **opensuse**, **opensuse test** build 0091

Passed: 2 Failed: 0 Scheduled: 1 Running: 2 None: 1

Filter no filter

## Flavor: DVD

Test	i586	x86_64
RAID0		-
kde	 kate shutdown +1	
textmode	 zypper_up	-

## Flavor: GNOME-Live

Test	i686
RAID0	

## Flavor: NET

Test	x86_64
kde	

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.

Specifying multiple groups with no build will yield the latest build of the first group. This can be useful to have a static URL for bookmarking.

## Description of test suites

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

Name	Settings	Description	Actions
textmode	DESKTOP=textmode VIDEOMODE=text	foo2	  
kde	DESKTOP=kde PATTERNS=gnome,base,enhanced_base,apparmor,yast2_basis,sw_management,multimedia, office,fonts,x11,imaging,games,non_oss,xen_server		
uEFI	DESKTOP=kde		

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.

## Flavor: DVD



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

## Review badges

Based on comments in the individual job results for each build a certificate icon is shown on the group overview page as well as the index page to indicate that every failure has been reviewed, e.g. a bug reference or a test issue reason is stated:

## openSUSE Leap 42.2 Updates

- [Build20170117-3](#) (about 2 hours ago) 
- [Build20170117-2](#) (about 7 hours ago) 
- [Build20170117-1](#) (about 14 hours ago) 

## openSUSE Argon

- [Build1.4](#) (7 days ago) 

## openSUSE Krypton

- [Build6.81](#) (7 days ago) 
- [Build6.80](#) (7 days ago) 

### Meaning of the different colors

- The green icons shows up when there is no work to be done.
- No icon is shown if at least one failure still need to be reviewed.
- The black icon is shown if all review work has been done.

(To simplify, checking for false-negatives is not considered here.)

## Show bug or label icon on overview if labeled [gh#550](#)

- Show bug icon with URL if mentioned in test comments
- Show bug or label icon on overview if labeled

For bugreferences write `<bugtracker_shortname>#<bug_nr>` in a comment, e.g. "bsc#1234", for generic labels use `label:<keyword>` where `<keyword>` can be any valid character up to the next whitespace, e.g. "false\_positive". The keywords are not defined within openQA itself. A valid list of keywords should be decided upon within each project or environment of one openQA instance.

Test	x86_64
awesome	  
gnome	   false_positive  center

Figure 5. Example for a generic label



Figure 6. Example for bug label

Related issue: [#10212](#)

'Hint:' You can also write (or copy-paste) full links to bugs and issues. The links are automatically changed to the shortlinks (e.g. <https://progress.opensuse.org/issues/11110> turns into [poo#11110](#)). Related issue: [poo#11110](#)

Also github pull requests and issues can be linked using the generic format ``<marker>[#<project/repo>]<id>``, e.g. [gh#os-autoinst/openQA#1234](#), see [gh#973](#)

All issue references are stored within the internal database of openQA. The status can be updated using the `/bugs` API route for example using external tools.

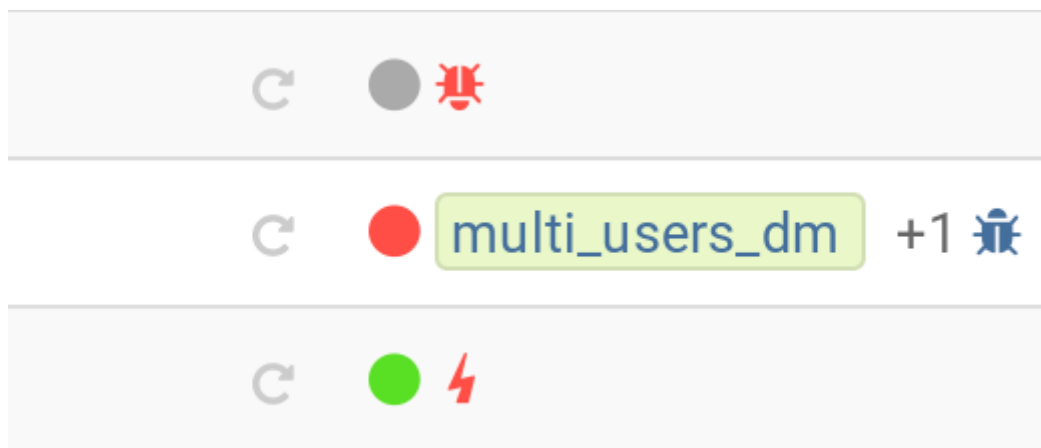


Figure 7. Example for visualization of closed issue references. Upside down icons in red visualize closed issues.

## Build tagging

### Tag builds with special comments on group overview

Based on comments on the group overview individual builds can be tagged. As 'build' by themselves do not own any data the job group is used to store this information. A tag has a build to link it to a build. It also has a type and an optional description. The type can later on be used to distinguish tag types.

The generic format for tags is

```
tag:<build_id>:<type>[:<description>], e.g. tag:1234:important:Beta1.
```

The more recent tag always wins.

A 'tag' icon is shown next to tagged builds together with the description on the `group_overview` page. The index page does not show tags by default to prevent a potential performance regression. Tags can be enabled on the index page using the corresponding option in the filter form at the bottom of the page.

Build0091 (less than a minute ago)



Build0048 (less than a minute ago)  GM



## Comments



Demo wrote less than a minute ago  
tag:0048:important:GM

## Keeping important builds

As builds can now be tagged we come up with the convention that the 'important' type - the only one for now - is used to tag every job that corresponds to a build as 'important' and keep the logs for these jobs longer so that we can always refer to the attached data, e.g. for milestone builds, final releases, jobs for which long-lasting bug reports exist, etc.

## Filtering test results and builds

At the top of the test results overview page is a form which allows filtering tests by result, architecture and TODO-status.

Filter

Result

☐ Passed ☐ Incomplete ☒ Softfailed ☐ Failed

Architecture

x86\_64

Misc

☐ TODO






















































APPLY



There is also a similar form at the bottom of the index page which allows filtering builds by group and customizing the limits.

## 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.

	 RAID 10@aarch64 	 
	 qam-regression-firefox@64bit 	23  2  2  
	 qam-regression-message@64bit 	12  1  
	 qam-regression-documentation@64bit 	22  
	 qam-regression-gnome@64bit 	10  
	 qam-regression-documentation@64bit 	22  1  
	 qam-regression-firefox@64bit 	26  
	 qam-regression-other@64bit 	16  
	 qam-regression-installation@64bit 	21 
	 mau-qa_openssl@64bit 	3  

5 Chained children

# Asset cleanup

Assets like ISO files consume a huge amount of disk space. Therefore openQA removes assets automatically according to configurable limits.

