openQA Documentation

openQA Team

Table of Contents

openQA starter guide

Introduction

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

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

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

This document describes the general operation and usage of openQA. The main goal is to provide a general overview of the tool, with all the information needed to become a happy user.

For a quick start, if you already have an openQA instance available you can refer to the section Cloning existing jobs - openqa-clone-job directly to trigger a new test based on already existing job. For a quick installation refer directly to Quick bootstrapping under openSUSE or Container based setup.

For the installation of openQA in general see the Installation Guide, as a user of an existing instance see the Users Guide. More advanced topics can be found in other documents. All documents are also 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 can run on the same machine as well as connected via network on multiple machines within the same network or distributed. Running the web application as well as the workers in the cloud is perfectly possible.

Note that the diagram shown above is simplified. There exists a more sophisticated version which is more complete and detailed. (The diagram can be edited via its underlying GraphML file.)

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", also called "Medium Types" in the web interface of openQA

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. Here is (an incomplete list) of these states:

- **scheduled** Initial state for newly created jobs. Queued for future execution.

- **setup**/**running**/**uploading** In progress.

- **cancelled** The job was explicitly cancelled by the user or was replaced by a clone (see below) and the worker has not acknowledged the cancellation yet.

- **done** The worker acknowledged that the execution finished or the web UI considers the job as abandoned by the worker.

Jobs in the final states 'cancelled' and '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 one of the final states.

- **passed** No critical check failed during the process. It does not 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 known, non-critical issue has been found. That could be that workaround needles are in place, a softfailure has been recorded explicitly via `record_soft_failure` (from os-autoinst) or a job failure has been ignored explicitly via a job label.

- **timeout_exceeded** The job was aborted because `MAX_JOB_TIME` or `MAX_SETUP_TIME` has been exceeded, see Changing timeout for details.

- **skipped** Dependencies failed so the job was not started.

- **obsoleted** The job was superseded by scheduling a new product.

- **parallel_failed**/**parallel_restarted** The job could not continue because a job which is supposed to run in parallel failed or was restarted.

- **user_cancelled**/**user_restarted** The job was cancelled/restarted by the user.

- **incomplete** The test execution failed due to an unexpected error, e.g. the network connection to the worker was lost.

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
            "click_point" : CLICK_POINT, // Optional click point
        },
        ...
    ],
    "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.

Click points

Each regular match area in a needle can optionally contain a **click point**. This is used with the `testapi::assert_and_click` function to match GUI elements such as buttons and then click inside the matched area.

```
{
  "xpos" : INTEGER, // Relative coordinates inside the match area
  "ypos" : INTEGER,
  "id" : STRING,    // Optional
}
```

Each click point can have an `id`, and if a needle contains multiple click points you must pass it to `testapi::assert_and_click` to select which click point to use.

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 when there are recent test results in these job groups 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 old jobs and assets are deleted at some point.

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

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

NOTE Checkout the Cleanup section for more details and the Build tagging section for how to mark a job as important.

NOTE New groups use the limits configured in the `[default_group_limits]` section of `/etc/openqa/openqa.ini`. Jobs outside of any group use the limits configured in the `[no_group_limits]` section of `/etc/openqa/openqa.ini`.

NOTE Archiving of important jobs can be enabled. Checkout the related settings within the `[archiving]` section of the config file for details.

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`.

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/openqa-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
openqa-cli api -X POST isos \
        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
openqa-cli api -X POST isos \
        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 is 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 is assumed that the reader is already familiar with the concepts of openQA and has already read the Getting Started Guide, also available at the official repository.

Continue with the next section Container based setup to setup a simple, ready-to-use container based openQA instance which is useful for a single user setup. For a quick boostrapping under openSUSE go to Quick bootstrapping. Else, continue with the more advanced section about Custom installation. For a setup suitable to develop openQA itself, have a look at the Development setup section.

Container based setup

openQA is provided in containers. Multiple variants exist.

Single-instance container

The easiest and quickest way to spawn a single instance of openQA with a single command line using the `podman` container engine is the following:

```
podman run -p 1080:80 -p 1443:443 --rm -it
registry.opensuse.org/devel/openqa/containers/openqa-single-instance
```

Once the startup has finished, the web UI is accessible via http://localhost:1080 or https://localhost:1443.

Separate web UI and worker containers

As an alternative also separate containers are provided for both the web UI and worker.

For example the web UI container can be pulled and started using the `podman` container engine:

```
podman run -p 1080:80 -p 1443:443 --rm -it
registry.opensuse.org/devel/openqa/containers15.5/openqa_webui:latest
```

The worker container can be pulled and started with:

```
podman run --rm -it
registry.opensuse.org/devel/openqa/containers15.5/openqa_worker:latest
```

Custom configuration for containers

To supply a custom openQA config file, use the `-v` parameter. This also works for the database config file. Note that if a custom database config file is specified, no database is launched within the container. By default, the web UI container uses the self-signed certificate that comes with Mojolicious. To supply a different certificate, use the `-v` parameter. Example for running openQA with a custom config and certificate:

```
podman run -p 1080:80 -p 1443:443 \
  -v ./container/webui/test-cert.pem:/etc/apache2/ssl.crt/server.crt:z \
  -v ./container/webui/test-key.pem:/etc/apache2/ssl.key/server.key:z \
  -v ./container/webui/test-cert.pem:/etc/apache2/ssl.crt/ca.crt:z \
  -v ./container/webui/conf/openqa.ini:/data/conf/openqa.ini:z \
  --rm -it registry.opensuse.org/devel/openqa/containers15.5/openqa_webui:latest
```

The same works for the workers container where you most likely want to to supply `workers.ini` and `client.conf`:

```
podman run \
  -v ./container/worker/conf/workers.ini:/data/conf/workers.ini:z \
  -v ./container/worker/conf/client.conf:/data/conf/client.conf:z \
  --rm -it registry.opensuse.org/devel/openqa/containers15.5/openqa_worker:latest
```

This examples assume the working directory is an openQA checkout. To avoid doing a checkout, you can also grab the config files from the webui/conf and worker/conf directory listings on GitHub.

To learn more about how to run workers checkout the Run openQA workers section.

For creating a first test job, checkout the Triggering tests section. Note that the commands mentioned there can also be invoked within a container, e.g.:

```
podman run \
    --rm -it registry.opensuse.org/devel/openqa/containers15.5/openqa_webui:latest \
    openqa-cli --help
```

Checkout the containerized setup section for more details.

Take a look at openSUSE's registry for all available container images.

Quick bootstrapping under openSUSE

To quickly get a working openQA installation, you can use the openqa-bootstrap script. It essentially automates the steps mentioned in the Custom installation section.

Directly on your machine

On openSUSE Leap and openSUSE Tumbleweed to setup openQA on your machine simply download and execute the openqa-bootstrap script as root - it will do the rest for you:

```
curl -s https://raw.githubusercontent.com/os-autoinst/openQA/master/script/openqa-
bootstrap | bash -x
```

The script is also available from an openSUSE package to install from:

```
zypper in openQA-bootstrap
/usr/share/openqa/script/openqa-bootstrap
```

openQA-bootstrap supports to immediately clone an existing job simply by supplying openqa-clone-job parameters directly for a quickstart:

```
/usr/share/openqa/script/openqa-bootstrap --from openqa.opensuse.org 12345
SCHEDULE=tests/boot/boot_to_desktop,tests/x11/kontact
```

The above command will bootstrap an openQA installation and immediately afterwards start a local test job clone from a test job from a remote instance with optional, overridden parameters. More information about openqa-clone-job can be found in Cloning existing jobs - openqa-clone-job.

openQA in a container

You can also setup a systemd-nspawn container with openQA with the following commands. and you need to have no application listening on port 80 yet because the container will share the host system's network stack.

```
zypper in openQA-bootstrap
/usr/share/openqa/script/openqa-bootstrap-container

systemd-run -tM openqa1 /bin/bash # start a shell in the container
```

Custom installation - repositories and procedure

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 packages distributed with older versions of openSUSE Leap are 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 SUSE Linux Enterprise (SLE), openSUSE Leap and Tumbleweed packages are available.
- 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 -p 95 -f
'http://download.opensuse.org/repositories/devel:openQA/openSUSE_Tumbleweed'
devel_openQA

# openSUSE Leap/SLE
zypper ar -p 95 -f
'http://download.opensuse.org/repositories/devel:openQA/$releasever' devel_openQA
zypper ar -p 90 -f
'http://download.opensuse.org/repositories/devel:openQA:Leap:$releasever/$releasever'
devel_openQA_Leap
```

NOTE   If you installed openQA from the official repository first, you may need to change the vendor of the dependencies.

```
# openSUSE Tumbleweed and Leap
zypper dup --from devel_openQA --allow-vendor-change

# openSUSE Leap
zypper dup --from devel_openQA_Leap --allow-vendor-change
```

Installation

You can install the main openQA server package using these commands.

```
# openSUSE
zypper in openQA

# SLE
. /etc/os-release
SUSEConnect -p sle-module-desktop-applications/$VERSION_ID/$CPU
SUSEConnect -p sle-module-development-tools/$VERSION_ID/$CPU
SUSEConnect -p sle-we/$VERSION_ID/$CPU -r $sled_key
SUSEConnect -p PackageHub/$VERSION_ID/$CPU
zypper in openQA

# Fedora
dnf install openqa openqa-httpd
```

To install the openQA worker package use the following.

```
# SLE/openSUSE
zypper in openQA-worker
```

Different convenience packages exist for convenience in openSUSE, for example: `openQA-local-db` to install the server including the setup of a local PostgreSQL database or `openQA-single-instance` which sets up a web UI server, a web proxy as well as a local worker. Install `openQA-client` if you only want to interact with existing, external openQA instances.

Installation from sources

Installing is not required for development purposes and most components of openQA can be called directly from the repository checkout.

To install openQA from sources make sure to install all dependencies as explained in Dependencies. Then one can call

```
make install
```

The directory prefix can be controlled with the optional environment variable `DESTDIR`.

From then on continue with the Basic configuration.

System requirements

To run tests based on the default qemu backend the following hardware specifications are recommended per openQA worker instance:

- 1x CPU core with 2x hyperthreads (or 2x CPU cores)
- 8GB RAM
- 40GB HDD (preferably SSD or NVMe)

Basic configuration

For a local instance setup you can simply execute the script:

```
/usr/share/openqa/script/configure-web-proxy
```

This will automatically setup a local Apache http proxy. Read on for more detailed setup instructions with all the details.

If you wish to run openQA behind an http proxy (Apache, NGINX, ...) then see the **openqa.conf.template** config file in **/etc/apache2/vhosts.d** (openSUSE) or `/etc/httpd/conf.d` (Fedora) when using apache2 or the config files in `/etc/nginx/vhosts.d` for NGINX.

Apache proxy

To make everything work correctly on openSUSE when using Apache, 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` setting if required. This will direct all HTTP traffic to openQA.

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

NGINX proxy

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

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

Note that the default config in **openqa.conf.template** is using the keyword `default_server` in the `listen` statement. This will only change the behaviour when accessing the server via its IP address. This means that the default vhost for `localhost` in **nginx.conf** will take precedence when accessing the server via `localhost`. You might want to disable it.

For openQA you need to set `httpsonly = 0` as described in the TLS/SSL section below, if you do not setup NGINX for SSL.

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

openQA uses PostgreSQL as database. By default, a database with name `openqa` and `geekotest` user as owner is used. An automatic setup of a freshly installed PostgreSQL instance can be done using this script. The database connection can be configured in `/etc/openqa/database.ini` (normally the `[production]` section is relevant). More info about the `dsn` value format can be found in the DBD::Pg documentation.

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
```

User authentication

openQA supports three different authentication methods: OpenID (default), OAuth2 and Fake (for

development).

Use the `auth` section in `/etc/openqa/openqa.ini` to configure the method:

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

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

Note that only one authentication method and only one OpenID/OAuth2 provider can be configured at a time. When changing the method/provider no users/permissions are lost. However, a new and distinct user (with default permissions) will be created when logging in via a different method/provider because there is no automatic mapping of identities across different methods/providers.

For authentication to work correctly the clocks on workers and the web UI need to be in sync. The best way to achieve that is to install a service that implements the time-sync target. Otherwise a "timestamp mismatch" may be reported when clocks are too far apart.

OpenID

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

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

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

This method supports OpenID version up to 2.0.

OAuth2

An additional Mojolicious plugin is required to use this feature:

```
# openSUSE
zypper in 'perl(Mojolicious::Plugin::OAuth2)'
```

Example for configuring OAuth2 with GitHub:

```
[auth]
# method name is case sensitive!
method = OAuth2

[oauth2]
provider = github
key = mykey
secret = mysecret
```

In order to use GitHub for authorization, an "OAuth App" needs to be registered on GitHub. Use ⋯/login as callback URL. Afterwards the key and secret will be visible to the application owner(s).

As shown in the comments of the default configuration file, it is also possible to use different providers.

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:

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

[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

To start openQA and enable it to run on each boot call

```
systemctl enable --now postgresql
systemctl enable --now openqa-webui
systemctl enable --now openqa-scheduler
# to use Apache as reverse proxy under openSUSE
systemctl enable apache2
systemctl restart apache2
# to use Apache as reverse proxy under Fedora
# for now this is necessary to allow Apache to connect to openQA
setsebool -P httpd_can_network_connect 1
systemctl enable httpd
systemctl restart httpd
```

The openQA web UI should be available on http://localhost/ now. To simply start openQA without enabling it permanently one can simply use `systemctl start` instead.

Run openQA workers

Workers are services running backends to perform the actual testing. The testing is commonly performed by running virtual machines but depending on the specific backend configuration different options exist.

It is possible to run openQA workers on the same machine as the web UI as well as on different machines, even in different networks, for example instances in public cloud. The only requirement is access to the web UI host over HTTP/HTTPS. For running tests based on virtual machines KVM support is recommended.

The openQA worker is distributed as a separate package which be installed on multiple machines while still using only one web UI.

```
# 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 need, you just need to supply a different 'instance number' (the 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.

If you start openQA workers on a different machine than the web UI host make sure to have synchronized clocks, for example using NTP, to prevent inconsistent test results.

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

Cleanup

The automated cleanup is enabled and configured by default. Cleanup tasks are scheduled via systemd timer units and run via `openqa-gru.service`. The configuration is done in `/etc/openqa/openqa.ini` and various places within the web UI. If you want to tweak the cleanup to your needs, have a look at the Cleanup of assets, results and other data section.

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.

It might also be useful to rebase first. To enable that, add the remote to get the latest updates from and the branch to rebase against to your openqa.ini:

```
[scm git]
update_remote = origin
update_branch = origin/master
```

If rebasing, it may be useful to perform a hard reset of the local repository to ensure that the rebase

will not fail. To enable that, add the following to your openqa.ini (along with the previous snippet):

```
[scm git]
do_cleanup = yes
```

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 referrers 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.

Further systemd units for the worker

The following information is partially openSUSE specific. The `openQA-worker` package provides further systemd units:

- `openqa-worker-plain@.service`: standard worker service, this is the default and `openqa-worker@.service` is just a symlink to this service

- `openqa-worker-no-cleanup@.service`: see enabling snapshots

- `openqa-worker-auto-restart@.service`: worker that restarts automatically after processing assigned jobs

- `openqa-worker-cacheservice/openqa-worker-cacheservice-minion`: services for the asset cache

- openqa-worker.target

  - Starts openqa-worker@.service (but no other worker units) when started.

    - The number of started worker slots depends on the pool directories present under /var/lib/openqa/pool. This information is determined via a systemd generator and can be refreshed via systemctl daemon-reload.

  - Stops openqa-worker-no-cleanup@.service and other units conflicting with openqa-worker@.service when started.

  - Stops/restarts **all** worker units when stopped/restarted.

  - Is restarted automatically when the openQA-worker package is updated (unless DISABLE_RESTART_ON_UPDATE="yes" is set in /etc/sysconfig/services).

- openqa-reload-worker-auto-restart@.path: allows to restart the worker service automatically on configuration changes without interrupting jobs (see next section for details)

Stopping/restarting workers without interrupting currently running jobs

It is possible to stop a worker as soon as it becomes idle and immediately if it is already idling by sending SIGHUP to the worker process.

When the worker is setup to be always restarted (e.g. using a systemd unit with Restart=always like openqa-worker-auto-restart@*.service) this leads to the worker being restarted without interrupting currently running jobs. This can be useful to apply configuration changes and updates without interfering ongoing testing. Example:

```
systemctl reload 'openqa-worker-auto-restart@*.service' # sends SIGHUP to worker
```

There is also the systemd unit openqa-reload-worker-auto-restart@.path which invokes the command above (for the specified slot) whenever the worker configuration under /etc/openqa/workers.ini changes. This unit is not enabled by default and only affects openqa-worker-auto-restart@.service but not other worker services.

This kind of setup makes it easy to take out worker slots temporarily without interrupting currently running jobs:

```
# prevent worker services from restarting and being automatically reloaded
systemctl stop openqa-reload-worker-auto-restart@{1..28}.{service,path}
systemctl mask openqa-worker-auto-restart@{1..28}.service
# ensure idling worker services stop now (`--kill-who=main` ensures only the
# worker receives the signal and *not* isotovideo)
systemctl kill --kill-who=main --signal HUP openqa-worker-auto-restart@{1..28}
```

Configuring remote workers

There are some additional requirements to get remote worker running. First is to ensure shared storage between openQA web UI 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 web UI 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`:

```
# Enable the AMQP plugin
[global]
plugins = AMQP

# 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]
HOST = 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. Workers 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 a worker registers to an openQA instance, scheduled jobs (of matching worker class) can be assigned to it. Dependencies between jobs will be considered for ordering the job assignment. It is possible to mix local openQA instance with remote instances or use only remote instances.

Asset and test/needle caching

If your network is slow or you experience long time to load needles you might want to consider enabling caching on your remote workers. To enable caching, `CACHEDIRECTORY` must be set in `workers.ini`. There are also further settings one can optionally configure. Example:

```
[global]
HOST = http://webui
CACHEDIRECTORY = /var/lib/openqa/cache # desired cache location
CACHELIMIT = 50 # max. cache size in GiB, defaults to 50
CACHE_MIN_FREE_PERCENTAGE = 10 # min. free disk space to preserve in percent
CACHEWORKERS = 5 # number of parallel cache minion workers, defaults to 5

[http://webui]
TESTPOOLSERVER = rsync://yourlocation/tests # also cache tests (via rsync)
```

The specified `CACHEDIRECTORY` must exist and must be writable by the cache service (which usually

runs as `_openqa-worker` user). If you install openQA through the repositories, said directory will be created for you.

The shown configuration causes workers to download the assets from the web UI and use them locally. The `TESTPOOLSERVER` setting causes also tests and needles to be downloaded via `rsync` from the specified location. You can find further examples in the comments in `/etc/openqa/workers.ini`.

It is suggested to have the cache and pool directories on the same filesystem to ensure assets used by tests are available as long as needed. This is achieved by using hard links, resorting to symlinks in other cases with the risk of assets being deleted from the cache before tests relying on these assets end.

The caching is provided by two additional services which need to be started on the worker host:

```
systemctl enable --now \
    openqa-worker-cacheservice openqa-worker-cacheservice-minion
```

The rsync server daemon needs to be configured and started on the web UI host.

Example `/etc/rsyncd.conf`:

```
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

[tests]
path = /var/lib/openqa/share/tests
comment = openQA test distributions
```

```
systemctl enable --now rsyncd
```

Enable linking files referred by job settings

Specific job settings might refer to files within the test distribution. You can configure openQA to display links to these files within the job settings tab. To enable particular settings to be presented as a link within the settings tab one can setup the relevant keys in `/etc/openqa/openqa.ini`.

```
[job_settings_ui]
keys_to_render_as_links=FOO,AUTOYAST
```

The files referenced by the configured keys should be located either under the root of `CASEDIR` or the data folder within `CASEDIR`.

Enable custom hook scripts on "job done" based on result

If a job is done, especially if no label could be found for carry-over, often more steps are needed for the review of the test result or providing the information to either external systems or users. As there can be very custom requirements openQA offers a point for optional configuration to let the instance administrators define specific actions.

By setting custom hooks it is possible to call external scripts defined in either environment variables or config settings.

If an environment variable corresponding to the job result is found following the name pattern `OPENQA_JOB_DONE_HOOK_$RESULT`, any executable specified in the variable as absolute path or executable name in `$PATH` is called with the job ID as first and only parameter. For example for a job with result "failed", the corresponding environment variable would be `OPENQA_JOB_DONE_HOOK_FAILED`. As alternative to an environment variable a corresponding config variable in the section `[hooks]` in lower-case without the `OPENQA_` prefix can be used in the format `job_done_hook_$result`. The corresponding environment value has precedence. The exit code of the externally called script is not evaluated and will have no effect.

It is also possible to specify one general hook script via `job_done_hook` and enable that one for specific results via e.g. `job_done_hook_enable_failed = 1`.

The job setting `_TRIGGER_JOB_DONE_HOOK=0` allows to disable the hook script execution for a particular job. It is also possible to specify `_TRIGGER_JOB_DONE_HOOK=1` to execute the general hook script configured via `job_done_hook` regardless of the result.

The execution time of the script is by default limited to five minutes. If the script does not terminate after receiving `SIGTERM` for 30 seconds it is terminated forcefully via `SIGKILL`. One can change that by setting the environment variables `OPENQA_JOB_DONE_HOOK_TIMEOUT` and `OPENQA_JOB_DONE_HOOK_KILL_TIMEOUT` to the desired timeouts. The format from the `timeout` command is used (see `timeout --help`).

For example there is already an approach called "auto-review" https://github.com/os-autoinst/scripts/#auto-review---automatically-detect-known-issues-in-openqa-jobs-label-openqa-jobs-with-ticket-references-and-optionally-retrigger which offers helpful, external scripts. Config settings for openqa.opensuse.org enabling the auto-review scripts could look like:

```
[hooks]
job_done_hook_incomplete = /opt/openqa-scripts/openqa-label-known-issues-hook
job_done_hook_failed = /opt/openqa-scripts/openqa-label-known-issues-hook
```

or for a host openqa.example.com:

```
[hooks]
job_done_hook_incomplete = env host=openqa.example.com /opt/openqa-scripts/openqa-
label-known-issues-hook
job_done_hook_failed = env host=openqa.example.com /opt/openqa-scripts/openqa-label-
known-issues-hook
```

The environment variable should be set in a systemd service override for the GRU service. A corresponding systemd override file `/etc/systemd/system/openqa-gru.service.d/override.conf` could look like this:

```
[Service]
Environment="OPENQA_JOB_DONE_HOOK_INCOMPLETE=/opt/os-autoinst-scripts/openqa-label-
known-issues-hook"
```

When using `apparmor` the called hook scripts must be covered by according `apparmor` rules, for example for the above in `/etc/apparmor.d/usr.share.openqa.script.openqa`:

```
/opt/os-autoinst-scripts/** rix,
/usr/bin/cat rix,
/usr/bin/curl rix,
/usr/bin/jq rix,
/usr/bin/mktemp rix,
/usr/share/openqa/script/client rix,
```

Additions should be added to `/etc/apparmor.d/local/usr.share.openqa.script.openqa` after which the **apparmor** service needs to be restarted for changes to take effect. Note that in case of symlinks the target must be specified, and the link itself is irrelevant. So for example `Can␀t exec "/bin/sh"` can occur if `/bin/sh` is a link to a path that's not allowed.

Apparmor denials and stderr output of the hook scripts are visible in the system logs of the openQA GRU service, except for messages in "complain" mode which end up in `audit.log`. General status and stdout output is visible in the GRU minion job dashboard on the route `/minion/jobs?offset=0&task=finalize_job_results` of the openQA instance.

Automatic cloning of incomplete jobs

By default, when a worker reports an incomplete job due to a cache service related problem, the job is automatically cloned. It is possible to extend the regex to cover other types of incompletes as well by adjusting `auto_clone_regex` in the `global` section of the config file. It is also possible to assign `0` to prevent the automatic cloning.

Note that jobs marked as incomplete by the stale job detection are not affected by this configuration and cloned in any case.

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 blocklist:

```
[audit]
blocklist = job_grab job_done
```

The `audit/storage_duration` section of `/etc/openqa/openqa.ini` allows to set the retention policy for different audit event types:

```
[audit/storage_duration]
startup = 10
jobgroup = 365
jobtemplate = 365
table = 365
iso = 60
user = 60
asset = 30
needle = 30
other = 15
```

In this example events of the type `startup` would be cleaned up after 10 days, events related to job groups after 365 days and so on. Events which do not fall into one of these categories would be cleaned after 15 days. By default, cleanup is disabled.

Use `systemctl enable --now openqa-enqueue-audit-event-cleanup.timer` to schedule the cleanup automatically every day. It is also possible to trigger the cleanup manually by invoking `/usr/share/openqa/script/openqa minion job -e limit_audit_events`.

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_update user_login user_deleted
- Comments:
  - comment_create comment_update comment_delete
- 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 on the blocklist by default.

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

Automatic system upgrades and reboots of openQA hosts

The distribution package `openQA-auto-update` offers automatic system upgrades and reboots of openQA hosts. To use that feature install the package `openQA-auto-update` and enable the corresponding systemd timer:

```
systemctl enable openqa-auto-update.timer
```

This triggers a nightly system upgrade which first looks into configured openQA repositories for stable packages, then conducts the upgrade and schedules reboots during the configured reboot maintenance windows using `rebootmgr`. As an alternative to the systemd timer the script `/usr/share/openqa/script/openqa-auto-update` can be called when desired.

The distribution package `openQA-continuous-update` can be used to continuously upgrade the system. It will frequently check whether `devel:openQA` contains updates and if it does it will upgrade the whole system. This approach is independent of `openQA-auto-update` but can be used complementary. The configuration is analogous to `openQA-auto-update`.

Migrating from older databases

For older versions of openQA, you can migrate from SQLite to PostgreSQL according to DB migration from SQLite to PostgreSQL.

For migrating from older PostgreSQL versions read on.

Migrating PostgreSQL database on openSUSE

The PostgreSQL `data`-directory needs to be migrated in order to switch to a newer major version of PostgreSQL. The following instructions are specific to openSUSE's PostgreSQL and openQA packaging but with a little adaption they can likely be used for other setups as well. These instructions can migrate big databases in seconds without requiring additional disk space. However, services need to be stopped during the (short) migration.

1. Locate the `data`-directory. Its path is configured in `/etc/sysconfig/postgresql` and should be `/var/lib/pgsql/data` by default. The paths in the next steps assume the default.

2. To ease migrations, it is recommended making the `data`-directory a symlink to a versioned directory. So the file system layout would look for example like this:

   ```
   $ sudo -u postgres ls -l /var/lib/pgsql | grep data
   lrwxrwxrwx  1 root     root        7  8. Sep 2019  data -> data.10
   drwx------ 20 postgres postgres 4096 30. Aug 00:00 data.10
   drwx------ 20 postgres postgres 4096  8. Sep 2019  data.96
   ```

   The next steps assume such a layout.

3. Install same set of postgresql* packages as are installed for the old version:

   ```
   oldver=10 newver=12
   sudo zypper in postgresql$newver-server postgresql$newver-contrib
   ```

4. Change to a directory where the user postgres will be able to write logs to, e.g.:

   ```
   cd /tmp
   ```

5. Prepare the migration:

   ```
   sudo -u postgres /usr/lib/postgresql$newver/bin/initdb [locale-settings] -D
   /var/lib/pgsql/data.$newver
   ```

IMPORTANT	Be sure to use initdb from the target version (like it is done here) and also no newer version which is possibly installed on the system as well.

IMPORTANT	Lookup the locale settings in `/var/lib/pgsql/data.$oldver/postgresql.conf` or via `sudo -u geekotest psql openqa -c 'show all;' | grep lc_` to pass locale settings listed by `initdb --help` as appropriate. On some machines additional settings need to be supplied, e.g. from an older database version on openqa.opensuse.org it was necessary to pass the following settings: `--encoding=UTF8` `--locale=en_US.UTF-8` `--lc-collate=C` `--lc-ctype=en_US.UTF-8` `--lc-messages=C` `--lc-monetary=C` `--lc-numeric=C` `--lc-time=C`

6. Take over any relevant changes from the old config to the new one, e.g.:

```
sudo -u postgres vimdiff \
    /var/lib/pgsql/data.$oldver/postgresql.conf \
    /var/lib/pgsql/data.$newver/postgresql.conf
```

IMPORTANT	There shouldn't be a diff in the locale settings, otherwise `pg_upgrade` will complain.

7. Shutdown postgres server and related services as appropriate for your setup, e.g.:

```
sudo systemctl stop openqa-{webui,websockets,scheduler,livehandler,gru}
sudo systemctl stop postgresql
```

8. Perform the migration:

```
sudo -u postgres /usr/lib/postgresql$newver/bin/pg_upgrade --link \
    --old-bindir=/usr/lib/postgresql$oldver/bin \
    --new-bindir=/usr/lib/postgresql$newver/bin \
    --old-datadir=/var/lib/pgsql/data.$oldver \
    --new-datadir=/var/lib/pgsql/data.$newver
```

IMPORTANT	Be sure to use pg_upgrade from the target version (like it is done here) and also no newer version which is possibly installed on the system as well. Checkout the PostgreSQL documentation for details.

NOTE	This step only takes a few seconds for multiple production DBs because the `--link` option is used.

9. Change symlink (shown in step 2) to use the new data directory:

```
sudo ln --force --no-dereference --relative --symbolic /var/lib/pgsql/data.$newver
/var/lib/pgsql/data
```

10. Start services again as appropriate for your setup, e.g.:

```
sudo systemctl start postgresql
sudo systemctl start openqa-{webui,websockets,scheduler,livehandler,gru}
```

NOTE	There is no need to take care of starting the new version of the PostgreSQL service. The start script checks the version of the data directory and starts the correct version.

11. Check whether usual role and database are present and running on the new version:

```
sudo -u geekotest psql -c 'select version();' openqa
```

12. Remove old postgres packages if not needed anymore:

```
sudo zypper rm postgresql$oldver-server postgresql$oldver-contrib postgresql$oldver
```

13. Delete the old data directory if not needed anymore:

```
sudo -u postgres rm -r /var/lib/pgsql/data.$oldver
```

Working on database-related performance problems

Without extra setup, PostgreSQL already gathers many statistics, checkout [the official documentation](#).

Enable further statistics

These statistics help to identify the most time-consuming queries.

1. Configure the PostgreSQL extension `pg_stat_statements`, see example on [the official documentation](#).

2. Ensure the extension library is installed which might be provided by a separate package (e.g. `postgresql14-contrib` for PostgreSQL 14 on openSUSE).

3. Restart PostgreSQL.

4. Enable the extension via `CREATE EXTENSION pg_stat_statements`.

Make use of these statistics

Simply query the table `pg_stat_statements`. Use `\x` in `psql` for extended mode or `substring()` on the `query` parameter for readable output. The columns are explained in the previously mentioned documentation. Here an example to show similar queries which are most time-consuming:

```
SELECT
  substring(query from 0 for 250) as query_start, sum(calls) as calls,
max(max_exec_time) as max_exec_time,
  sum(total_exec_time) as total_exec_time, sum(rows) as rows
  FROM pg_stat_statements group by query_start ORDER BY total_exec_time DESC LIMIT 10;
```

After significant schema changes consider resetting query statistics (`SELECT pg_stat_statement_reset()`) and checking the query plans (`EXPLAIN (ANALYZE, BUFFERS) …`) for the slowest queries showing up afterwards to make sure they are using indexes (and not just sequential scans).

Further things to try

1. Try to tweak database configuration parameters. For example increasing `work_mem` in `postgresql.conf` might help with some heavy queries.

2. Run `VACUUM VERBOSE ANALYZE table_name;` for any table that shows to be impacting the performance. This can take some seconds or minutes but can help to improve performance in particular after bigger schema migrations for example type changes.

Further resources

- Checkout the official documentation for more details about `EXPLAIN`. There is also service for formatting those explanations to be more readable.

- Checkout the official documentation for more details about `VACUUM ANALYZE`.

- Checkout the following documentation pages.

Filesystem layout

Tests, needles, assets, results and working directories (a.k.a. "pool directories") are located in certain subdirectories within `/var/lib/openqa`. This directory is configurable (see Customize base directory). Here we assume the default is in place.