This section provides an overall description of the cleanup strategy and how to configure the limits. Cleanup-related parameter for the REST API can be found in the 'Asset handling' section under 'Use of the REST API'.

## Cleanup strategy

openQA frequently checks whether assets need to be removed according to the configured limits.

To find out whether an asset should be removed, openQA determines by which job groups the asset is used. If at least one job within a certain job group is using an asset, the asset is considered to be used by that group.

So an asset can be accounted to multiple groups. The assets table which is accessible via the admin menu shows these groups for each asset and also the latest job.

If the size limit for assets by a certain group is exceeded, openQA will remove assets accounted to that group:

- Assets belonging to old jobs are preferred.
- Assets belonging to jobs which are still scheduled or running are not considered.
- Assets which are also accounted to another group that has still space left are not considered.

Assets which do *not* belong to any group are removed after a configurable duration. Keep in mind that this behavior is also enabled on local instances and affects all cloned jobs (unless cloned into a job group).

## Configure limit for assets within groups

To configure the maximum size for the assets of a group, open 'Job groups' in the operators menu and select a group. The size limit for assets can be configured under 'Edit job group properties'. It also shows the size of assets which belong to that group and not to any other group.

Job groups inherit the size limit from their parent group unless the limit is set explicitly. The default size limit for groups can be adjusted in the `default_group_limits` section of the openQA config file.

## Configure limit for groupless assets

Assets not belonging to jobs within a group are deleted automatically after a certain number of days. That duration can be adjusted by setting `untracked_assets_storage_duration` in the `misc_limits` section of the openQA config to the desired number of days.

# 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`).

Basics are described in the [Getting Started](#) guide.

## Triggering tests

Tests can be triggered over multiple ways, using `clone_job.pl`, `jobs post`, `isos post` as well as retriggering existing jobs or whole media over the web UI.

### Cloning existing jobs - clone\_job.pl

If one wants to recreate an existing job from any publically available openQA instance the script `clone_job.pl` can be used to copy the necessary settings and assets to another instance and schedule the test. For the test to be executed it has to be ensured that matching ressources can be found, for example a worker with matching `WORKER_CLASS` must be registered. More details on `clone_job.pl` can be found in [Writing Tests](#).

### Spawning single new jobs - jobs post

Single jobs can be spawned using the `jobs post` API route. All necessary settings on a job must be supplied in the API request. The "openQA client" has examples for this.

### Spawning multiple jobs based on templates - isos post

The most common way of spawning jobs on production instances is using the `isos post` API route. Based on previously defined settings for media, job groups, machines and test suites jobs are triggered based on template matching. The [Getting Started](#) guide already mentioned examples. Additionally to the necessary template matching parameters more parameters can be specified which are forwarded to all triggered jobs. There are also special parameters which only have an influence on the way the triggering itself is done. These parameters all start with a leading underscore but are set as request parameters in the same way as the other parameters.

*The following scheduling parameters exist*

#### **`_NO_OBSOLETE`**

Do not obsolete jobs in older builds with same DISTRI and VERSION (as is the default behavior). With this option jobs which are currently pending, for example scheduled or running, are not cancelled when a new medium is triggered.

#### **`_DEPRIORITIZEBUILD`**

Setting this switch '1' will not immediately obsolete jobs of old builds but rather deprioritize them up to a configurable limit of priority.

#### **`_DEPRIORITIZE_LIMIT`**

The configurable limit of priority up to which jobs should be deprioritized. Needs `_DEPRIORITIZEBUILD`. Default 100.

### `_ONLY_OBSOLETE_SAME_BUILD`

Only obsolete (or deprioritize) jobs for the same BUILD. This is useful for cases where a new build appearing does not necessarily mean existing jobs for earlier builds with the same DISTRI and VERSION are no longer interesting, but you still want to be able to re-submit jobs for a build and have existing jobs for the exact same build obsoleted.

### `_GROUP`

Job templates **not** matching the given group name are ignored. Does **not** affect obsolescence behavior, so you might want to combine with `_NO_OBSOLETE`.

### `_GROUP_ID`

Same as `_GROUP` but allows to specify the group directly by ID.

### `_PRIORITY`

Sets the priority for the new jobs (which otherwise defaults to the priority of the job template)

Example for `_DEPRIORITIZEBUILD` and `_DEPRIORITIZE_LIMIT`.

```
openqa-client isos post ISO=my_iso.iso DISTRI=my_distri FLAVOR=sweet \
  ARCH=my_arch VERSION=42 BUILD=1234 \
  _DEPRIORITIZEBUILD=1 _DEPRIORITIZE_LIMIT=120 \
```

## Asset handling

Multiple parameters exist to reference assets to be used by tests, for example `ISO`, `HDD`, `ASSET`, `KERNEL`, `INITRD`. Corresponding files must be provided within the path `/var/lib/openqa/share/factory` local to the openQA web-UI. All assets specified in this way in any job are tracked by openQA and considered by the automatic cleanup described under 'Asset cleanup'.

Multiple options can be used based on special suffix types. These types can also be combined, for example `ISO_1_DECOMPRESS_URL`.

*The following options exist*

### `_<NR>`

To specify multiple assets of the same type use a number digit, for example `ISO_1`, `ISO_2`.

### `_URL`

Before starting these jobs try to download these assets into the asset directory of the openQA web-UI from trusted domains specified in `/etc/openqa/openqa.ini`.

### `_DECOMPRESS_URL`

Specify a compressed asset to be downloaded that will be uncompressed by openQA. Specify the non-suffixed parameter additionally to provide a rename target, for example `ISO_1_DECOMPRESS_URL=http://host/foo2.iso.xz` and `ISO_1=foo.iso`

# 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 should happen when test execution of the current test module is finished depending on the result. If we should skip execution of the following test modules if current one failed, or it should be used to create a snapshot of SUT to rollback to. See example below.

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 {
    # 'fatal'          - abort whole test suite if this fails (and set overall state
    # 'failed')
    # 'ignore_failure' - if this module fails, it will not affect the overall result
    # at all
    # 'milestone'      - after this test succeeds, update 'lastgood'
    # 'no_rollback'    - don't roll back to 'lastgood' snapshot if this fails
    # 'always_rollback' - roll back to 'lastgood' snapshot even if test was
    # successful (supported on QEMU backend only)
    return { fatal => 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.

# Advanced test features

## Capturing kernel exceptions and/or any other exceptions from the serial console

Soft and hard failures can be triggered on demand by regular expressions when they match the serial output which is done after the test is executed. In case it doesn't make sense to continue test run even if current test module doesn't have fatal flag, use `fatal` as serial failure type, so all subsequent test modules won't be executed if such failure was detected. To use this functionality the test developer needs to define the patterns to look for in the serial output either in the `main.pm` or in the test itself. Any pattern change done in a test it will be reflected in the next tests.