Note that the sub directories within `/var/lib/openqa` must be accessible by the user that runs the openQA web UI (by default 'geekotest') or by the user that runs the worker/isotovideo (by default '_openqa-worker').

These are the most important sub directories within `/var/lib/openqa`:

- `db` contains the web UI's database lockfile
- `images` is where the web UI stores test screenshots and thumbnails
- `testresults` is where the web UI stores test logs and test-generated assets
- `webui` is where the web UI stores miscellaneous files
- `pool` contains working directories of the workers/isotovideo
- `share` contains directories shared between the web UI and (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

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 worker instances

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

Terms and variables for certain directories used by openQA and isotovideo

- the "base directory"
  - by default `/var/lib`
  - configurable via environment variable `OPENQA_BASEDIR`
  - referred as `$basedir` within openQA
- the "project directory"
  - defined as `$basedir/openqa`, by default `/var/lib/openqa`
  - referred as `$prjdir` within openQA
- the "share directory": contains directories shared between web UI and (remote) workers
  - defined as `$prjdir/share`, by default `/var/lib/openqa/share`
  - referred as `$sharedir` within openQA
- the "test case directory": contains a test distribution
  - by default `$sharedir/tests/$distri` or `$sharedir/tests/$distri-$version`
  - configurable via the test variable `CASEDIR` (see backend variables documentation)
  - this default is provided by openQA; when starting isotovideo manually the `CASEDIR` variable **must** be initialized by hand
  - might contain the sub directory `lib` for placing Perl modules used by the tests
- the "product directory": contains the test schedule (`main.pm`) for a certain product within a test distribution
  - by default identical to the "test case directory"

- usually a directory `products/$distri` within the "test case directory"
- configurable via the test variable `PRODUCTDIR` (see backend variables documentation)
- the "needles directory": contains reference images for a certain product within a test distribution
  - by default `$PRODUCTDIR/needles`
  - configurable via the test variable `NEEDLES_DIR` (see backend variables documentation)

Further notes

- Setting the test variables has only an influence on os-autoinst. The web UI on the other hand always relies on the directory structure described above. For the exact details how these paths are computed by the web UI have a look at `lib/OpenQA/Utils.pm`.
- When enabling the worker cache parts of the usual "share directory" are located in the specified cache directory on the worker host.

Automatic installation of the operating systems for openQA machines

As a maintainer of an openQA infrastructure running multiple openQA worker machines one likely wants to use installation recipes for automatic installations to provide a consistent and easy setup of new machines.

For this AutoYaST can be used. An example template that provides the bare basics of installing a machine with SSH and salt, e.g. to be used with https://github.com/os-autoinst/salt-states-openqa/, can be found in https://github.com/os-autoinst/openQA/blob/master/contrib/ay-openqa-worker.xml

Troubleshooting

Tests fail quickly

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

KVM does not 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
```

Performance testing

If openQA is very slow and e.g. the test setup times out because the asset caching downloads take too long it makes sense to cross-check the networking performance. This can be done via `iperf3`.

Launch the server via `iperf3 -s` on one host (e.g. the openQA web UI host). Then run a test on another host (e.g. an openQA worker host) like this:

```
iperf3 -c serverhost -i 1 -t 30   # 30 second tests, giving results every second
```

Use `-4`/`-6` to check IPv4 vs. IPv6 performance. Use `-R` to check in the other direction. Both can make a huge difference.

More examples: https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/iperf

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.

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. Define job groups in 'Job groups' menu for groups of tests
5. Select individual 'Job groups' and decide what combinations make sense and need to be tested

Machines, mediums, test suites and job templates can all set various configuration variables. The so called job templates within the job groups 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, machine and job template 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 parameterize a test to describe for what product it will run and for what kind of machines it will be executed. For example, a test suite `kde` can be run for any product that has the KDE software stack installed, like `openSUSE-DVD-x86_64` and `openSUSE-NET-i586`, and can be tested in different x86-64 and i586 machines like `64bit`, `64bit_USBBoot`, `32bit`. In this example we could have the following test scenarios considering that the "x86_64" flavor is not compatible with the `32bit` machine:

- openSUSE-DVD-x86_64-kde-64bit
- openSUSE-DVD-x86_64-kde-64bit_USBBoot
- openSUSE-NET-i586-kde-64bit
- openSUSE-NET-i586-kde-64bit_USBBoot

- openSUSE-NET-i586-kde-32bit

For every test scenario we need to configure a different instance of the test backend, for example `os-autoinst`, with a different set of parameters.

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.

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

A test suite consists of a name and a set of test variables that are used inside this particular test together with an optional description. The test variables can be used to parameterize the actual test code and influence the behaviour according to the settings.

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

Job Groups

The job groups are the place where the actual test scenarios are defined by the selection of the medium type, the test suite and machine together with a priority value.

The priority value 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 priority value are triggered. The id is the second sorting key: Of two jobs with equal requirements and same priority

value the one with lower id is triggered first.

Job groups themselves can be created over the web UI as well as the REST API. Job groups can optionally be nested into categories. The display order of job groups and categories can be configured by drag-and-drop in the web UI.

The scenario definitions within the job groups can be created and configured by different means:

- A simple web UI wizard which is automatically shown for job groups when a new medium is added to the job group.

- An intuitive table within the web UI for adding additional test scenarios to existing media including the possibility to configure the priority values.

- The scripts `openqa-load-templates` and `openqa-dump-templates` to quickly dump and load the configuration from custom plain-text dump format files using the REST API.

- Using declarative schedule definitions in the YAML format using REST API routes or an online-editor within the web UI including a syntax checker.

Variable expansion

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

For example this variable defined for Test Suite:

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

may be expanded to this job variable:

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

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
- Job template
- 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. In the special case of the `BACKEND` variable, if there is a `MACHINE` specified, the `BACKEND` value for this machine defined in openQA has highest precedence.

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.

Note that the `WORKER_CLASS` variable is not overridden in the way described above. Instead multiple occurrences are combined.
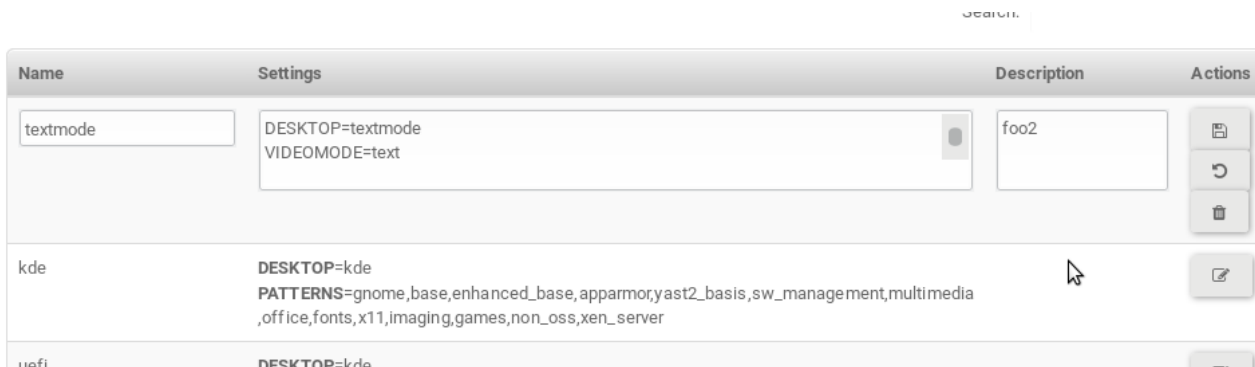
Use of the web interface

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.

Description of test suites

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

Figure 2. Entering a test suite description in the admin table using the web interface:

If a description is defined, the name of the test suite on the tests overview page shows up as a link. Clicking the link will show the description in a popup. The same syntax as for comments can be used, that is Markdown with custom extensions such as shortened links to ticket systems.
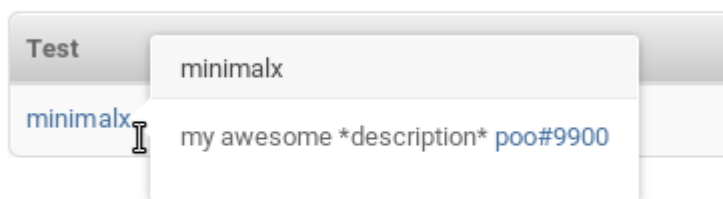
Figure 3. popover in test overview with content as configured in the test suites database:

/tests/overview - Customizable test overview page

The overview page is configurable by the filter box. Also, some additional query parameters can be provided which can be considered advanced or experimental. For example specifying no build will

resolve the latest build which matches the other parameters specified. Specifying no group will show all jobs from all matching job groups. Also specifying multiple groups works, see the following example.
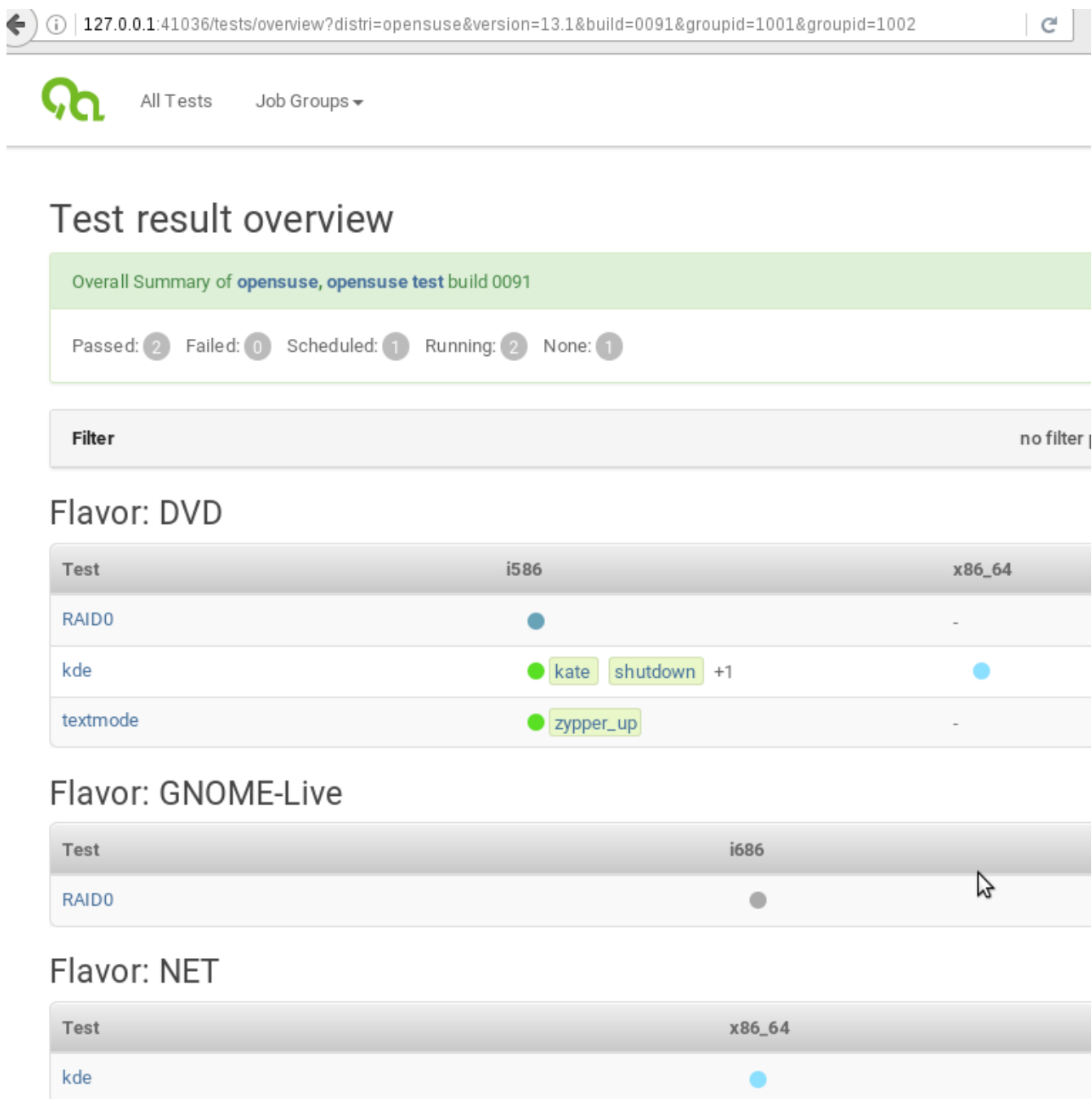
Figure 4. The openQA test overview page showing multiple groups at once. The URL query parameters specify the groupid parameter two times to resolve both the "opensuse" and "opensuse test" group.

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

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:

Meaning of the different colors

- No icon is shown if at least one failure still need to be reviewed.

- The green tick icon shows up when there is no work to be done.

- The black certificate icon is shown if all review work has been done.

- The grey comment icon is shown if all failures have at least one comment.

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

Bug references, labels and flags

Bug references

It is possible to reference a bug by writing `<bugtracker_shortname>#<bug_nr>` in a comment, e.g. `bsc#1234`. It is also possible to spell out the full URL, e.g. `https://bugzilla.suse.com/show_bug.cgi?id=1234` which will then be shortened automatically. A bug reference is rendered as link and a bug icon is displayed for the job in various places as shown in the figure below. A comment containing a bug reference will also be carried over to reduce manual review work. Refer to the Flags section below for other ways to trigger automated comment carryover.

WARNING	If you want to reference a bug without making it count as a bug reference you need to wrap it into a label (see subsequent section), e.g. `label:bsc#1234`.

Figure 5. Bug icon for job with bug reference on test result overview

All bug references are stored within the internal database of openQA. The status can be updated using the `/bugs` API route with external tools. One can set the bug status this way which will then be

shown in the web UI, see the figure below.

Figure 6. Example for visualization of closed issues: The upside down icons in red visualize closed issues.

NOTE	Also GitHub pull requests and issues can be linked. Use the generic format `<marker>[<project/repo>]<id>`, e.g. `gh#os-autoinst/openQA#1234`.

Labels

A comment can also contain labels. Use `label:<keyword>` where `<keyword>` can be any valid character up to the next whitespace, e.g. `false_positive`. A label is rendered as yellow box. 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. If a job has a label a special icon will be shown next to it in various places as shown in the figure below.

Figure 7. Label icon for job with a label on test result overview

NOTE	A label containing a bug reference will still be treated as a label, not a bugref. The bugref will still be rendered as a link. That means no bug icon is shown and the comment does **not** become subject to carry over.

Overwrite result of job

One special label format is available which allows to forcefully overwrite the result of an openQA job using a convenient openQA comment. The expected format is `label:force_result:<new_result>[:<description>]`, for example `label:force_result:failed` or `label:force_result:softfailed:bsc#1234`. For this command to be effective the according user needs to have at least operator permissions.

NOTE	`force_result`-labels are evaluated when when a comment is carried over. However, the carry over will only happen when the comment **also** contains a bug reference or `flag:carryover`.

Flags

Currently there is only one flag for job comments supported.

flag:carryover

Adding `flag:carryover` to a comment, will result in this comment being carried over to a new job failing for the same reason, without a bugref required.

Distinguish product and test issues bugref gh#708

"progress.opensuse.org" is used to track test issues, bugzilla for product issues, at least for SUSE/openSUSE. openQA bugrefs distinguish this and show corresponding icons

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. Note that openQA does not define further tag types besides the *important* tag. However, the user is free to choose any tag type as needed.

The generic format for tags is

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

The `build_id` should be set to the `BUILD` setting of the jobs (**without** the `Build`-prefix shown in dashboard pages). It is also possible to include the `VERSION` setting which then needs to be prepended and sparated by a dash (e.g. `tag:15-SP5-25.1:important:Alpha-202210-1` where `15-SP5` is the `VERSION` and `25.1` the `BUILD`).

The more recent tag always wins. Tags specifying the `VERSION` as well win over generic tags.

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.
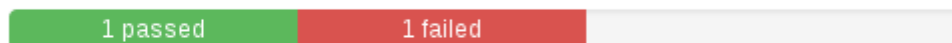
Keeping important builds

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. "TODO" means that tests still require review.

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

and customizing the limits. Also the 'All tests' table allows filtering by the TODO-status.

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.

Show previous results in test results page gh#538

On a tests result page there is a tab for "Next & previous results" showing the result of test runs in the same scenario. This shows next and previous builds as well as test runs in the same build. This way you can easily check and compare results from before including any comments, labels, bug references (see next section). This helps to answer questions like "Is this a new issue", "Is it reproducible", "has it been seen in before", "how does the history look like".

Querying the database for former test runs of the same scenario is a rather costly operation which we do not want to do for multiple test results at once but only for each individual test result (1:1 relation). This is why this is done in each individual test result and not for a complete build.

Related issue: #10212

Screenshot of the feature:

Results for opensuse-Tumbleweed-NET-x86_64-Build20190620-update_Leap_42.1_gnome@64

Result: **failed** finished about 3 hours ago (30:16 minutes)
Clone of 964587 Cloned as 965220
Assigned worker: openqaworker1:6

| Details | Logs & Assets | Settings | Comments (0) | Next & previous results |

ext & previous results for *opensuse-Tumbleweed-NET-x86_64-update_Leap_42.1_g*
test job for this scenario)

ow 10 ▾ entries

	Result			Build
L	🔵			20190620
C	🔴	logs_from_installation_system	+1	20190620
	🔴	await_install		20190620

Link to latest in scenario name gh#836

Find the always latest job in a scenario with the link after the scenario name in the tab "Next & previous results" Screenshot:

| Details | Logs & Assets | Settings | Comments (0) | Next & previous results |

lext & previous results for *opensuse-Tumbleweed-DVD-x86_64-kde@64bit* (latest job for this scenario)

how 10 ▾ entries

Add `latest' query route gh#815

Should always refer to most recent job for the specified scenario.

- have the same link for test development, i.e. if one retriggers tests, the person has to always update the URL. If there would be a static URL even the browser can be instructed to reload the page automatically
- for linking to the always current execution of the last job within one scenario, e.g. to respond faster to the standard question in bug reports "does this bug still happen?"

Examples:

- `tests/latest?distri=opensuse&version=13.1&flavor=DVD&arch=x86_64&test=kde&machine=64bit`
- `tests/latest?flavor=DVD&arch=x86_64&test=kde`
- `tests/latest?test=foobar` - this searches for the most recent job using test_suite `foobar'

covering all distri, version, flavor, arch, machines. To be more specific, add the other query entries.

Allow group overview query by result gh#531

This allows e.g. to show only failed builds. Could be included like in http://lists.opensuse.org/ opensuse-factory/2016-02/msg00018.html for "known defects".

Example: Add query parameters like `···&result=failed&arch=x86_64` to show only failed for the single architecture selected.

Add web UI controls to select more builds in group_overview gh#804

The query parameter `limit_builds' allows to show more than the default 10 builds on demand. Just like we have for configuring previous results, the current commit adds web UI selections to reload the same page with higher number of builds on demand. For this, the limit of days is increased to show more builds but still limited by the selected number.

Example screenshot:

More query parameters for configuring last builds gh#575

By using advanced query parameters in the URLs you can configure the search for builds. Higher numbers would yield more complex database queries but can be selected for special investigation use cases with the advanced query parameters, e.g. if one wants to get an overview of a longer history. This applies to both the index dashboard and group overview page.

Example to show up to three week old builds instead of the default two weeks with up to 20 builds instead of up to 10 being the default for the group overview page:

```
http://openqa/group_overview/1?time_limit_days=21&limit_builds=20
```

Web UI controls to filter only tagged or all builds gh#807

Using a new query parameter `only_tagged=[0|1]' the list can be filtered, e.g. show only tagged (important) builds.

Example screenshot:

Related issue: #11052

Test result badges gh#5022

For each job result including the latest job result page, there is a corresponding route to get an SVG status badge that can eg. be used to build a status dashboard or for showing the status within a GitHub comment.


```
http://openqa/tests/123/badge
http://openqa/tests/latest/badge
```

There is an optional parameter 'show_build=1' that will prefix the status with the build number.

Carry over of bug references from previous jobs in same scenario

Many test failures within the same scenario might be due to the same reason. To avoid human reviewers having to add the same bug references again and again, bug references are carried over from previous failures in the same scenario if a job fails. The same behaviour can be achieved by adding `flag:carryover` to a comment. This idea is inspired by the Claim plugin for Jenkins.

| NOTE | The carry-over feature works on test module level. Only if the same set of test module as in a predecessor job fails the latest bug reference is carried over. |

NOTE	The lookup-depth is limited. The search for candidates will also stop early if too many different kinds of failures were seen. Checkout the descriptions of the relevant settings in the `carry_over` section of `openqa.ini` for details.
NOTE	For an approach to add bug references based on a search expression found in the job reason for incomplete jobs or job logs consider to Enable custom hook scripts on "job done" based on result.

Pinning comments as group description

This is possible by adding the keyword `pinned-description` anywhere in a comment on the group overview page. Then the comment will be shown at the top of the group overview page. However, it only works as operator or admin.

Dark mode

A dark mode theme can be enabled via "Appearance" settings for all logged in users. It can either be forced with the "dark mode" setting, or left to browser detection. Switching automatically between light and dark mode is natively supported by most modern browsers and can also be controlled manually via flags:

- On Firefox, go to `about:preferences#general` and search for "Website appearance".
- On Chrome, go to `chrome://flags/` and search for "Dark mode".

For more information, see developer.mozilla.org/CSS/@media/prefers-color-scheme

Developer mode

The developer mode allows to:

- Create or update needles from `assert_screen` mismatches ("re-needling")
- Pause the test execution (at a certain module) for manual investigation of the SUT

It can be accessed via the "Live View" tab of a running test. Only registered users can take control over tests. Basic instructions and buttons providing further information about the different options are already contained on the web page itself. So I am not repeating that information here and rather explain the overall workflow.

In case the developer mode in not working on your instance, try to follow the steps for debugging the developer mode under 'Pitfalls'.

Workflow for creating or updating needles

1. In case a new needles should be created, add the corresponding `assert_screen` calls to your test.
2. Start the test with the `assert_screen` calls which are supposed to fail.
3. Select "`assert_screen` timeout" under "Pause on screen mismatch" and confirm.

4. Wait until the test has paused. There is a button to skip the current timeout to speed this up.

5. A button for accessing the needle editor should occur. It may take a few seconds till it occurs because the screenshots created so far need to be uploaded from the worker to the web UI. Of course it is also possible to go back to the "Details" tab to create a new needle from any previous screenshot/match available.

6. After creating the new needle, click the resume button to test whether it worked.

Steps 4. to 6. can be repeated for further needles without restarting the test.

Job group editor gh#2111

Scenarios are defined as part of a job group. The `Edit job group` button exposes the editor.

YAML job templates editor

Settings can be specified as a key/value pair for each scenario. There is no equivalent in the table view so you need to migrate groups to use this feature.

Any settings specified on test suites, machines or products are also used and can still be modified independently. However, the YAML document should be updated before renaming or deleting test suites, products or machines used by it, otherwise that would create an inconsistent state.

Job groups can be updated through the YAML editor or the YAML-related REST API routes.

Deprecated: Table-based (pre-migration)

In old versions openQA had a table-based UI for defining job templates, listed in a table per medium. Machines can be added by selecting the architecture column and picking a machine from the list. Remove scenarios by removing all of their machines. Add new scenarios via the blue Plus icon at the top of the table. Changes to the priority are applied immediately.

If job groups still exist showing the old mode, the `Edit YAML` button can be used to reveal the YAML editor and migrate a group. After saving for the first time, the group can only be configured in YAML. The table view will not be shown anymore.

Note that making a backup before migrating groups may be a good idea, for example using `openqa-dump-templates`.

To migrate an old job group using the API the current schedule can be retrieved in YAML format and sent back to save as a complete YAML document. For example for all job groups in the old format:

```
for i in $(ssh openqa.example.com "sudo -u geekotest psql --no-align --tuples-only
--command=\"select id from job_groups where template is null order by id;\" openqa") ;
do
    curl -s http://openqa.example.com/api/v1/job_templates_scheduling/$i | openqa-cli
api --host http://openqa.example.com -X POST job_templates_scheduling/$i
schema=JobTemplates-01.yaml template="$(cat -)"
done
```

Note that in some cases you might run into errors where old test suites or products have invalid names which the old editor did not enforce:

Product names may not contain `:` or `@` characters. Something like `Server-DVD-Staging:A` would require replacing the `:` with eg. a `-`.

Test suites may not contain `:` or `@` characters. A test suite such as `ext4_uefi@staging` would have been allowed previously. The use of the `@` as a suffix could be replaced with a `-` or if it is used for variants of the same test suite with different settings, settings can be specified in YAML directly.

More generally the regular expression `[A-Za-z0-9._*-]+` could be used to check if a name is allowed for a product or test suite.

Configuring job groups via YAML documents

A new job group starts out empty, which in YAML means that the two mandatory sections are present but contain nothing. This is what can be seen when editing a completely group, and what is also the state to revert to before deleting a job group that is no longer useful:

```
products: {}
scenarios: {}
```

A job group is comprised of up to three main sections. `products` defines one or more mediums to run the scenarios in the group. At least one needs to be specified to be able to run tests. Going by an example of openSUSE 15.1 the name, distri, flavor and version could be written like so. Note that the version is a string in single quotes.

```
products:
  opensuse-15.1-DVD-Updates-x86_64:
    distri: opensuse
    flavor: DVD-Updates
    version: '15.1'
```

To complete the job group at least one scenario has to be added. A scenario is a combination of a test suite, a machine and an architecture. Scenarios must also be unique across job groups - trying to add it to multiple job groups is an error. Case in point, `textmode` and `gnome` could be defined like so:

```
scenarios:
  x86_64:
    opensuse-15.1-DVD-Updates-x86_64:
    - textmode
    - gnome:
      machine: uefi
      priority: 70
      settings:
        QEMUVGA: cirrus
```

Defaults

Now there are two scenarios for `x86_64`, one by giving just the name of the test suite and another which has a machine, priority and settings. Both are allowed. However since at least one scenario relies on defaults those need to be specified once in their own section:

```
defaults:
  x86_64:
    machine: 64bit
    priority: 50
```

The defaults section is only required whenever a scenario is not completely defined in-place. When it is used, the available parameters are identical to those for a single scenario. For instance the example could be amended to use settings and run every test suite for that architecture on several machines by default.

```
defaults:
  x86_64:
    machine: [64bit, 32bit]
    priority: 50
    settings:
      FOO: '1'
```

Defaults are always overwritten by explicit parameters on scenarios. Further more, all settings can be specified in YAML. Using this together with custom job template names, variants of a scenario can even be specified when they would normally be considered duplicated:

```
scenarios:
  x86_64:
    opensuse-15.1-DVD-Updates-x86_64:
    - textmode
    - gnome:
      machine: uefi
      priority: 70
      settings:
        QEMUVGA: cirrus
    - gnome_staging:
      testsuite: gnome
      machine: [32bit, 64bit-staging]
      settings:
        FOO: '2'
```

YAML Aliases

Even more flexibility can be achieved by using aliases in YAML, or in other words re-using a scenario by reference, such as to run the same scenarios in two different mediums. & is used to define an anchor, while * is the alias referencing the anchor:

```yaml
products:
  opensuse-15.1-DVD-Updates-x86_64:
    distri: opensuse
    flavor: DVD-Updates
    version: '15.1'
  opensuse-15.2-GNOME-Live-x86_64:
    distri: opensuse
    flavor: GNOME-Live
    version: '15.2'
scenarios:
  x86_64:
    opensuse-15.1-DVD-Updates-x86_64:
    - textmode
    - gnome: &gnome
      machine: uefi
      priority: 70
      settings:
        QEMUVGA: cirrus
    - gnome_staging: &gnome_staging
      testsuite: gnome
      machine: [32bit, 64bit-staging]
      settings:
        FOO: '2'
    opensuse-15.2-GNOME-Live-x86_64:
    - textmode
    - gnome: *gnome
    - gnome_staging: *gnome_staging
```

YAML Merge Keys

Also YAML Merge Keys are supported. This way you can reuse previously defined anchors and add other values to it. Values in the merged alias will be overridden.

You can even merge more than one alias.

```yaml
products:
  opensuse-15.1-DVD-Updates-x86_64:
    distri: opensuse
    flavor: DVD-Updates
    version: '15.1'
  opensuse-15.2-GNOME-Live-x86_64:
    distri: opensuse
    flavor: GNOME-Live
    version: '15.2'
scenarios:
  x86_64:
    opensuse-15.1-DVD-Updates-x86_64:
    - textmode
    - gnome:
      machine: uefi
      priority: 70
      settings: &common1
        QEMUVGA: cirrus
        FOO: default foo
    - gnome:
      machine: [32bit, 64bit-staging]
      priority: 70
      settings: &common2
        QEMUVGA: cirrus
        FOO: default foo
        BAR: default bar
    - gnome_staging:
      testsuite: gnome
      machine: [32bit, 64bit-staging]
      settings:
        # Merge
        <<: *common1
        FOO: foo # overrides the value from the merge keys
    - gnome_staging:
      testsuite: gnome
      machine: [32bit, 64bit-staging]
      settings:
        # Merge
        <<: [*common1, *common2] # *common1 overrides *common2
        FOO: foo # overrides the value from the merge keys
```

General YAML documentation

The job templates are written in YAML 1.2. In YAML, strings usually do not have to be quoted, except if it is a special value that would be loaded as a Boolean, NULL or Number. The following table shows all special values (See the documentation for the default YAML 1.2 Core Schema for more information).

Type	Special Values
`bool`	`true	True	TRUE	false	False	FALSE`
`int (Base 8)`	`0o7, 0o10, 0o755` Regular Expression: `0o [0-7]+`
`int (Base 10)`	`23, 42`, `0123`, `-314`` Regular Expression: `[-\]? [0-9]+`
`int (Base 16)`	`0xFF, 0xa, 0xc0ffee` Regular Expression: `0x [0-9a-fA-F]+`
`float (Number)`	`3.14, 3.14`, `-3.14`, `3.3e+3`, `3.3e3`, `.14`, `001.23`, `.3E-1`, `3e3`` Regular Expression: `[-\]? (\. [0-9]+	[0-9]+ (\. [0-9]*)?) ([eE] [-+]? [0-9]+)?`
`float (Infinity)`	`.inf, .inf`, `-.inf`, `.Inf`` etc. Regular Expression: `[-]? \. ( inf | Inf | INF )`
`float (Not a number)`	`.nan, .NaN, .NAN` Regular Expression: `\. ( nan | NaN |NAN )`
`null`	`null	Null	NULL	~	# empty`
`str`	everything else

Because we are using the Merge Keys feature, also the unquoted string `<<` is special. If you need the literal string `<<` (for example as a value in the job settings), you have to quote it.

Use of the REST API

openQA includes a *client* script which - depending on the distribution - is packaged independently to allow interfacing with an existing openQA instance without needing to install openQA itself. Call `openqa-cli --help` for help. The sub-commands provide further help, e.g. `openqa-cli api --help` contains a lot of examples.

This section focuses on particular API use-cases. Checkout the openQA client section for further information about the client itself, how authentication works and how plain `curl` can be used.

Finding tests

The following example lists all jobs within the job group with the ID `1` and the setting `BUILD=20210707` on `openqa.opensuse.org`:

```
curl -s "https://openqa.opensuse.org/api/v1/jobs?groupid=1&build=20210707" | jq
```

The tool `jq` is used for pretty-printing in this example but it is also useful for additional filtering (see js's tutorial).

However, openQA's API provides many more filters on its own. These can be used by adding additional query parameters, e.g.:

- `ids`/`state`/`result`: Return only jobs with matching ID/state/result. Multiple IDs/states/results can be specified by repeating the parameter or by passing comma-separated values.

- `distri`/`version`/`build`/`test`/`arch`/`machine` /`worker_class`/`iso`/`hdd_1`: Return only jobs where the job settings match the specified values like in the example above. Note that it is not possible to filter by arbitrary job settings although this list might not be complete.

- `groupid`/`group`: Return only jobs within the job group with the specified ID/name like in the example above. These parameters are mutually exclusive, `groupid` has precedence.

- `latest=1`: De-duplicates, so that for the same `DISTRI`, `VERSION`, `BUILD`, `TEST`, `FLAVOR`, `ARCH` and `MACHINE` only the latest job is returned.

- `limit`/`page`: Limit the number of returned jobs and allow pagination, e.g. `page=2&limit=10` would only show results 11-20.

- `modules`/`modules_result`: Return only jobs which have a test module with the specified name/result.

- `before`/`after`: Return only jobs with a job ID less/greater than the specified job ID.

- `scope=current`: Returns only jobs which have not been cloned yet.

- `scope=relevant`: Returns only jobs which have not been obsoleted yet and which have not been cloned yet. Clones which are still pending do **not** count.

Remarks

- All parameters can be combined with each other unless stated otherwise.

- When specifying the same parameter multiple times, only the last occurrence is taken into account.

- All values are matched exactly, so e.g. `group=openSUSE+Leap+15` returns only jobs within the group `openSUSE Leap 15` but not jobs from the group `openSUSE Leap 15 ARM`. This applies to parameters for filtering job settings as well.

Triggering tests

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

Cloning existing jobs - openqa-clone-job

If one wants to recreate an existing job from any publicly available openQA instance the script `openqa-clone-job` 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 resources can be found, for example a worker with matching `WORKER_CLASS` must be registered. More details on `openqa-clone-job` 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.

Further examples for advanced dependency handling

It is possible to spawn a single set of jobs using just one API call, e.g.:

```
openqa-cli api -X POST jobs TEST:0=first-job TEST:1=second-job _START_AFTER:1=0
```

The suffixes `0` and `1` are actually freely chosen and are merely used to specify which parameters belong to which job and how they depend on each other.

This creates a job with `TEST=first-job` and one with `TEST=second-job` and the second job will be started after the first. Of course other types of dependencies are possible as well (via `_PARALLEL` and `_START_DIRECTLY_AFTER`). Note that this kind of call will return the resulting job ID for each suffix that has been used, e.g.:

```
{"ids":{"0":2531,"1":2530}}
```

To use colons within a settings key, just add a trailing `:`, e.g.:

```
openqa-cli api -X POST jobs TEST=test KEY:WITH:COLONS:=example
```

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 settings for media, job groups, machines and test suites jobs are triggered based on template matching. These settings need to be defined before on the corresponding pages of the web UI (accessible to operators from the user menu). The section on job templates already explains details about these tables. Alternatively, these settings can be supplied via a YAML document.

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

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

_DEPRIORITIZEBUILD
Setting this switch to '1' will deprioritize the unfinished jobs of old builds, and it will obsolete the jobs once the configurable limit of the priority value is reached.

_DEPRIORITIZE_LIMIT
The configurable limit of priority value up to which jobs should be deprioritized. Needs `_DEPRIORITIZEBUILD`. Defaults to 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. Needs `_OBSOLETE`.

_SKIP_CHAINED_DEPS
Do not schedule parent test suites which are specified in `START_AFTER_TEST` or `START_DIRECTLY_AFTER_TEST`.

_GROUP
Job templates **not** matching the given group name are ignored. Does **not** affect obsoletion behavior.

_GROUP_ID	Same as `_GROUP` but allows to specify the group directly by ID.
_PRIORITY	Sets the priority value for the new jobs (which otherwise defaults to the priority of the job template)
__...	All parameters starting with `__` will **not** be added as job settings. Those parameters can be used to store additional information about the scheduled product itself, e.g. the URL of a web page with more context.

Example for `_DEPRIORITIZEBUILD` and `_DEPRIORITIZE_LIMIT`.

```
openqa-cli api -X POST isos async=0 ISO=my_iso.iso DISTRI=my_distri \
        FLAVOR=sweet ARCH=my_arch VERSION=42 BUILD=1234 \
        _DEPRIORITIZEBUILD=1 _DEPRIORITIZE_LIMIT=120 \
```

NOTE By default scheduling products is done synchronously within the requests, corresponding to the parameter `async=0`. Use `async=1` to avoid possible timeouts by performing the task in background. This is recommended on big instances but means that the results (and possible errors) need to be polled via `openqa-cli api isos/$scheduled_product_id`.

Statistical investigation

In case issues appear sporadically and are therefore hard to reproduce it can help to trigger many more jobs on a production instance to gather more data first, for example the failure ratio.

Example of triggering 50 jobs in a development group so that the result of passed/failed jobs is counted by openQA itself on the corresponding overview page:

```
openqa-clone-job --skip-chained-deps --repeat=50 --within-instance \
https://openqa.opensuse.org 123456  BUILD=poo32242_investigation \
_GROUP="Test Development:openSUSE Tumbleweed"
```

To get an overview about the fail ratio and confidence interval of sporadically failing applications you can also use a script like this.

Defining test scenarios in YAML

Instead of relying on the tables for machines, mediums/products, test suites and job templates of the openQA instance, one can provide these definitions/settings also via a YAML document. This YAML document could be specific to a certain test distribution and stored in the same repository as those tests (making the versioning easier).

WARNING	This feature is still experimental and may change in an incompatible way in future versions.

This YAML document can be specified via the scheduling parameter `SCENARIO_DEFINITIONS_YAML`:

```
openqa-cli api … -X POST isos --param-file SCENARIO_DEFINITIONS_YAML=/local/file.yaml
…
```

This command will upload the contents of the local file `/local/file.yaml` to a possibly remote openQA instance. The YAML document will only be used within the scope of this particular API request. No settings are stored/altered on the openQA instance.

If the YAML document already exists on the openQA host, you can also use `SCENARIO_DEFINITIONS_YAML_FILE` which expects the file path of the YAML document on the openQA host.

The YAML document itself should define at least one product, machine and job template like this:

```
job_templates:
  create_hdd:
    settings:
      PUBLISH_HDD_1: 'example-%VERSION%-%ARCH%-%BUILD%@%MACHINE%.qcow2'
  boot_from_hdd:
    settings:
      HDD_1: 'example-%VERSION%-%ARCH%-%BUILD%@%MACHINE%.qcow2'
      START_AFTER_TEST: 'create_hdd'
      WORKER_CLASS: 'job-specific-class'
```

These definitions are used like their openQA-instance-wide counterparts (so continue reading the next section for more details on job templates).

Remarks

When scheduling a single test (variable `TEST` is specified) attempts to obsolete/deprioritize are prevented by default because this is likely not wanted. Use `_FORCE_OBSOLETE` or `_FORCE_DEPRIORITIZEBUILD` to nevertheless obsolete/deprioritize **all** jobs with matching `DISTRI`, `VERSION`, `FLAVOR` and `ARCH`.

Job template YAML

Job groups can be queried via the experimental REST API:

```
api/v1/experimental/job_templates_scheduling
```

The GET request will get the YAML for one or multiple groups while a POST request conversely updates the YAML for a particular group.

Two scripts using these routes can be used to import and export YAML templates:

```
openqa-dump-templates --json --group test > test.json
```

```
openqa-load-templates test.json
```

Asset handling

Multiple parameters exist to reference "assets" to be used by tests. "Assets" are essentially content that is stored by the openQA web-UI and provided to the workers. Things that are typically assets include the ISOs and other images that are tested, for example.

Some assets can also be produced by a job, sent back to the web-UI, and used by a later job (see explanation of 'storing' and 'publishing' assets, below). Assets can also be seen in the web-UI and downloaded directly (though there is a configuration option to hide some or all asset types from public view in the web-UI).

Assets may be shared between the web-UI and the workers by having them literally use a shared filesystem (this used to be the only option), or by having the workers download them from the server when needed and cache them locally. Checkout the documentation about asset caching for more on this.

Specifying assets required by a job

The following job settings are specifying that an asset is required by a job:

- `ISO` (type `iso`)
- `ISO_n` (type `iso`)
- `HDD_n` (type `hdd`)
- `UEFI_PFLASH_VARS` (type `hdd`) (in some cases, see below)
- `REPO_n` (type `repo`)
- `ASSET_n` (type `other`)
- `KERNEL` (type `other`)
- `INITRD` (type `other`)

Where you see e.g. `ISO_n`, that means `ISO_1`, `ISO_2` etc. will all be treated as assets.

The values of the above parameters are expected to be the name of a file - or, in the case of `REPO_n`, a directory - that exists under the path `/var/lib/openqa/share/factory` on the openQA web-UI. That path has subdirectories for each of the asset types, and the file or directory must be in the correct subdirectory, so e.g. the file for an asset `HDD_1` must be under `/var/lib/openqa/share/factory/hdd`. You may create a subdirectory called `fixed` for any asset type and place assets there (e.g. in `/var/lib/openqa/share/factory/hdd/fixed` for `hdd`-type assets): this exempts them from the automatic cleanup described in the section about asset cleanup. Non-fixed assets are always subject to the cleanup.

`UEFI_PFLASH_VARS` is a special case: whether it is treated as an asset depends on the value. If the value looks like an absolute path (starts with `/`), it will not be treated as an asset (and so the value should be an absolute path for a file which exists on the relevant worker system(s)). Otherwise, it is treated as an `hdd`-type asset. This allows tests to use a stock base image (like the ones provided by edk2) for a simple case, but also allows a job to upload its image on completion - including any

changes made to the UEFI variables during the execution of the job - for use by a child job which needs to inherit those changes.

You can also use special suffixes to the basic parameter forms to access some special handling for assets.

The following suffixes exist:

_URL	Before starting these jobs, try to download these assets into the relevant asset directory of the openQA web-UI from trusted domains specified in `/etc/openqa/openqa.ini`. For e.g., `ISO_1_URL=http://trusted.com/foo.iso` would, if `trusted.com` is set as a trusted domain, cause openQA to download the file `foo.iso` to `/var/lib/openqa/share/factory/iso` and set `ISO_1=foo.iso`. If you set both `ISO_1` and `ISO_1_URL`, the file pointed to by `ISO_1_URL` will be downloaded and renamed to the name set as `ISO_1`.
_DECOMPRESS_URL	Specify a compressed asset to be downloaded that will be uncompressed by openQA. For e.g. `ISO_1_DECOMPRESS_URL=http://host/foo2.iso.xz` will download the file `foo2.iso.xz`, uncompress it to `foo2.iso`, store it in `/var/lib/openqa/share/factory/iso` and set `ISO_1=foo2.iso`. Again, you can also set `ISO_1` to change the name the file will be downloaded and uncompressed as.

Specifying assets created by a job

Jobs can upload assets to the web-UI so other jobs can used them as `HDD_n` and `UEFI_PFLASH_VARS` assets as described in the previous section.

To declare an asset to be uploaded, you can use the job settings `PUBLISH_HDD_n` and `PUBLISH_PFLASH_VARS`. For instance, if you specify `PUBLISH_HDD_1=updated.qcow2`, the `HDD_1` disk image as it exists at the end of the test will be uploaded back to the web-UI and stored under the name `updated.qcow2`. Any other job can then specify `HDD_1=updated.qcow2` to use this published image as its `HDD_1`.

IMPORTANT	Assets that are already existing will be overridden. If the same asset is uploaded by multiple jobs concurrently this will lead to file corruption. So be sure to use unique names or use private assets as explained in the subsection below.
NOTE	Note that assets are by default only uploaded if the job completes successfully. To force publishing assets even in case of a failed job one can specify the `FORCE_PUBLISH_HDD_` variable.
NOTE	When using this mechanism you will often also want to use the variable expansion mechanism.

Private assets

There is a mechanism to alter an asset's file name automatically to associate it with the particular job that produced it (currently, by prepending the job ID to the filename). To make use of it, use `STORE_HDD_n` (instead of `PUBLISH_HDD_n`). Those assets can then be consumed by chained jobs. For instance, if a parent job uploads an asset via `STORE_HDD_1=somename.qcow2`, its children can use it via `HDD_1=somename.qcow2` without having to worry about naming conflicts.

IMPORTANT	This only works if the jobs uploading and consuming jobs have a chained dependency. For more on "chained" jobs, see the documentation of job dependencies.
NOTE	Access to private assets is not protected. Theoretically, jobs outside the chain can still access the asset by explicitly prepending the ID of the creating job.

Cleanup of assets, results and other data

The cleanup of assets, test results and certain other data is automated. That means openQA removes assets, job results and other data automatically according to configurable limits.

All cleanup jobs run within the Minion job queue, normally provided by `openqa-gru.service`. The dashboard for Minion jobs is accessible via the administrator menu in the web UI. Only one cleanup job can run at the same time unless `concurrent` is set to `1` in the `[cleanup]` settings of `openqa.ini`. Many other cleanup-related settings can be found within `openqa.ini` as well, e.g. the `[···_limits]` sections contain various tweaks and allow to change certain defaults. Checkout the sub section Timers and triggers to learn more about how those jobs are triggered.

The cleanup of **assets** and job **results** (and certain other data) is happening independently of each other using different strategies and retention settings:

- The further sub sections provide an overall description of the **asset** cleanup strategy and how to configure it.

- The Basic cleanup settings section explains how to configure retentions, covering the job **result** cleanup as well. Also have a look at Build tagging which allows to keep certain jobs longer by marking them as important.

- The Auditing section explains the cleanup of the audit log.

Cleanup strategy for assets

To find out whether an asset should be removed, openQA determines by which 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 job group. If that job group is within a parent job group, the asset is considered part of that parent job group.

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

While an asset might belong to multiple groups it is only **accounted** to the group with the highest asset limit which has still enough room to hold that asset. That basically mean that an asset is never counted twice.

If the size limit for assets of a group is exceeded, openQA will remove assets which belong 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 have been 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 unless the files are still being updated. Keep in mind that this behavior is also enabled on local instances and affects all cloned jobs (unless cloned into a job group).

If an asset is just a symlink then only the symlink is cleaned up (but not the file or directory it points to).

'Fixed' assets - those placed in the `fixed` subdirectory of the relevant asset directory - are counted against the group size limit, but are never cleaned up. This is intended for things like base disk images which must always be available for a test to work. Note that relative symlinks in the regular assets directory that point into the `fixed` subdirectory are also preserved.

Configuring limit for assets within job 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.

The default size limit for job groups can be adjusted in the `default_group_limits` section of the openQA config file.

Configuring 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.

In less trivial cases where a common limit is not enough or certain assets need more fine-grained control, patterns based on the filename can be used. The patterns are interpreted as Perl regular expressions and if a pattern matches the basename of an asset the specified duration in days will be used. In simple cases the pattern is just a match on a word.

Consider the following examples to specify custom limits that would match assets with the names `testrepo-latest` and `openSUSE-12.3-x86_64.iso`.