The patterns defined in the `main.pm` will be valid for all the tests.

To simplify tests results review, if job fails with the same message, which is defined for the pattern, as previous job, automatic comment carryover will work even if test suites have failed due to different test modules.

*Example: Defining serial exception capture in the main.pm*

```
$testapi::distri->set_expected_serial_failures([
    {type => 'soft', message => 'known issue', pattern => quotemeta 'Error'},
    {type => 'hard', message => 'broken build', pattern => qr/exception/},
    {type => 'fatal', message => 'critical issue build', pattern => qr/kernel
oops/},
]);
```

*Example: Defining serial exception capture in the test*

```
sub run {
    my ($self) = @_;
    $self->{serial_failures} = [
        {type => 'soft', message => 'known issue', pattern => quotemeta 'Error'},
        {type => 'hard', message => 'broken build', pattern => qr/exception/},
        {type => 'fatal', message => 'critical issue build', pattern => qr/kernel
oops/},
    ];
    ...
}
```

Example: Adding serial exception capture in the test

```
sub run {
  my ($self) = @_;
  push @$self->{serial_failures}, {type => 'soft', message => 'known issue',
  pattern => quotemeta 'Error'};
  ...
}
```

## 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.

## Inter-machine dependencies

Those dependencies make it possible to create job dependencies between tests which are scheduled for the different machines. To use it, simply append the machine name for each dependent test suite with colon separated. If machine is not explicitly defined, the variable `%MACHINE%` of the current job is used for the dependent test suite. Eg. `START_AFTER_TEST="kde:64bit-1G,dhcp-server:64bit-8G"` or `PARALLEL_WITH="web-server:ipmi-fly,dhcp-server:ipmi-bee",http-server` Then, in job templates, add test suite(s) and all of its dependent test suite(s). Keep in mind to place the machines which have been explicitly defined in a variable for each dependent test suite. Checkout the example sections to get a better understanding: ===== Example: correct dependency and machine placed

There is a test suite A placed on machine 64bit-8G,  
then test suite B placed on machine 64bit-1G with variable `START_AFTER_TEST=A:64bit-8G`

This results in the following dependency:

```
A:64bit-8G <- B:64bit-1G
```

**Example: best match with correct dependency and machine placed**

If test suite A placed on both of machines 64bit and ppc  
then test suite B placed on same machines 64bit and ppc with variable  
START\_AFTER\_TEST=A  
openQA take the best matches.

This results in the following dependency:

```
A:64bit <- B:64bit  
A:ppc <- B:ppc
```

**Example: wrong dependency or wrong machine placed**

If test suite A placed on machine 64bit-8G,  
then test suite B placed on machine 64bit-1G with variable START\_AFTER\_TEST=A  
openQA won't create job dependency and give an error message unless you have been  
explicitly define variable as START\_AFTER\_TEST=A:64bit-8G  
If test suite A placed on machine ppc, 64bit and s390x  
then there are 3 testsuites B on ppc-1G, C on ppc-2G, D on ppc64le with same variable  
PARALLEL\_WITH=A:ppc

This results in the following dependency:

```
      A:ppc  
      ^  
     / | \  
    /  | \  
 B:ppc-1G C:ppc-2G D:ppc64le
```

openQA will also show errors to inform users that test suite A on machines 64bit and s390x are not necessary. ===== Example: dependency on same machine

If variable START\_AFTER\_TEST or PARALLEL\_WITH \*only\* with name(s) of test suite(s).  
START\_AFTER\_TEST=A,B or PARALLEL\_WITH=A,B

openQA will create job dependencies that are scheduled on the same machine if all test suites are placed on 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 functions: `mutex_create`, `mutex_lock`, `mutex_unlock`, `mutex_wait`. Each of these functions take at least one parameter: name of the lock. Note that lock name can't contain "-" character. 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_wait` is combination of `mutex_lock` & `mutex_unlock` that displays more information about mutex state (time spent waiting, location of lock). Use this if you wait for specific action from single place (apache is running on master node)

`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.

```

use lockapi;
use mmapi;

# On parent job
sub run {
    # ftp service started automatically on boot
    assert_screen 'login', 300;

    # unlock by creating the lock
    mutex_create 'ftp_service_ready';

    # wait until all children finish
    wait_for_children;
}

# On child we wait for ftp server to be ready
sub run {
    # wait until ftp service is ready
    # performs mutex lock & unlock internally
    mutex_wait 'ftp_service_ready';

    # connect to ftp and start downloading
    script_run 'ftp parent.job.ip';
    script_run 'get random_file';
}

# Mutexes can be used also for granting exclusive access to resource
# Example on child when only one job should access ftp at time
sub run {
    # wait until ftp service is ready
    mutex_lock 'ftp_service_ready';

    # Perform operation with exclusive access
    script_run 'ftp parent.job.ip';
    script_run 'put only_i_am_here';
    script_run 'bye';

    # Allow other jobs to connect afterwards
    mutex_unlock 'ftp_service_ready';
}

```

Sometimes it is useful to wait for certain action from child or sibling job, not parent. In this case child or sibling will create a mutex and any cluster job can lock/unlock it.

The child can however die at any time. To prevent parent deadlock in this situation, it's required to pass mutex owner job ID as a second parameter to `mutex_lock` and `mutex_wait`. Mutex owner is the job that creates the mutex. If a child job with given ID already finished, `mutex_lock()` calls die. Job ID is also required when unlocking such mutex.

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

```
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_wait('child_reached_given_point', $child_id);

    # continue with the test
}
```

Mutexes are a way to wait for specific event from single job. When we need multiple jobs to reach required state we need to use barriers.

Before first use barrier needs to be created with `barrier_create` with 2 parameters - name and count. Name behaves as ID (same as with mutexes), count is number of jobs needed to call `barrier_wait` to unlock barrier.

There is optional `barrier_wait` parameter called `check_dead_job`. When used it will kill all jobs waiting in `barrier_wait` if one of cluster jobs die.

It prevents waiting for state that will never be reached (and eventually die on job timeout). Should be set only on one of `barrier_wait` calls.

Example is situation with 1 master and 3 worker jobs. We need to wait until 3 worker jobs perform initial setup. After that we can make cluster from them, but if one of them fails it makes no sense waiting.

```
use lockapi;

# In main.pm
barrier_create('NODES_CONFIGURED', 4);

# On master job
sub run {
    assert_screen 'login', 300;

    # Master is ready, waiting while workers are configured (check_dead_job is
    optional)
    barrier_wait {name => "NODES_CONFIGURED", check_dead_job => 1};

    # When 4 jobs called barrier_wait they are all unblocked
    script_run 'create_cluster';
    script_run 'test_cluster';

    # Notify all nodes we are finished
    mutex_create 'CLUSTER_CREATED';
    wait_for_children;
}

# On 3 worker jobs
sub run {
    assert_screen 'login', 300;

    # do initial worker setup
    script_run 'zypper in HA';
    script_run 'echo IP > /etc/HA/node_setup';

    # Join the group of jobs waiting for each other
    barrier_wait 'NODES_CONFIGURED';

    # Don't finish until cluster is created & tested
    mutex_wait 'CLUSTER_CREATED';
}
```

Getting information about parents and 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
    #   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 downloading 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 in many situations you will need to enter commands into a textual shell (e.g Bash), TTY, text terminal, command prompt, TUI etc.