```
[assets/storage_duration]
latest = 30
openSUSE.+x86_64 = 10
```

Note that modifications to the file will count against the limit, so if an asset was updated within the timespan it will not be removed.

Timers and triggers

Cleanup can be triggered in different ways. One option is to use `minion_task_triggers` and specify tasks via `on_job_done`. Another way to do that is to use the systemd timers `openqa-enqueue-*-cleanup` to periodically run tasks. Both can be used separately or in combination.

The relevant Minion tasks are:

- limit_assets

- limit_audit_events

- limit_bugs

- limit_results_and_logs

- limit_screenshots

These are no-ops if a task is already running so they can safely be enqueued repeatedly. Note that the tasks can still take considerable time computing what to delete, from seconds to minutes. The tasks can be enabled in the corresponding config file section.

Disabling cleanup

By default the cleanup is enabled with systemd timers if available. To completely disable cleanup make sure that no minion cleanup tasks are enabled over the config file and prevent individual or all cleanup systemd timers, for example for the asset cleanup:

```
systemctl mask openqa-enqueue-asset-cleanup.timer
```

CLI interface

Beside the `daemon` argument to run the actual web service the openQA startup script `/usr/share/openqa/script/openqa` supports further arguments.

For a full list of those commands, just invoke `/usr/share/openqa/script/openqa -h`. This also works for sub-commands(e.g. `/usr/share/openqa/script/openqa minion -h`, `/usr/share/openqa/script/openqa minion job -h`).

Note that `prefork` is only supported for the main web service but not for other services like the live view handler.

Suggested workflow for test review

If an openQA instance is only used by one or few individuals often no strict process needs to defined how openQA tests should be reviewed and how individual results should be handled. If the group of test reviewers grows openQA and the ecosystem around openQA offer some helpful features and approaches.

In particular for a big user base it is important to formalize how decisions are made and how tasks are delegated. For this structured comments on the openQA platform can be used. With a comment on openQA in the right format one can make a decision, inform automatic tools at the same time as other users and have a traceable documentation of the actions taken.

- In openQA parent job groups can be defined with multiple job groups. This allows to segment tests for scopes of individual review teams. The parent job group overview pages as well as the central index page of openQA show "bullet list" icons that bring you directly to a combined test overview showing results from all sub groups. This allows to have queries ready like https://openqa.opensuse.org/tests/overview?groupid=1&groupid=2&groupid=3 which show all openQA test failures within the hierarchy of test results. This can be combined with the flag "todo=1" (click the "TODO" checkbox in the filter box on test overview pages) to show only tests that need review. Other combinations of queries are possible, e.g. https://openqa.opensuse.org/tests/overview?build=my-build&todo=1 to show all test results that need review for build "my-build"

- https://github.com/os-autoinst/openqa_review can be used to produce multiple different generated reports, e.g. all tests that need review, tests that are linked to closed bugs, etc.

- Use auto-review to handle flaky issues and even automatically retrigger according tests

- In case of known sporadic issues that can not be fixed quickly consider automatic retries of jobs http://open.qa/docs/#_automatic_retries_of_jobs

- In case of known non-sporadic test issues that can not be fixed quickly consider overwriting the result of jobs http://open.qa/docs/#_overwrite_result_of_job

- For the SUSE maintenance test workflows a "branding" specific approach is provided: In case of needing to urgently release individual maintenance updates before test failures can be resolved consider instructing qem-bot, the automation validating and approving release requests based on openQA test results, to ignore individual job failures for specific incidents. See https://progress.opensuse.org/issues/95479#Suggestions for the necessary comment format or use the comment template from the openqa.suse.de comment edit window.

Where to now?

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

openQA test 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 an openQA test and the API available. Tests are written in the **Perl** programming language. However there is support for the **Python** programming language (through the Perl module `Inline::Python`).

Some basic but no in-depth knowledge of Perl or Python is needed. This document assumes that the reader is already familiar with Perl or Python.

Test API

os-autoinst provides the API for the test using the os-autoinst backend. Take a look at the test API documentation for further information. Note that this test API is sometimes also referred to as an openQA DSL, because in some contexts it can look like a domain specific language.

How to write tests

Test module interface

An openQA test needs to implement at least the `run` subroutine containing the actual test code and the test needs to be loaded in the distribution's `main.pm`.

Here is an example in Perl:

```perl
use Mojo::Base "basetest";
use testapi;

sub run () {
    # write in this block the code for your test.
}
```

- **Note:** `Mojo::Base` *automatically enables: strict, warnings, utf8, perl5.16. See* [*Mojo::Base Description*](#)

And here is an example in Python:

```python
from testapi import *

def run(self):
    # write in this block the code for your test.
```

There are more optional subroutines that can be defined to extend the behavior of a test. A test must comply with the interface defined below. *Please note that the subroutine marked with* *1 *are optional.*

```python
# Written in type-hinted python to indicate explicitly return types
def run(self): -> None
def test_flags(): -> dict # *1
def post_fail_hook(): -> None # *1
def pre_run_hook(): -> None # *1
def post_run_hook(): -> None # *1
```

run

Defines the actual steps to be performed during the module execution.

An example usage:

```
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;
}
```

`assert_screen` & `send_key` are provided by [os-autoinst](#).

test_flags

Specifies what should happen when test execution of the current test module is finished depending on the result.

Each flag is defined with a hash key, the possible hash keys are:

- **fatal**: When set to 1 the whole test suite is aborted if the test module fails. The overall state is set to `failed`.

- **ignore_failure**: When set to 1 and the test module fails, it will not affect the overall result at all.

- **milestone**: After this test succeeds, update the 'lastgood' snapshot of the SUT.

- **no_rollback**: Don't roll back to the 'lastgood' snapshot of the SUT if the test module fails.

- **always_rollback**: Roll back to the 'lastgood' snapshot of the SUT even if test was successful.

See the example below for how to enable a test flag. Note that snapshots are only supported by the QEMU backend. When using other backends `fatal` is therefore enabled by default. One can explicitly set it to `0` to disable the behavior for all backends even though it is not possible to roll back.

An example usage:

```
sub test_flags () {
    return {fatal => 1};
}
```

pre_run_hook

It is called before the run function - mainly useful for a whole group of tests. It is useful to setup the start point of the test.

An example usage:

```perl
sub pre_run_hook () {
    # Make sure to begin the test in the root console.
    select_console 'root-console';
}
```

post_fail_hook

It is called after `run()` failed. It is useful to upload log files or to determine the state of the machine.

An example usage:

```perl
sub post_fail_hook () {
    # Take an screenshot when the test failed
    save_screenshot;
}
```

post_run_hook

It is called after `run()` regardless of the result of the test run.

An example usage:

```perl
sub post_fail_hook () {
    send_key 'ctrl-alt-f3';

    assert_script_run 'openqa-cli api experimental/search q=shutdown.pm' ;
}
```

Notes on the Python API

The Python integration that OpenQA offers through `Inline::Python` also allows the test modules to import other Perl modules with the usage of the `perl` virtual package provided by `Inline::Python`.

Because of the way `Inline::Python` binds Perl functions to Python it is not possible to use keywords arguments from Python to Perl functions. They must be passed as positional arguments, for example `"key", "value"`.

See the following snippet of Perl code

```perl
use x11utils;

# [...] omitted for brevity

sub run () {
    # [...] omitted for brevity

    # Start vncviewer - notice the named arguments
    x11_start_program('vncviewer :0',
        target_match => 'virtman-gnome_virt-install',
        match_timeout => 100
    );
    # [...] omitted for brevity
}
```

versus the equivalent python code:

```python
from testapi import *

# [...] omitted for brevity

def run(self):
    perl.require('x11utils')

    # Start vncviewer - notice the named arguments passed as positional arguments
    # Formatted in pairs for better visibility.

    perl.x11utils.x11_start_program('vncviewer :0',
        'target_match', 'virtman-gnome_virt-install',
        'match_timeout', 100
    )
    # [...] omitted for brevity
```

Additionally, Python tests do not support `run_args`. An error will be present when a Python test detects the presence of `run_args` on schedule.

This is because of the way `Inline::Python` handles argument passing between Perl <→ Python, references to complex Perl objects do not reach Python properly and they can't be used.

Example Perl test modules

The following examples are short complete test modules written in Perl implementing the interface described above.

Boot to desktop

Boots into desktop when pressing enter at the boot loader screen.

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 Mojo::Base "basetest";
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 () {
    return {fatal => 1};
}
```

Install software via `zypper`

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';
}
```

Sample X11 Test

Example: Typical X11 test testing kate

```perl
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';
}
```

Example Python test modules

The following examples are short complete test modules written in Python implementing the interface described above.

openQA web UI sample test

Example: Test for the openQA web UI written in Python

```python
from testapi import *

def run(self):
    assert_screen('openqa-logged-in')
    assert_and_click('openqa-search')
    type_string('shutdown.pm')
    send_key('ret')
    assert_screen('openqa-search-results')

    # import further Perl-based libraries (besides `testapi`)
    perl.require('x11utils')

    # use imported Perl-based libraries; call Perl function that would be called via
"named arguments" in Perl
    # note: In Perl the call would have been: x11_start_program('flatpak run
com.obsproject.Studio', target_match => 'obsproject-wizard')
    #
    # See the explanation in the "Notes on the Python API" section.
    perl.x11utils.x11_start_program('flatpak run com.obsproject.Studio',
'target_match', 'obsproject-wizard')

def switch_to_root_console():
    send_key('ctrl-alt-f3')

def post_fail_hook(self):
    switch_to_root_console()
    assert_script_run('openqa-cli api experimental/search q=shutdown.pm')

def test_flags(self):
    return {'fatal': 1}
```

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

Changing timeouts

By default, tests are aborted after two hours by the worker. To change this limit, set the test variable `MAX_JOB_TIME` to the desired number of seconds.

The download of assets, synchronization of tests and other setup tasks do **not** count into `MAX_JOB_TIME`. However, the setup time is limited by default to one hour. This can be changed by setting `MAX_SETUP_TIME`.

To save disk space, increasing `MAX_JOB_TIME` beyond the default will automatically disable the video by adding `NOVIDEO=1` to the test settings. This can be prevented by adding `NOVIDEO=0` explicitly.

The variable `TIMEOUT_SCALE` allows to scale `MAX_JOB_TIME` and timeouts within the backend, for example the test API. This is supposed to be set within the worker settings on slow worker hosts. It has no influence on the video setting.

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 does not make sense to continue the test run even if the current test module does not have the fatal flag, use `fatal` as serial failure type, so all subsequent test modules will not 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 `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

```perl
$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

```perl
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

```perl
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 the job variable WORKER_CLASS taking a comma-separated list of worker class values. The values are combined from multiple places where defined. Typically machines and test suite configurations set the worker class. Jobs with this variable set are assigned only to workers, which have all corresponding worker class values in their configuration (and-combination).

For example, the following configuration ensures, that jobs with WORKER_CLASS=desktop can be assigned *only* to worker instances 1 and 2. Jobs with WORKER_CLASS=desktop,foo can only be assigned to worker instance 2 which has both the values desktop *and* foo:

File: workers.ini

```ini
[1]
WORKER_CLASS = desktop

[2]
WORKER_CLASS = desktop,foo,bar

[3]
# WORKER_CLASS is not set
```

Worker class values can also be set to additionally qualify workers or worker instances for

informational purposes, for example region and location tags based on company conventions:

File: workers.ini

```
[global]
WORKER_CLASS = planet-earth,continent-antarctica,location-my_station
```

Running a custom worker engine

By default the openQA workers run the "isotovideo" application from PATH on the worker host, that is in most cases isotovideo. A custom worker engine command can be set with the test variable ISOTOVIDEO. For example to run isotovideo from a custom container image one could use the test variable setting ISOTOVIDEO=podman run --pull=always --rm -it registry.example.org/my/container/isotovideo /usr/bin/isotovideo -d

Automatic retries of jobs

You might encounter flaky openQA tests that fail sporadically. The best way to address flaky test code is of course to fix the test code itself. For example, if certain steps rely on external components over network, retries within the test modules should be applied.

However, there can still be cases where you might want openQA to automatically retrigger jobs. This can be achieved by setting the test variable RETRY in the format <retries>[:<description>] to an integer value with the maximum number of retries and an optional description string separated by a colon. For example triggering an openQA job with the variable RETRY=2:bug#42 will retrigger an openQA test on failure up to 2 totalling to up to 3 jobs. Note that the retry jobs are scheduled immediately and will be executed as soon as possible depending on available worker slots. Many factors can change in retries impacting the reproducibility, e.g. the used worker host and instance, any network related content, etc. By default openQA tests do not retry. The optional, additional description string is used only for reference and has no functional impact.

See Automatic cloning of incomplete jobs for an additional solution intended for administrators handling known issues causing incomplete jobs.

Custom hook scripts on "job done" based on result can be used to apply more elaborate issue detection and retriggering of tests.

Job dependencies

There are different dependency **types**, most importantly *chained* and *parallel* dependencies.

A dependency is always between two jobs where one of the jobs is the *parent* and one the *child*. The concept of parent and child jobs is **orthogonal** to the concept of types.

A job can have multiple dependencies. So in conclusion, a job can have multiple children and multiple parents at the same time and each child/parent-relation can be of an arbitrary type.

Additionally, dependencies can be machine-specific (see Inter-machine dependencies section).

Declaring dependencies

Dependencies are declared by adding a job setting on the child job specifying its parents. There is one variable for each dependency type.

When starting jobs based on templates the relevant settings are `START_AFTER_TEST`, `START_DIRECTLY_AFTER_TEST` and `PARALLEL_WITH`. Details are explained for the different dependency types specifically in the subsequent sections. Generally, if declaring a dependency does not work as expected, be sure to check the "scheduled product" for the jobs (which is linked on the info box of the details page of any created job).

When starting a single set of new jobs, the dependencies must be declared as explained in the Further examples for advanced dependency handling section. The variables mentioned in the subsequent sections do **not** apply.

Chained dependencies

Chained dependencies declare that one test must only run after another test has concluded. For instance, extra tests relying on a successfully finished installation should declare a chained dependency on the installation test.

There are also *directly-chained* dependencies. They are similar to *chained* dependencies but are strictly a distinct type. The difference between *chained* and *directly-chained* dependencies is that directly-chained means the tests must run directly after another on the same worker slot. This can be useful to test efficiently on bare metal SUTs and other self-provisioning environments.

Tests that are waiting for their *chained* parents to finish are shown as "blocked" in the web UI. Tests that are waiting for their *directly-chained* parents to finish are shown as "assigned" in the web UI.

To declare a *chained* dependency add the variable `START_AFTER_TEST` with the name(s) of test suite(s) after which the selected test suite is supposed to run. Use a comma-separated list for multiple test suite dependencies, e.g. `START_AFTER_TEST="kde,dhcp-server"`.

To declare a *directly-chained* dependency add the variable `START_DIRECTLY_AFTER_TEST`. It works in the same way as for *chained* dependencies. Mismatching worker classes between jobs to run in direct sequence on the same worker are considered an error.

NOTE	The set of all jobs that have direct or indirect *directly-chained* dependencies between each other is sometimes called a *directly-chained cluster*. All jobs within the cluster will be assigned to a single worker-slot at the same time by the scheduler.

Parallel dependencies

Parallel dependencies declare that tests must be scheduled to run at the same time. An example are "multi-machine tests" which usually test some kind of server and multiple clients. In this example the client tests should declare a parallel dependency on the server tests.

To declare a *parallel* dependency, use the `PARALLEL_WITH` variable with the name(s) of test suite(s) that need other test suite(s) to run at the same time. In other words, `PARALLEL_WITH` declares "I need this test suite to be running during my run". Use a comma separated list for multiple test suite

dependencies (e.g. `PARALLEL_WITH="web-server,dhcp-server"`).

Keep in mind that the parent job *must be running until all children finish*. Otherwise the scheduler will cancel child jobs once parent is done.

NOTE	The set of all jobs that have direct or indirect *parallel* dependencies between each other is sometimes called a *parallel cluster*. The scheduler can only assign these jobs if there is a sufficient number of free worker-slots. To avoid a parallel cluster from starvation its priority is increased gradually and eventually workers can be held back for the cluster.

Inter-machine dependencies

Those dependencies make it possible to create job dependencies between tests which are supposed to run on different machines.

To use it, simply append the machine name for each dependent test suite with an `@` sign separated. If a machine is not explicitly defined, the variable `MACHINE` of the current job is used for the dependent test suite.