OpenQA has two basic methods for interacting with a text shell. The first uses the same input and output methods as when interacting with a GUI, plus a serial port for getting raw text output from the SUT. This is primarily implemented with VNC and so I will refer to it as the VNC text console.

The serial port device which is used with the VNC text console is the default virtual serial port device in QEMU (i.e. the device configured with the `-serial` command line option). I will refer to this as the "default serial port". OpenQA currently only uses this serial port for one way communication from the SUT to the host.

The second method uses another serial port for both input and output. The SUT attaches a TTY to the serial port which `os-autoinst` logs into. All communication is therefore text based, similar to if you SSH'd into a remote machine. This is called the serial terminal console (or the virtio console, see implementation section for details).

The VNC text console is very slow and expensive relative to the serial terminal console, but allows you to continue using `assert_screen` and is more widely supported. Below is an example of how to use the VNC text console.

*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 (if necessary). Now that we are on a text console it is possible to run scripts and observe their output either as raw text or on the video feed.

Note that `root-console` is defined by the distribution, so on different distributions or operating systems this can vary. There are also many utility functions that wrap `select_console`, so check your distribution's utility library before using it directly.

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

This returns the contents of the SUT's `/proc/cpuinfo` file to the test script and then searches it for the term 'avx2' using a regex.

The `script_run` and `script_output` 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.

The command `wait_serial` watches the SUT's serial port for text output and matches it against a regex. `type_string` sends a string to the SUT like it was typed in by the user over VNC.

## 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 default 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 shell commands and received program output.

Communicating entirely using text also means that you no longer have to worry about your needles being invalidated due to a font change or similar. It is also much cheaper to transfer text and test it against regular expressions than encode images from a VNC feed and test them against sample images (needles).

On the other hand you can no longer use `assert_screen` or take a screen shot because the text is never rendered as an image. A lot of programs will also send ANSI escape sequences which will appear as raw text to the test script instead of being interpreted by a terminal emulator which then renders the text.

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

The above code will cause `type_string` and `wait_serial` to write and read from a virtio serial port. A

distribution specific call back will be made which allows os-autoinst to log into a serial terminal session running on the SUT. Once `select_console` returns you should be logged into a TTY as root.

If you are struggling to visualise what is happening, imagine SSH-ing into a remote machine as root, you can then type in commands and read the results as if you were sat at that computer. What we are doing is much simpler than using an SSH connection (it is more like using GNU screen with a serial port), but the end result looks quite similar.

As mentioned above, changing input and output to a serial terminal has the effect of changing where `wait_serial` reads output from. On a QEMU VM `wait_serial` usually reads from the default serial port which is also where the kernel log is usually output to.

When switching to a virtio based serial terminal, `wait_serial` will then read from a virtio serial port instead. However the default serial port still exists and can receive output. Some utility library functions are hard coded to redirect output to the default serial port and expect that `wait_serial` will be able to read it. Usually it is not too difficult to fix the utility function, you just need to remove some redirection from the relevant shell command.

Another common problem is that some library or utility function tries to take a screen shot. The hard part is finding what takes the screen shot, but then it is just a simple case of checking `is_serial_terminal` and not taking the screen shot if we are on a serial terminal console.

Distributions usually wrap `select_console`, so instead of using it directly, you can use something like the following which is from the OpenSUSE test suite.

```
if (select_virtio_console()) {  
    # Do something which only works, or is necessary, on a serial terminal  
}
```

This selects the virtio based serial terminal console if possible. If it is available then it returns true. It is also possible to check if the current console is a serial terminal by calling `is_serial_terminal`.

Once you have selected a serial terminal, the video feed will disappear from the live view, however at the bottom of the live screen there is a separate text feed. After the test has finished you can view the serial log(s) in the assets tab. You will probably have two serial logs; `serial0.txt` which is written from the default serial port and `serial_terminal.txt`.

Now that you are on a serial terminal console everything will start to go a lot faster. So much faster in fact that race conditions become a big issue. Generally these can be avoided by using the higher level functions such as `script_run` and `script_output`.

It is rarely necessary to use the lower level functions, however it helps to recognise problems caused by race conditions at the lower level, so please read the following section regardless.

So if you do need to use `type_string` and `wait_serial` directly then try to use the following pattern:

1) Wait for the terminal prompt to appear. 2) Send your command 3) Wait for your command text to be echoed by the shell (if applicable) 4) Send enter 5) Wait for your command output (if applicable)

To illustrate this is a snippet from the LTP test runner which uses the lower level commands to achieve a little bit more control. I have numbered the lines which correspond to the steps above.

```
my $fin_msg      = "### TEST $test->{name} COMPLETE >>> ";
my $cmd_text     = qq($test->{command}; echo "$fin_msg\$?");
my $klog_stamp   = "echo 'OpenQA::run_ltp.pm: Starting $test->{name}' >
/dev/$serialdev";

# More variables and other stuff

if (is_serial_terminal) {
    script_run($klog_stamp);
    wait_serial(serial_term_prompt(), undef, 0, no_regex => 1); #Step 1
    type_string($cmd_text);                                     #Step 2
    wait_serial($cmd_text, undef, 0, no_regex => 1);           #Step 3
    type_string("\n");                                         #Step 4
} else {
    # None serial terminal console code (e.g. the VNC console)
}
my $test_log = wait_serial(qr/$fin_msg\d+/, $timeout, 0, record_output => 1); #Step 5
```

The first `wait_serial` (Step 1) ensures that the shell prompt has appeared. If we do not wait for the shell prompt then it is possible that we can send input to whatever command was run before. In this case that command would be 'echo' which is used by `script_run` to print a 'finished' message.

It is possible that echo was able to print the finish message, but was then suspended by the OS before it could exit. In which case the test script is able to race ahead and start sending input to echo which was intended for the shell. Waiting for the shell prompt stops this from happening.

INFO: It appears that echo does not read STDIN in this case, and so the input will stay inside STDIN's buffer and be read by the shell (Bash). Unfortunately this results in the input being displayed twice: once by the terminal's echo (explained later) and once by Bash. Depending on your configuration the behavior could be completely different

The function `serial_term_prompt` is a distribution specific function which returns the characters previously set as the shell prompt (e.g. `export PS1="# "`, see the `bash(1)` or `dash(1)` man pages). If you are adapting a new distribution to use the serial terminal console, then we recommend setting a simple shell prompt and keeping track of it with utility functions.

The `no_regex` argument tells `wait_serial` to use simple string matching instead of regular expressions, see the implementation section for more details. The other arguments are the timeout (`undef` means we use the default) and a boolean which inverts the result of `wait_serial`. These are explained in the `os-autoinst/testapi.pm` documentation.

Then the test script enters our command with `type_string` (Step 2) and waits for the command's text to be echoed back by the system under test. Terminals usually echo back the characters sent to them so that the user can see what they have typed.

However this can be disabled (see the `stty(1)` man page) or possibly even unimplemented on your

terminal. So this step may not be applicable, but it provides some error checking so you should think carefully before disabling echo deliberately.

We then consume the echo text (Step 3) before sending enter, to both check that the correct text was received and also to separate it from the command output. It also ensures that the text has been fully processed before sending the newline character which will cause the shell to change state.

It is worth reminding oneself that we are sending and receiving data extremely quickly on an interface usually limited by human typing speed. So any string which results in a significant state change should be treated as a potential source of race conditions.

Finally we send the newline character and wait for our custom finish message. `record_output` is set to ensure all the output from the SUT is saved (see the next section for more info).

What we do **not** do at this point, is wait for the shell prompt to appear. That would consume the prompt character breaking the next call to `script_run`.

We choose to wait for the prompt just before sending a command, rather than after it, so that Step 5 can be deferred to a later time. In theory this allows the test script to perform some other work while the SUT is busy.

## Sending new lines and continuation characters

The following command will timeout: `script_run("echo \"1\n2\"")`. The reason being `script_run` will call `wait_serial("echo \"1\n2\"")` to check that the command was entered successfully and echoed back (see above for explanation of serial terminal echo, note the echo shell command has not been executed yet). However the shell will translate the newline characters into a newline character plus '>', so we will get something similar to the following output.

```
echo "1
> 2"
```

The '>' is unexpected and will cause the match to fail. One way to fix this is simply to do `echo -e \"1\n2\"`. In this case Perl will not replace `\n` with a newline character, instead it will be passed to `echo` which will do the substitution instead (note the '-e' switch for `echo`).

In general you should be aware that, Perl, the guest kernel and the shell may transform whatever character sequence you enter. Transformations can be spotted by comparing the input string with what `wait_serial` actually finds.

## Sending signals - ctrl-c and ctrl-d

On a VNC based console you simply use `send_key` like follows.

```
send_key('ctrl-c');
```

This usually (see `termios(3)`) has the effect of sending SIGINT to whatever command is running. Most commands terminate upon receiving this signal (see `signal(7)`).



On a serial terminal console the `send_key` command is not implemented (see implementation section). So instead the following can be done to achieve the same effect.

```
type_string('', terminate_with => 'ETX');
```

The ETX ASCII code means End of Text and usually results in SIGINT being raised. In fact pressing ctrl-c may just be translated into ETX, so you might consider this a more direct method. Also you can use 'EOT' to do the same thing as pressing ctrl-d.

You also have the option of using Perl's control character escape sequences in the first argument to `type_string`. So you can also send ETX with:

```
type_string("\cC");
```

The `terminate_with` parameter just exists to display intention. It is also possible to send any character using the hex code like `\x0f` which may have the effect of pressing the magic SysRq key if you are lucky.

## The virtio serial terminal implementation

The `os-autoinst` package supports several types of 'consoles' of which the virtio serial terminal is one. The majority of code for this console is located in `consoles/virtio_terminal.pm` and `consoles/virtio_screen.pm`. However there is also related code in `backends/qemu.pm` and `distribution.pm`.

You may find it useful to read the documentation in `virtio_terminal.pm` and `virtio_screen.pm` if you need to perform some special action on a terminal such as triggering a signal or simulating the SysRq key. There are also some console specific arguments to `wait_serial` and `type_string` such as `record_output`.

The virtio 'screen' essentially reads data from a socket created by QEMU into a ring buffer and scans it after every read with a regular expression. The ring buffer is large enough to hold anything you are likely to want to match against, but not too large as to cause performance issues. Usually the contents of this ring buffer, up to the end of the match, are returned by `wait_serial`. This means earlier output will be overwritten once the ring buffer's length is exceeded. However you can pass `record_output` which saves the output to a separate unlimited buffer and returns that instead.

Like `record_output`, the `no_regex` argument is a console specific argument supported by the serial terminal console. It may or may not have some performance benefits, but more importantly it allows you to easily match arbitrary strings which may contain regex escape sequences. To be clear, `no_regex` hints that `wait_serial` should just treat its input as a plain string and use the Perl library function `index` to search for a match in the ring buffer.

The `send_key` function is not implemented for the serial terminal console because the OpenQA console implementation would need to map key actions like ctrl-c to a character and then send that character. This may mislead some people into thinking they are actually sending ctrl-c to the SUT and also requires OpenQA to choose what character ctrl-c represents which varies across terminal

configurations.

Very little of the code (perhaps none) is specific to a virtio based serial terminal and can be reused with a physical serial port, SSH socket, IPMI or some other text based interface. It is called the virtio console because the current implementation just uses a virtio serial device in QEMU (and it could easily be converted to an emulated port), but it otherwise has nothing to do with the virtio standard and so you should avoid using the name 'virtio console' unless specifically referring to the QEMU virtio implementation.

As mentioned previously, ANSI escape sequences can be a pain. So we try to avoid them by informing the shell that it is running on a 'dumb' terminal (see the SUSE distribution's serial terminal utility library). However some programs ignore this, but piping there output into tee is usually enough to stop them outputting non-printable characters.

# 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 F00=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, 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
```

## Backend variables for faster test execution

The `os-autoinst` backend offers multiple test variables which are helpful for test development. For example:

- Set `_EXIT_AFTER_SCHEDULE=1` if you only want to evaluate the test schedule before the test modules are executed
- Use `_SKIP_POST_FAIL_HOOKS=1` to prevent lengthy `post_fail_hook` execution in case of expected and known test fails, for examples when you need to create needles anyway

## Using snapshots to speed up development of tests

For lower turn-around times during test development based on virtual machines the `QEMU` backend provides a feature that allows a job to start from a snapshot which can 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 the 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 as more than just

the snapshot of the last failed module is saved.

- Create a snapshot **after every successful** test module while **always overwriting** the existing snapshot to preserve only the latest (TESTDEBUG)
  - Allows to skip just before the start of the first failed test module, which can be limiting, but preserves disk space in comparison to MAKETESTSNAPSHOTS.
  - This mode is useful for iterative test development

In both modes there is no need to modify tests (i.e. adding milestone test flag as the behaviour is implied). In the later mode every test module is also considered fatal. This means the job is aborted after the 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 successful 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. The snapshot is named `lastgood` in all cases.

```
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. Make sure the new job will also have `TESTDEBUG=1` set. This can be ensured by the use of the `clone_job` script on the clone source job or specifying the variable explicitly:

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

# **openQA test harness result processing**

# Introduction

From time to time, a test developer might want to use openQA to execute a test suite from a different test harness than openQA, but still use openQA to setup test scenarios and prepare the environment for a test suite run; for this case openQA has the ability to process logs from external harnesses, and display the results integrated within the job results of the webUI.

One could say that a Test Harness is supported if its output is compatible with the available {parser-format}, such as LTP, and also xUnit or JUnit, but this can be easily extended to include more formats, such as RSpec or TAP.

The requirements to use this functionality, are quite simple:

- The test harness must produce a compatible format with supported {parser-format}.
- The test results can be uploaded via `testapi::parse_extra_log` within an openQA tests.
- The test results can also be uploaded via web [Web Api endpoint](#).

openQA will store these results in its own internal format for easier presentation, but still will allow the original file to be downloaded.

# Usage

If a test developer wishes to use the functional interface, after finishing the execution of the the testing too, calling `testapi::parse_extra_log` with the location to a the file generated.

## openQA test distribution

From within a common openQA test distribution, a developer can use `parse_extra_log` to upload a text file that contains a supported test output:

```
script_run('prove --verbose --formatter=TAP::Formatter::JUnit t/28-logging.t > junit-logging.xml');  
parse_extra_log('JUnit', 'junit-logging.xml');
```

# Available parser formats

Current parser formats:

- `OpenQA::Parser::Format::TAP`,
- `OpenQA::Parser::Format::JUnit`
- `OpenQA::Parser::Format::LTP`
- `OpenQA::Parser::Format::XUnit`,



# Extending the parser

## OOP Interface

The parser is a base class that acts as a serializer/deserializer for the elements inside of it, it allows to be extended so new formats can be easily added.

The base class is exposing 4 Mojo::Collections available, according to what openQA would require to map the results correctly, 1 extra collection is provided for arbitrary data that can be exposed. The collections represents respectively: test results, test definition and test output.

## Structured data

In structured data mode, elements of the collections are objects. They can be of any type, even though subclassing or objects of type of OpenQA::Parser::Result are preferred.

One thing to keep in mind, is that in case deeply nested objects need to be parsed like hash of hashes, array of hashes, they would need to subclass OpenQA::Parser::Result or OpenQA::Parser::Results respectively.

As an example, JUnit format can be parsed this way:

```

use OpenQA::Parser::Format::JUnit;

my $parser_result = OpenQA::Parser::Format::JUnit->new->load("file.xml");

# Now we can access to parsed tests as seen by openQA:

$parser_result->tests->each(sub {

    my $test = shift;
    print "Test name: ".$test->name;

});

my @all = $parser_result->tests->each;
my @tests = $parser->tests->search(name => qr/1_running_upstream_tests/);
my $first = $parser->tests->search(name => qr/1_running_upstream_tests/)->first();

my $binary_data = $parser->serialize();

# Now, we can also store $binary_data and retrieve it later.

my $new_parser_from_binary = OpenQA::Parser::Format::JUnit->new->deserialize($binary_data);

# thus this works as expected:
$new_parser_from_binary->tests->each( sub {

    my $test = shift;
    print "Test name: ".$test->name;

});

# We can also serialize all to JSON

my $json_serialization = $parser->to_json;

# save it and access it later

my $from_json = OpenQA::Parser::Format::JUnit->from_json($json_serialization);

```

## openQA internal test result storage

It is important to know that openQA's internal mapping for test results works operating almost entirely on the filesystem, leaving only the test modules to be registered into the database, this leads to the following relation: A test module's name is used to create a file with details (details-\$testmodule.json), that will contain a reference to step details, which is a collection of references to files, using a field "text" as tie in, and expecting a filename.

# 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. However, when opening the needle editor, a warning about the new needle will be shown and it can be selected as base.
- 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.

# 403 messages when using scripts

- If you come across messages displaying ERROR: 403 - Forbidden, make sure that the correct API key is present in client.conf file.
- If you are using a hostname other than localhost, pass **--host foo** to the script.
- If you are using fake authentication method, and the message says also "api key expired" you can simply logout and log in again in the webUI and the expiration will be automatically updated

# 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+rx /var/lib/openqa/` and `chgrp -R openqa /var/lib/openqa` can not fix.

# Performance impact

openQA workers can cause high I/O load, especially when creating VM snapshots. The impact therefore gets more severe when MAKETESTSNAPSHOTS is enabled. should not impact the stability of openQA jobs but can increase job execution time. If you run jobs on a machine where responsiveness of other services matter, for example your desktop machine, consider patching the IOSchedulingPriority of a workers service file as described in the [systemd documentation](#), for example set IOSchedulingPriority=7 for the lowest priority. If not available then you can try to execute the worker processes with ionice to reduce the risk of your system becoming significantly impacted by snapshot creation. Loading VM snapshots can also have an impact on SUT behavior as the execution of the first step after loading a snapshot might be delayed. This can lead to problems if the executed tests do not foresee an appropriate timeout margin.

# DB migration from SQLite to postgresSQL

As a first step to start using postgresSQL, please, configure postgresSQL database according to the [postgresSQL setup guide](#)

To migrate api keys run following commands:

- Export data from the SQLite db:

```
sqlite3 db.sqlite -csv -separator ',' 'select * from api_keys;' > apikeys.csv
```

Note: SQLite database file is located in `/var/lib/openqa/db` by default.

- Import data to the postgresSQL

```
# openqa is the postgresSQL database name and apikeys.csv is api keys export file
psql -U postgres -d openqa -c "copy api_keys from 'apikeys.csv' with (format csv);"
```

In case you need to migrate job groups, test suites, use `dump_templates` and `load_templates` scripts accordingly.



# 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.

## 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 Setup

The section provides one of the ways for setting up openQA environment to run tests that require network connection between several machines (e.g. client — server tests).

The example of the configuration is applicable for openSUSE and will use *Open vSwitch* for virtual switch, *SuSEfirewall2* or *firewalld* for NAT and *wicked* as network manager.

**NOTE** Other way to setup the environment with *iptables* and *firewalld* is described on [Fedora wiki](#).

## Set Up Open vSwitch

Compared to VDE setup, Open vSwitch is a little bit more complicated to configure, but provides more robust and scalable network.

- Install and Run Open vSwitch:

```
zypper in openvswitch
systemctl start openvswitch.service
systemctl enable openvswitch.service
```

- Install and configure *os-autoinst-openvswitch.service*:

**NOTE** *os-autoinst-openvswitch.service* is a support service that 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.

```
zypper in os-autoinst-openvswitch
systemctl start os-autoinst-openvswitch
systemctl enable os-autoinst-openvswitch
```

*os-autoinst-openvswitch.service* uses *br0* bridge by default. As it might be used by KVM, configure *br1* instead.

```
# /etc/sysconfig/os-autoinst-openvswitch
OS_AUTOINST_USE_BRIDGE=br1
```

- Create virtual bridge *br1*:

```
ovs-vsctl add-br br1
```

## Configure Virtual Interfaces

- Add tap interface for every multi-machine worker:

## NOTE

Create as many interfaces as needed for a test. The instructions are provided for three interfaces *tap0*, *tap1*, *tap2* to be used by *worker@1*, *worker@2*, *worker@3* workers. TAP interfaces have to be owned by the *\_openqa-worker* user for openQA to be able to access them.

To create tap interfaces automatically on startup, add appropriate configuration files to the */etc/sysconfig/network/* directory. Files have to be named as *ifcfg-tap<N>*, replacing *<N>* with the number for the interface, such as *0*, *1*, *2* (e.g. *ifcfg-tap0*, *ifcfg-tap1*).

```
# /etc/sysconfig/network/ifcfg-tap0
BOOTPROTO='none'
IPADDR=''
NETMASK=''
PREFIXLEN=''
STARTMODE='auto'
TUNNEL='tap'
TUNNEL_SET_GROUP='nogroup'
TUNNEL_SET_OWNER='_openqa-worker'
```

- Add bridge config with all tap devices that should be connected to it:

The file have to be located in */etc/sysconfig/network/* directory. File name is *ifcfg-br<N>*, where *<N>* is the id of the bridge (e.g. *1*).

```
# /etc/sysconfig/network/ifcfg-br1
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'
```

## Configure NAT with SuSEfirewall

The IP 10.0.2.2 can be also served as a gateway to access outside network. For this, a NAT between *br1* and *eth0* must be configured with SuSEfirewall or iptables.

```
# /etc/sysconfig/SuSEfirewall2
FW_DEV_INT="br1"
FW_ROUTE="yes"
FW_MASQUERADE="yes"
```

Start SuSEfirewall2 and allow to run on startup:

```
systemctl start SuSEfirewall2
systemctl enable SuSEfirewall2
```

## Configure NAT with firewalld

To configure NAT with firewalld assign the bridge interface to the internal zone and the interface with access to the network to the external zone:

```
firewall-cmd --permanent --zone=external --add-interface=eth0
firewall-cmd --permanent --zone=internal --add-interface=br1
```

Reload firewall configuration using `firewall-cmd --reload` command. To enable masquerade one can use the following command:

```
firewall-cmd --permanent --zone=external --add-masquerade
```

`ip_forward` is enabled automatically if masquerading is enabled:

```
cat /proc/sys/net/ipv4/ip_forward
1
```

In case interface is in trusted network it is possible to accept connections by default by changing zone target:

```
firewall-cmd --permanent --zone=external --set-target=ACCEPT
```

Alternatively, you can assign interface to the `trusted` zone.

If you do not have firewalld service running, you can use `firewall-cmd-offline` command for the configuration. Enable service to run on startup:

```
systemctl start firewalld
systemctl enable firewalld
```

Also, the `firewall-config` GUI tool for firewalld can be used for configuration.

## Configure OpenQA Workers

- Allow workers to run multi-machine jobs:

```
# /etc/openqa/workers.ini
[global]
WORKER_CLASS = qemu_x86_64,tap
```

- Enable workers to be started on system boot:

```
systemctl enable openqa-worker@1
systemctl enable openqa-worker@2
systemctl enable openqa-worker@3
```

## Grant CAP\_NET\_ADMIN Capabilities to QEMU

In order to use TAP device which doesn't exist on the system, it is required 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
```

## Configure Network Manager

- Check the configuration for the *eth0* interface:

### IMPORTANT

Ensure, that *eth0* interface is configured in */etc/sysconfig/network/ifcfg-eth0*. Otherwise, wicked will not be able to bring up the interface on start and host will loose network connection.

```
# /etc/sysconfig/network/ifcfg-eth0
BOOTPROTO='dhcp'
BROADCAST=''
ETHTOOL_OPTIONS=''
IPADDR=''
MTU=''
NAME=''
NETMASK=''
REMOTE_IPADDR=''
STARTMODE='auto'
DHCLIENT_SET_DEFAULT_ROUTE='yes'
```

- Start *wicked* as network service:

Check the network service currently being used:

```
systemctl show -p Id network.service
```

If the result is different from *Id=wicked.service* (e.g. *NetworkManager.service*), stop the network

service:

```
systemctl stop network.service  
systemctl disable network.service
```

Then switch to wicked:

```
systemctl enable --force wicked  
systemctl start wicked
```

- Bring up *br1* interface:

```
wicked ifup br1
```

- REBOOT

## Debugging Open vSwitch Configuration

Boot sequence with wicked < 0.6.23:

1. wicked - creates tap devices
2. openvswitch - creates the bridge br1, adds tap devices to it
3. wicked handles br1 as hotplugged device, assigns 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 br1 and tap devices, adds 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 br1, the tap devices are assigned to it  
ovs-ofctl dump-flows br1 # shows the rules installed by os-autoinst-openvswitch in  
table=0
```

- packets from tapX to br1 create additional rules in table=1
- packets from br1 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



```
iptables -L -v
```

As long as the SUT has access to external network, there should be nonzero packet count in the forward chain between br1 and external interface.

# GRE Tunnels

By default all multi-machine workers have to be on single physical machine. You can join multiple physical machines and its ovs bridges together by GRE tunnel.

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

Create gre\_tunnel\_preup script (change remote\_ip value correspondingly on both hosts)

```
# /etc/wicked/scripts/gre_tunnel_preup.sh
#!/bin/sh
action="$1"
bridge="$2"
ovs-vsctl --may-exist add-port $bridge gre1 -- set interface gre1 type=gre
options:remote_ip=<IP address of other host>
```

And call it by PRE\_UP\_SCRIPT="wicked:gre\_tunnel\_preup.sh" entry

```
# /etc/sysconfig/network/ifcfg-br1
<..>
PRE_UP_SCRIPT="wicked:gre_tunnel_preup.sh"
```

Allow GRE in firewall

```
# /etc/sysconfig/SuSEfirewall2
FW_SERVICES_EXT_IP="GRE"
FW_SERVICES_EXT_TCP="1723"
```

## NOTE

When using GRE tunnels keep in mind that VMs inside the ovs bridges have to use MTU=1458 for their physical interfaces (eth0, eth1). If you are using support\_server/setup.pm the MTU will be set automatically to that value on support\_server itself and it does MTU advertisement for DHCP clients as well.

# 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 tidy script 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.

To execute a single test, one can use prove. You must set TEST\_PG so the database can be found. If you set a custom base directory, be sure to unset it when running tests. Example:

```
TEST_PG='DBI:Pg:dbname=openqa_test;host=/dev/shm/tpg' OPENQA_BASEDIR= prove -v t/14-grutasks.t
```

To speed up the test initialization, start PostgreSQL using t/test\_postgresql instead of using the system service. Eg.

```
t/test_postgresql /dev/shm/tpg
```

- 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 `cpanfile` at the root of the openQA repository.

openQA relies on PostgreSQL to store the information. It used to support SQLite, but that is no longer possible.

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.



# Starting the webserver from local Git checkout

- To start the webserver for development, use the scripts/openqa daemon.
- openQA will pull the required assets on the first run.
- openQA uses SASS, so Ruby development files are required. Under openSUSE, installing the packages `devel_C_C++` and `ruby-devel` should be sufficient. openQA will install the required files automatically under `.gem`. Add `.gem/ruby/2.4.0/bin` to the `PATH` variable to let it find the `sass/scss` binaries. I also had to create symlinks of those binaries without `.ruby2.4` suffix so openQA could find them.
- It is also useful to start openQA with `morbo` which allows applying changes without restarting the server: `morbo -m development -w assets -w lib -w templates -l http://localhost:9526 script/openqa daemon`

# Managing the database

During the development process there are cases in which the database schema needs to be changed. there are some steps that have to be followed so that new database instances and upgrades include those changes.

## When is it required to update the database schema?

After modifying files in lib/OpenQA/Schema/Result. However, not all changes require to update the schema. Adding just another method or altering/adding functions like has\_many doesn't require an update. However, adding new columns, modifying or removing existing ones requires to follow the steps mentioned above.

## How to update the database schema

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 PostgreSQL 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. Migration scripts to upgrade from previous versions can be added under `dbicdh/_common/upgrade`. Create a `<prev_version>-<new_version>` directory and put some files there with DBIx commands for the migration. For examples just have a look at the migrations which are already there.

The above steps are only for preparing the required SQL statements, but do not actually alter the database. Before doing so, it is recommended **to backup your database** to be able to downgrade again if something goes wrong or you just need to continue working on another branch. To do so, the following command can be used to create a copy:

```
createdb -O ownername -T originaldb newdb
```

To actually create or update the database (after creating a backup as described), you should run either `./script/initdb --init_database` or `./script/upgradedb --upgrade_database`. This is also required when the changes are installed in a production server.

# How to add fixtures to the database

Note: This section is not about the fixtures for the testsuite. Those are located under `t/fixtures`.

Note: This section might not be relevant anymore. At least there are currently none of the mentioned directories with files containing SQL statements present.

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.

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 PostgreSQL - under openSUSE the following packages 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 [Database](#) 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_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`.

# Running tests of openQA itself

There's two ways of executing the testsuite locally:

1. with docker

The goal of running the tests with docker is to have consistent tests results (as sometimes the tests have different outcomes because missing packages or different package versions amongst other reasons). This is the preferred way if the user wants to run a full test battery or if it needs to setup a test database

2. without docker

## How to run tests with docker

To run them in docker please be sure that docker is installed and the docker daemon is running. To launch the test suite first it's required to pull the docker image:

```
docker pull registry.opensuse.org/devel/openqa/containers/openqa_dev:latest
```

Build the image using Makefile target:

```
make docker-test-build
```

Launch the tests using Makefile target:

```
make docker-test-run
```

Run tests by invoking Docker manually:

```
docker run -v OPENQA_LOCAL_CODE:/opt/openqa -v /var/run/dbus:/var/run/dbus -e VAR1=1  
-e VAR2=1 openqa:latest make docker-tests
```

Replace OPENQA\_LOCAL\_CODE to the location where you have the openqa code.

Replace VAR1 and VAR2 in -e switch to match a test battery of the test matrix:

FULLSTACK=0	UITESTS=0
FULLSTACK=0	UITESTS=1
GH_PUBLISH=true	FULLSTACK=1
SCHEDULER_FULLSTACK=1	
DEVELOPER_FULLSTACK=1	



## tips

Running commands will be executed after the initialization script (database creation and so on..). So if there's the need to run an interactive session after it just do:

```
docker run -it -v OPENQA_LOCAL_CODE:/opt/openqa -v /var/run/dbus:/var/run/dbus
registry.opensuse.org/devel/openqa/containers/openqa_dev bash
```

Of course you can also use `make docker-tests \; bash` to run the tests first and then open a shell for further investigation.

There's also the possibility to change the initialization scripts with the `--entrypoint` switch. This allows us to go into an interactive session without any initialization script run:

```
docker run -it --entrypoint /bin/bash -v OPENQA_LOCAL_CODE:/opt/openqa -v
/var/run/dbus:/var/run/dbus registry.opensuse.org/devel/openqa/containers/openqa_dev
```

In case there's the need to follow what's happening in the current running docker (the execution will terminate the session):

```
docker exec -ti $(docker ps | awk '!/CONTAINER/{print $1}') /bin/bash
```

Running UI tests in non-headless mode is also possible, eg.:

```
xhost +local:root
docker run --rm -ti --name openqa-testsuite -v /tmp/.X11-unix:/tmp/.X11-unix:rw -e
DISPLAY="$DISPLAY" -e NOT_HEADLESS=1 prove -v t/ui/14-dashboard.t
xhost -local:root
```

It is also possible to use a custom `os-autoinst` checkout using the following arguments:

```
docker run ... -e CUSTOM_OS_AUTOINST=1 -v /path/to/your/os-autoinst:/opt/os-autoinst
make docker-tests
```

By default, `configure` and `make` are still executed (so a clean checkout is expected). If your checkout is already prepared to use, set `CUSTOM_OS_AUTOINST_SKIP_BUILD` to prevent this. Be aware that the build produced outside of the container might not work inside the container if both environments provide different, incompatible library versions (eg. OpenCV).

It is also important to mention that your local repositories will be copied into the container. This can take very long if those are big, eg. when the openQA repo contains a lot of profiling data because you enabled `Mojolicious::Plugin::NYTProf`.

In general, if starting the tests via Docker seems to hang, it is a good idea to inspect the process tree to see which command is currently executed.

## How to run tests without docker

To execute the testsuite locally without docker, use `make test`. It is also possible to run a particular test for example `prove t/api/01-workers.t`.

To run UI tests the package `perl-Selenium-Remote-Driver` is required. Note that the version provided by Leap 42.2 is too old. The version from the repository `devel-languages-perl` can be used instead.

You need to install `chromedriver` and either `chrome` or `chromium` for the ui tests.

# 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.