Example 1:

```
START_AFTER_TEST="kde@64bit-1G,dhcp-server@64bit-8G"
```

Example 2:

```
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 following example sections to get a better understanding.

Handling of related jobs on failure / cancellation / restart

openQA tries to handle things sensibly when jobs with dependencies either fail, or are manually cancelled or restarted:

- When a chained or parallel parent fails or is cancelled, all children will be cancelled.
- When a parent is restarted, all children are also restarted recursively.
- When a parallel child is restarted, the parent and siblings will also be restarted.
- When a **regularly** chained child is restarted, the parent is only restarted if it failed. This will usually be fine, but be aware that if an asset uploaded by the chained parent has been cleaned up, the child may fail immediately. To deal with this case, just restart the parent to recreate the asset.
- When a **directly** chained child is restarted, all directly chained parents are recursively

restarted (but not directly chained siblings). Otherwise it would not be possible to guarantee that the jobs run directly after each other on the same worker.

- When a parallel **child** fails or is cancelled, the parent and all other children are also cancelled. This behaviour is intended for closely-related clusters of jobs, e.g. high availability tests, where it's sensible to assume the entire test is invalid if any of its components fails. A special variable can be used to change this behaviour. Setting a parallel parent job's PARALLEL_CANCEL_WHOLE_CLUSTER to a false value, i.e. 0, changes this so that, if one of its children fails or is cancelled but the parent has other pending or active children, the parent and the other children will not be cancelled. This behaviour makes more sense if the parent is providing services to the various children but the children themselves are not closely related and a failure of one does not imply that the tests run by the other children and the parent are invalid.

Further notes

- The API also allows to skip restarting parents via `skip_parents=1` and to skip restarting children via `skip_children=1`. It is also possible to skip restarting only passed and softfailed children via `skip_ok_result_children=1`.

- Restarting multiple directly chained children individually is not possible because the parent would be restarted twice which is not possible. So one needs to restart the parent job instead. Use the mentioned `skip_ok_result_children=1` to restart only jobs which are not ok

Handling of dependencies when cloning jobs

Be sure to have ready the job dependencies section to have an understanding of different dependency types and the distinction between parents and children.

When cloning a job via `openqa-clone-job`, parent jobs are cloned as well by default, regardless of the type. Use `--skip-deps` to avoid cloning parent jobs. Use `--skip-chained-deps` to avoid cloning parents of the types `CHAINED` and `DIRECTLY_CHAINED`.

When cloning a job via `openqa-clone-job`, child jobs of the type `PARALLEL` are cloned by default. Use `--clone-children` to clone child jobs of other types as well. By default, only direct children are considered (regardless of the type). Use `--max-depth` to specify a higher depth (`0` denotes infinity). Be aware that this affects siblings as well when cloning parents (as explained in the previous paragraph).

As a consequence it makes a difference which job of the dependency tree is cloned, especially with default parameters. Examples:

- Cloning a *chained child* (e.g. an "extra" test) will clone its parents (e.g. an "installation" test) as well but **not** vice versa.

- To clone a parallel cluster, the *parallel parent* should be cloned (e.g. the "server" test). When cloning a parallel child, only *that* child and the parent will be cloned but not the siblings (e.g. the other "client" tests).

Examples

Specify machine explicitly

Assume there is a test suite `A` supposed to run on machine `64bit-8G`. Additionally, test suite `B` supposed to run on machine `64bit-1G`.

That means test suite `B` needs the variable `START_AFTER_TEST=A@64bit-8G`. This results in the following dependency:

```
A@64bit-8G --> B@64bit-1G
```

Implicitly inherit machines from parent

Assume test suite `A` is supposed to run on the machines `64bit` and `ppc`. Additionally, test suite `B` is supposed to run on both of these machines as well. This can be achieved by simply adding the variable `START_AFTER_TEST=A` to test suite `B` (omitting the machine at all). openQA take the best matches. This results in the following dependencies:

```
A@64bit --> B@64bit
A@ppc --> B@ppc
```

Conflicting machines prevent inheritance from parent

Assume test suite `A` is supposed to run on machine `64bit-8G`. Additionally, test suite `B` is supposed to run on machine `64bit-1G`.

Adding the variable `START_AFTER_TEST=A` to test suite `B` will **not** work. That means openQA will **not** create a job dependency and instead shows an error message. So it is required to explicitly define the variable as `START_AFTER_TEST=A@64bit-8G` in that case.

Consider a different example: Assume test suite `A` is supposed to run on the machines `ppc`, `64bit` and `s390x`. Additionally, there are 3 testsuites `B` on `ppc-1G`, `C` on `ppc-2G` and `D` on `ppc64le`.

Adding the variable `PARALLEL_WITH=A@ppc` to the test suites `B`, `C` and `D` will result in the following dependencies:

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

openQA will also show errors that test suite `A` is not necessary on the machines `64bit` and `s390x`.

Implicitly creating a dependency on same machine

Assume the value of the variable `START_AFTER_TEST` or `PARALLEL_WITH` **only** contains a test suite name but no machine (e.g. `START_AFTER_TEST=A,B` or `PARALLEL_WITH=A,B`).

In this case openQA will create job dependencies that are scheduled on the same machine if all test suites are placed on the same machine.

Notes regarding directly chained dependencies

Having multiple jobs with `START_DIRECTLY_AFTER_TEST` pointing to the same parent job is possible, e.g.:

```
    --> B --> C
  /
 A
  \
    --> D --> E
```

Of course only either `B` or `D` jobs can really be started **directly** after `A`. However, the use of `START_DIRECTLY_AFTER_TEST` still makes sure that no completely different job is executed in the middle and of course that all of these jobs are executed on the same worker.

The directly chained sub-trees are executed in alphabetical order. So the above tree would result in the following execution order: `A, B, C, D, E`.

If `A` fails, none of the other jobs are attempted to be executed. If `B` fails, `C` is not attempted to be executed but `D` and `E` are. The assumption is that the average error case does not leave the system in a completely broken state and possibly required cleanup is done in the post fail hook.

Directly chained dependencies and regularly chained dependencies can be mixed. This allows to create a dependency tree which contains multiple directly chained sub-trees. Be aware that these sub-trees might be executed on **different** workers and depending on the tree even be executed in parallel.

Worker requirements

`CHAINED` and `DIRECTLY_CHAINED` dependencies require only one worker. `PARALLEL` dependencies on the other hand require as many free workers as jobs are present in the parallel cluster.

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 and 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.
```

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 parallel dependencies between individual jobs (which are explained in the previous section). For MM tests specifically, also take note of the following remarks:

- *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.

- The 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 (according to the specified job dependencies).

- openQA does *not* automatically synchronize individual steps of the tests.

- openQA provides a *locking server for basic synchronization* of tests (e.g. wait until services are ready for failover). The correct usage of these locks is the responsibility of the test writer (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)

Test synchronization and locking API

openQA provides a locking API. To use it in your test files import the lockapi package (*use lockapi;*). It provides the following functions: mutex_create, mutex_lock, mutex_unlock, mutex_wait

Each of these functions takes the name of the mutex lock as first parameter. The name must not

contain the "-" character. Mutex locks are associated with the caller's job.

`mutex_lock` tries to lock the mutex for the caller's job. The `mutex_lock` call blocks if the mutex does not exist or has been locked by a different job.

`mutex_unlock` tries to unlock the mutex. If the mutex is locked by a different job, `mutex_unlock` call blocks until the lock becomes available. If the mutex does not exist the call returns immediately without doing anything.

`mutex_wait` is a combination of `mutex_lock` and `mutex_unlock`. It displays more information about mutex state (time spent waiting, location of the lock). Use it if you need to wait for a specific action from single place (e.g. that Apache is running on the master node).

`mutex_create` creates a new mutex which is initially unlocked. If the mutex already exists the call returns immediately without doing anything.

Mutexes are addressed by *their name*. Each cluster of parallel jobs (defined via `PARALLEL_WITH` dependencies) has its own namespace. That means concurrently running jobs in different parallel job clusters use distinct mutexes (even if the same names are used).

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 garanting 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 a certain action from the child or sibling job rather than the parent. In this case the 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 is required to pass the mutex owner's job ID as a second parameter to `mutex_lock` and `mutex_wait`. The mutex owner is the job that creates the mutex. If a child job with a given ID has already finished, `mutex_lock` calls die. The job ID is also required when unlocking such a mutex.

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

```perl
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 the 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 events from a single job. When we need multiple jobs to reach a certain state we need to use barriers.

To create a barrier call `barrier_create` with the parameters name and count. The name serves as an ID (same as with mutexes). The count parameter specifies the number of jobs needed to call `barrier_wait` to unlock barrier.

There is an optional `barrier_wait` parameter called `check_dead_job`. When used it will kill all jobs waiting in `barrier_wait` if one of the cluster jobs dies. It prevents waiting for states that will never be reached (and eventually dies on job timeout). It should be set only on one of the `barrier_wait` calls.

An example would be one master and three worker jobs and you want to do initial setup in the three worker jobs before starting main actions. In such a case you might use `check_dead_job` to avoid useless actions when one of the worker jobs dies.

Example of barriers: Check for dead jobs while waiting for barrier

```perl
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 that 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

Example of mmapi: Getting info about parents / children

```perl
use Mojo::Base "basetest";
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

```perl
use Mojo::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 referrer 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.

```perl
use 5.018;
use Mojo::Base 'opensusebasetest';
use testapi;
use utils;

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

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.

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

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

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 (it is enabled by default for os-autoinst-distri- opensuse tests). The svirt backend uses the `SERIAL_CONSOLE` variable, but only on s390x machines it has been confirmed to be working (failing on Hyper-V, VMware and XEN, see poo#55985).

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.

> **NOTE** for os-autoinst-distri-opensuse tests instead of using `select_console('root-virtio-terminal')` directly is the preferred way to use wrapper `select_serial_terminal()`, which handles all backends:

```
# Selects a virtio based serial terminal if available or fallback to the best suitable
console
# for the current backend.
select_serial_terminal();
```

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_serial_terminal()) {
        # 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.

```perl
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/serial_screen.pm (used also by the svirt serial console). 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 serial_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

Trigger new tests by modifying settings from existing test runs

To trigger new tests with custom settings the command line client `openqa-cli` can be used. To trigger new tests relying on all settings from existing tests runs but modifying specific settings the `openqa-clone-job` script can be used. Within the openQA repository the script is located at `/usr/share/openqa/script/`. This tool can be used to create a new job that adds, removes or changes settings.

This example adds or overrides `FOO` to be `bar`, removes `BAZ` and appends `:PR-123` to `TEST`:

```
openqa-clone-job --from localhost --host localhost 42 FOO=bar BAZ= TEST+=:PR-123
```

NOTE	When cloning children via `--clone-children` as well, the children are also affected. Parent jobs (which are cloned as well by default) are *not* affected unless the `--parental-inheritance` flag is used.

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.:

```
openqa-clone-job --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.:

```
openqa-clone-job --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. If the worker(s) are being started via the systemd unit, then this can achieved by using the `openqa-worker-no-cleanup@.service` unit instead of `openqa-worker@.service`.
- 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:

```
openqa-clone-job --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, `openqa-clone-job` comes handy here:

```
openqa-clone-job --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.

```
openqa-clone-job --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:

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

Defining a custom test schedule or custom test modules

Normally the test schedule, that is which test modules should be executed and which order, is prescribed by the `main.pm` file within the test distribution. Additionally it is possible to exclude certain test modules from execution using the os-autoinst test variables `INCLUDE_MODULES` and `EXCLUDE_MODULES`. A custom schedule can be defined using the test variable `SCHEDULE`. Also test modules can be defined and overridden on-the-fly using a downloadable asset. For example for the common test distribution os-autoinst-distri-opensuse one could use `SCHEDULE=tests/boot/boot_to_desktop,tests/console/my_test` for a much faster test execution that can boot an existing system and only execute the intended test module.

https://github.com/os-autoinst/os-autoinst/blob/master/doc/backend_vars.asciidoc describes in detail the mentioned test parameters and more. Please consult this full reference as well.

EXCLUDE_MODULES

If a job has the following schedule:

- boot/boot_to_desktop

- console/systemd_testsuite

- console/docker

The module console/docker can be excluded with:

```
openqa-clone-job --from https://openqa.opensuse.org --host https://openqa.opensuse.org
24 EXCLUDE_MODULES=docker
```

The schedule would be:

- boot/boot_to_desktop

- console/systemd_testsuite

> **NOTE**   Excluding modules that are not scheduled does not raise an error.

INCLUDE_MODULES

If a job has the following schedule:

- boot/boot_to_desktop

- console/systemd_testsuite

- console/docker

The module console/docker can be excluded with:

```
openqa-clone-job --from https://openqa.opensuse.org --host https://openqa.opensuse.org
24 INCLUDE_MODULES=boot_to_desktop,systemd_testsuite
```

The schedule would be:

- boot/boot_to_desktop

- console/systemd_testsuite

> **NOTE**   Including modules that are not scheduled does not raise an error, but they are not scheduled.

SCHEDULE

Additionally it is possible to define a custom schedule using the test variable `SCHEDULE`.

```
openqa-clone-job --from https://openqa.opensuse.org --host https://openqa.opensuse.org
24 SCHEDULE=tests/boot/boot_to_desktop,tests/console/consoletest_setup
```

> **NOTE**   Any existing test module within **CASEDIR** can be scheduled.

SCHEDULE + ASSET_<NR>_URL

Test modules can be defined and overridden on-the-fly using a downloadable asset (combining **ASSET_<NR>_URL** and **SCHEDULE**).

For example one can schedule a job on a production instance with a custom schedule consisting of two modules from the provided test distribution plus one test module which is defined dynamically and downloaded as an asset from an external trusted download domain:

```
openqa-clone-job --from https://openqa.opensuse.org --host https://openqa.opensuse.org
24 SCHEDULE=tests/boot/boot_to_desktop,tests/console/consoletest_setup,foo,bar
ASSET_1_URL=https://example.org/my/test/bar.pm
ASSET_2_URL=https://example.org/my/test/foo.pm
```

NOTE	The asset number doesn't affect the schedule order. The test modules foo.pm and bar.pm will be downloaded into the root of the pool directory where tests and assets are used by isotovideo. For this reason, to schedule them, no path is needed.

A valid test module format looks like this:

```perl
use Mojo::Base 'consoletest';
use testapi;

sub run () {
    select_console 'root-console';
    assert_script_run 'foo';
}

sub post_run_hook () {}
```

For example this can be used in bug investigations or trying out new test modules which are hard to test locally. The section "Asset handling" in the Users Guide describes how downloadable assets can be specified. It is important to note that the specified asset is only downloaded once. New versions must be supplied as new, unambiguous download target file names.

Triggering tests based on an any remote Git refspec or open GitHub pull request

openQA also supports to trigger tests using test code from an open pull request on GitHub or any branch or Git refspec. That means that code changes that are not yet available on a production instance of openQA can be tested safely to ensure the code changes work as expected before merging the code into a production repository and branch. This works by setting the CASEDIR parameter of os-autoinst to a valid Git repository path including an optional branch/refspec specifier. It is also possible to set NEEDLES_DIR to a valid Git repository path to use custom needles. See https://github.com/os-autoinst/os-autoinst/blob/master/doc/backend_vars.asciidoc for details.

NOTE

The openQA worker normally default-initializes `CASEDIR` and `NEEDLES_DIR` to point to default repositories provided by the openQA instance. This behavior interacts with specifying a custom `CASEDIR` or `NEEDLES_DIR` in the following way:

- If `CASEDIR` or `NEEDLES_DIR` is customized the customized location is used instead of the default repository.

- If only one of `CASEDIR` or `NEEDLES_DIR` is customized the other variable will still be initialized to point to the default repository.

- A relative `NEEDLES_DIR` is treated to be relative to the default `CASEDIR` (even if `CASEDIR` is customized). To have it treated to be relative to the custom `CASEDIR`, prefix the relative path with `%CASEDIR%/`. So specifying e.g. `CASEDIR=https://github.com/⋯` and `NEEDLES_DIR=%%CASEDIR%%/the-needles` will lead to `%CASEDIR%` being substituted with the path of the Git checkout created for the custom `CASEDIR`. That results in needles found in https://github.com/.../tree/.../the-needles to be used. Note that double `%`-signs are to avoid variable substitution. When using `curl`, you need to escape the `%`-sign as `%25` **in addition**.

A helper script `openqa-clone-custom-git-refspec` is available for convenience that supports some combinations.

To clone one job within a remote instance based on an open github pull request the following syntax can be used:

```
openqa-clone-custom-git-refspec $GITHUB_PR_URL $OPENQA_TEST_URL
```

For example:

```
openqa-clone-custom-git-refspec https://github.com/os-autoinst/os-autoinst-distri-
opensuse/pull/6649 https://openqa.opensuse.org/tests/839191
```

Note that customizing `CASEDIR` does **not** mean needles will be loaded from there, even if the repository specified as `CASEDIR` contains needles. To load needles from that repository, it needs to be specified as `NEEDLES_DIR` as well.

Keep in mind that if `PRODUCTDIR` is overwritten as well, it might not relate to the state of the specified git refspec that is passed via the command line parameter to `openqa-clone-custom-git-refspec` or via the `PRODUCTDIR` variable to `openqa-clone-job`. Both can still be used when overwriting `PRODUCTDIR`, but special care must be taken if the schedule is modified (then it is safer to manually specify the schedule via the `SCHEDULE` variable).

Running openQA jobs as CI checks

It is possible to run openQA jobs as CI checks of a repository, e.g. a test distribution or an arbitrary repository containing software with openQA tests as part of the test suite.

Create and monitor openQA jobs from within the CI runner

The easiest approach is to create and monitor openQA jobs from within the CI runner. To make this easier, `openqa-cli` provides the `schedule` sub-command with the `--monitor` flag. This way you still need an openQA instance to run tests (as they are not executed within the CI runner itself) but you can also still conveniently view the test results on the openQA web UI.

An example using GitHub actions and the official container image we provide for `openqa-cli` can be found in the example distributions' workflow.

NOTE	This example makes use of the `SCENARIO_DEFINITIONS_YAML` variable which allows specifying scenario definitions in a way that is independent from openQA's normal scheduling tables. This feature is explained in further detail in the corresponding users guide section.
NOTE	This example shows how API credentials are supplied. It is important to note that using `on:pull_request` would only work for PRs created on the main repository but not for PRs created from forks. Therefore `on:pull_request_target` is used instead. To still run the tests on the PR version the variables under `github.event.pull_request.head.*` are utilized (instead of e.g. just `$GITHUB_REF`).
NOTE	Due to the use of `on:pull_request_target` the scenario definitions are read from the main repository in this example. This is the conservative approach. To allow scheduling jobs based on the PR version of the scenario definitions file one could use e.g. `SCENARIO_DEFINITIONS_YAML_FILE=https://raw.githubusercontent.com/$GH_REPO/$GH_REF/.github/workflows/openqa.yml` instead of `- uses: actions/checkout@v3` and `--param-file SCENARIO_DEFINITIONS_YAML=scenario-definitions.yaml`.

Use webhooks and status reporting APIs of GitHub

This approach is so far specific to GitHub and is a bit more effort to setup than the approach mentioned in the previous section. For this to work, GitHub needs to be able to inform openQA that a PR has been created or updated and openQA needs to be able to inform GitHub about the result of the jobs it ran. So authentication needs to be configured on both sides. On the upside, there is no additional CI runner required and the authentication also works when a PR is created from a fork repository branch which extra configuration.

The test scenarios for your repository need to be defined in the file `scenario-definitions.yaml` at the root of your repository. Checkout the scenario definitions from the example distribution for an

example. You may append a parameter like `SCENARIO_DEFINITIONS_YAML=path/of/yaml` to the query parameters of the webhook to change the lookup path of this file.

Setup a GitHub access token for openQA

This setup is required for openQA to be able to report the status back to GitHub.

1. Open https://github.com/settings/tokens/new and create a new token. It needs at least the scope "repo".

2. Open the openQA web UI's config file (usually `/etc/openqa/openqa.ini`) and add the token created in the previous step:

   ```
   [secrets]
   github_token = $token
   ```

3. Restart the web UI services.

IMPORTANT	The user the token has been created with needs at least "Write" permissions to access the repository the CI checks should appear on (for instance by being member of a team with that permissions). Otherwise, GitHub might respond with a 404 response (weirdly not necessarily 403) when submitting the CI check status.

Setup webhook on GitHub

This setup is required for GitHub to be able to inform openQA that a PR has been created or updated.

1. Open https://github.com/$orga/$project/settings/hooks/new. You need to substitute the placeholders `$orga` and `$project` with the corresponding value of the repository you want to add CI checks to.

2. Add https://$user:$apikey:$apisecret@$openqa_host/api/v1/webhooks/product?DISTRI=example&VERSION=0&FLAVOR=DVD&ARCH=x86_64&TEST=simple_boot as "Payload URL". You need to substitute the placeholders with valid API credentials and hostname for your openQA instance. If you don't have an API key/secret then you can create one on https://$openqa_host/api_keys. Make sure the casing of the user name is correct. The scheduling parameters need to be adjusted to produce the wanted set of jobs from your scenario definitions YAML file.

3. Select "application/json" as "Content type".

4. Add `$user:$apikey:$apisecret` as secret replacing placeholders again. You need to use the same credentials as in step 2.

5. Keep SSL enabled. (Be sure your openQA instance is reachable via HTTPS.)

6. Select "Let me select individual events." and then "Pull requests".

7. Ensure "Active" is checked and confirm.

8. GitHub should now have been delivering a "ping" event. Checkout whether it could be delivered. If you have gotten a 200 response then everything is setup correctly. Otherwise, checkout the response of the delivery to investigate what is wrong.

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('XUnit','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:

```perl
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 client

There are two ways to interact with openQA as a user. The web UI and the REST API. In this guide we will focus on the latter. You've probably already seen a few examples of its use with `openqa-cli` earlier in the documentation.

Here we will start again from the very beginning to give you a more complete overview of its capabilities. To get started all you need is an openQA instance with a few jobs and `curl`. Just replace `openqa.example.com` in the examples below with the hostname of your openQA instance.

```
curl http://openqa.example.com/api/v1/jobs/overview
```

That one-liner will show you the latest jobs from the overview in JSON format. You could also append various query parameters to filter the jobs further.

```
curl http://openqa.example.com/api/v1/jobs/overview?result=failed
```

But using `curl` directly can also get a bit clunky when the data you need to submit is more complex, you want to store host and authentication information in config files, or just get the returned JSON pretty printed.

For those cases openQA also contains a dedicated client to help you with that. It is called `openqa-cli` and can usually be installed with an `openQA-client` package (the name will vary depending on your Linux distribution).

```
openqa-cli api --host http://openqa.example.com jobs/overview result=failed
```

Our example above is quickly translated. The `api` subcommand of `openqa-cli` allows you to perform arbitrary HTTP requests against the REST API. The path will automatically get the correct version prefix applied (such as `/api/v1`), and query parameters can be passed along as `key=value` pairs.

Help

The `api` subcommand is not the only one available and more will be added over time. To get a complete list of all currently available subcommands you can use the `--help` option.

```
openqa-cli --help
```

And each subcommand also contains descriptions for all its available options, as well as many common usage examples.

```
openqa-cli api --help
```

Authentication

Not all REST endpoints are public, many will return a `403 Forbidden` error if you try to access them without proper credentials. The credentials (or API keys) are managed in the web UI, to which you will need operator access.

Once you have acquired a valid key and secret you can store them in a config file or use them ad-hoc from the command line. There are two config files `openqa-cli` will try, the global `/etc/openqa/client.conf`, and your personal `~/.config/openqa/client.conf`. The format is the same for both.

```
[openqa.example.com]
key = 1234567890ABCDEF
secret = ABCDEF1234567890
```

For ad-hoc use all `openqa-cli` subcommands use the `--apikey` and `--apisecret` options. Which will override whatever the config files may contain.

```
openqa-cli api --host http://openqa.example.com --apikey 1234567890ABCDEF \
    --apisecret ABCDEF1234567890 -X POST jobs/2/comments text=hello
```

Personal access token

The authentication mechanism used by `openqa-cli` was specifically designed to allow secure access to the REST API even via unencrypted HTTP connections. But when your openQA server has been deployed with HTTPS (and for HTTP connections originating from localhost) you can also use plain old Basic authentication with a personal access token. That allows for almost any HTTP client to be used with openQA.

This access token is made up of your username, and the same key/secret combo the `openqa-cli` authentication mechanism uses. All you have to do is combine them as `USERNAME:KEY:SECRET` and you can use `curl` to access operator and admin REST endpoints (depending on user privileges of course).

```
curl -u arthur:1234567890ABCDEF:ABCDEF1234567890 -X DELETE \
    https://openqa.example.com/api/v1/assets/1
```

Features

Many of the `openqa-cli api` features are designed to be similar to other commonly used tools like `curl`. It helps a lot if you are already familiar with the HTTP protocol and JSON. Both will be referenced extensively.

HTTP Methods

The `--method` option (or `-X` for short) allows you to change the HTTP request method from `GET` to something else. In the openQA API you will most commonly encounter `POST`, `PUT` and `DELETE`. For example to start testing a new ISO image you would use `POST`.

```
openqa-cli api --host http://openqa.example.com -X POST isos \
    ISO=openSUSE-Factory-NET-x86_64-Build0053-Media.iso DISTRI=opensuse \
    VERSION=Factory FLAVOR=NET ARCH=x86_64 BUILD=0053
```

HTTP Headers

With the `--header` option (or `-a` for short) you can add one or more custom HTTP headers to your request. This feature is currently not used much, but can be handy if for example the REST endpoint you are using supports content negotiation.

```
openqa-cli api --host http://openqa.example.com -a 'Accept: application/json' \
    jobs/overview
```

HTTP Body

To change the HTTP request body there are multiple options available. The simplest being `--data` (or `-d` for short), which allows you to use a plain string as request body. This can be useful for example to change the group id of a job.

```
openqa-cli api --host http://openqa.example.com -X PUT jobs/1 \
    --data '{"group_id":2}'
```

With the `--data-file` option (or `-D` for short) you can also use a file instead.

```
openqa-cli api --host http://openqa.example.com -X PUT jobs/1 \
    --data-file ./test.json
```

Or just pipe the data to `openqa-cli`.

```
echo '{"group_id":2}' | openqa-cli api --host http://openqa.example.com -X PUT \
    jobs/1
```

Forms

Most data you pass to the openQA API will be key/value form parameters. Either in the query string, or encoded as `application/x-www-form-urlencoded` HTTP request body. But you don't have to worry about this too much, because `openqa-cli api` knows when to use which format automatically, you just provide the key/value pairs.

Form parameters are most commonly passed as additional arguments after the path. For example to post a comment to a job.

```
openqa-cli api --host http://openqa.example.com -X POST jobs/2/comments text=abc
```

This value can also be quoted to include whitespace characters.

```
openqa-cli api --host http://openqa.example.com -X POST jobs/2/comments \
    text="Hello openQA!"
```

And you can use interpolation to include files.

```
openqa-cli api --host http://openqa.example.com -X POST jobs/2/comments \
    text="$(cat ./comment.markdown)"
```

Alternatively you can also use the `--form` option (or `-f` for short) to provide all form parameters in JSON format. Here you would reuse the HTTP body options, such as `--data` and `--data-file`, to pass the JSON document to be turned into form parameters.

```
openqa-cli api --host http://openqa.example.com --form --data '{"text":"abc"}' \
    -X POST jobs/2/comments
```

JSON

The primary data exchange format in the openQA API is JSON. And you will even see error messages in JSON format most of the time.

```
{"error":"no api key","error_status":403}
```

By default the returned JSON is often compressed, for better performance, and can be hard to read if the response gets larger. But if you add the `--pretty` option (or `-p` for short), `openqa-cli` can reformat it for you.

```
openqa-cli api --host http://openqa.example.com --pretty jobs/overview
```

The JSON will be re-encoded with newlines and indentation for much better readability.

```
{
    "error" : "no api key",
    "error_status" : 403
}
```

The `--json` option (or `-j` for short) can be used to set a `Content-Type: application/json` request header. Whenever you need to upload a JSON document.

```
openqa-cli api --host http://openqa.example.com -X PUT jobs/1 --json \
    --data '{"group_id":2}'
```

Unicode

Just use a UTF-8 locale for your terminal and Unicode will pretty much just work.

```
openqa-cli api --host http://openqa.example.com -X POST jobs/2/comments \
    text="I ⬜ Unicode"
```

JSON documents are always expected to be UTF-8 encoded.

```
openqa-cli api --host http://openqa.example.com --form \
    --data '{"text":"I ⬜ Unicode"}' -X POST jobobs/407/comments \
    -X POST jobs/2/comments
```

Host shortcuts

Aside from the `--host` option, there are also a few shortcuts available. If you leave out the `--host` option completely, the default value will be http://localhost, which is very convenient for debugging purposes.

```
openqa-cli api jobs/overview
```

And organisations that contribute to openQA and are invested in the project can also get their very own personalised shortcuts. Currently we have `--osd` for http://openqa.suse.de, and `--o3` for openqa.opensuse.org.

```
openqa-cli api --o3 jobs/overview
```

Debugging

Often times just seeing the HTTP response body might not be enough to debug a problem. With the `--verbose` option (or `-v` for short) you can also get additional information printed.

```
openqa-cli api --host http://openqa.example.com --verbose -X POST \
    jobs/407/comments text="Hello openQA!"
```

This includes the HTTP response status line, as well as headers.

```
HTTP/1.1 403 Forbidden
Content-Type: application/json;charset=UTF-8
Strict-Transport-Security: max-age=31536000; includeSubDomains
Server: Mojolicious (Perl)
Content-Length: 41
Date: Wed, 29 Apr 2020 12:03:11 GMT

{"error":"no api key","error_status":403}
```

And if that is not enough, you can experiment with the `MOJO_CLIENT_DEBUG` environment variable.

```
MOJO_CLIENT_DEBUG=1 openqa-cli api --host http://openqa.example.com -X POST \
    jobs/407/comments text="Hello openQA!"
```

It will activate a debug feature in the Mojolicious framework, which openQA uses internally, and show everything that is being sent or received.

```
POST /api/v1/jobs/407/comments HTTP/1.1
Content-Length: 20
User-Agent: Mojolicious (Perl)
Content-Type: application/x-www-form-urlencoded
Host: openqa.example.com
X-API-Microtime: 1588153057
X-API-Hash: 8a73f6c37920921d52a8b5352ab417d923ee979e
Accept-Encoding: gzip
X-API-Key: AAEAC3E147A1EEE0
Accept: application/json

text=Hello+openQA%21
```

Just be aware that this is a feature the openQA team does not control, and the exact output as well as how it escapes control characters will change a bit over time.

Archive mode

With the `archive` subcommand of `openqa-cli` you can download all the assets and test results of a job for archiving or debugging purposes.

```
openqa-cli archive --host http://openqa.example.com 408 /tmp/openqa_job_408
```

Thumbnails are not included by default, but can be added with the `--with-thumbnails` option (or `-t` for short).

```
openqa-cli archive --host http://openqa.example.com --with-thumbnails \
    408 ~/openqa_job_408
```

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+rwx /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.

On some less powerful systems (like Raspberry Pi), reducing screenshots size with optipng may take significant amount of time. In this case, you can switch to `pngquant` which is significantly faster, but uses lossy compression. To do that, install `pngquant` package and set `USE_PNGQUANT=1` in worker or job settings.

DB migration from SQlite to postgreSQL

As a first step to start using postgreSQL, please, configure postgreSQL database according to the postgreSQL 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 postgreSQL

```
# openqa is the postgreSQL 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 openqa-dump-templates and openqa-load-templates scripts accordingly.

Steps to debug developer mode setup

This is basically a checklist to go through in case the developer mode is broken in your setup (e.g. you are getting the error message `unable to upgrade ws to command server`):

1. Be sure to have everything up to date. That includes relevant packages on the machine hosting the web UI and on the worker.

2. Ensure the firewall configuration steps from https://github.com/os-autoinst/os-autoinst/blob/master/script/os-autoinst-setup-multi-machine have been applied to the worker machine

3. Check whether the web browser can reach the livehandler daemon. Go to a running test and open the live view. Then open the JavaScript console of the web browser. If it contains messages like `Received message via ws proxy: ⋯` the livehandler daemon can be reached. Otherwise, try the following sub-steps:

   a. The installation guide has been updated to cover the developer mode. In case you installed your instance before the developer mode has been introduced, make sure that the Apache module `rewrite` is enabled (via `a2enmod rewrite`). Also be sure the vhost configuration looks like the one found in the openQA Git repository (especially the part for the reverse proxies).

   b. Check whether `openqa-livehandler.service` is running. It is supposed to be run on the same machine as the web UI and should actually be started automatically as a dependency of `openqa-webui.service`.

4. Check whether the livehandler can reach the os-autoinst command server. Go to a running test and open the live view. Then open the JavaScript console of the web browser. If it contains messages like `Received message via ws proxy: {⋯,"type":"info","what":"cmdsrvmsg"}` the os-autoinst command server can be reached. Otherwise there should be at least a message like `Received message via ws proxy: {"what":"connecting to os-autoinst command server at ws:\/\/host:20053\/xhB84lUuPlMfhDEF\/ws",⋯}` which contains the URL the livehandler is attempting to query. In this case try the following sub-steps:

   a. If the host is wrong, add `WORKER_HOSTNAME = correcthost` to `workers.ini`. The worker should then tell the web UI that it is reachable via `correcthost` resulting in a correct URL for the os-autoinst command server. Be sure the setting appears after the `[global]` section header.

   b. It might also be the case that the firewall is blocking the HTTP/websocket connection on the required port. The required port is `QEMUPORT` plus 1. By default, `QEMUPORT` is set to `$worker_instance_number * 10 + 20002` by the worker. So to cover worker slots from 1 to 10 one would by default require the ports 20013, 20023, … and 20103.

5. It can also help to look at the architecture diagram which shows which component needs access to which other components and the ports used by default.

Networking in openQA

For tests using the QEMU backend the networking type used is controlled by the `NICTYPE` variable. If unset or empty `NICTYPE` defaults to `user`, i.e. 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.

Other backends can be treated just the same as bare-metal setups. Tests can be triggered in parallel same as for QEMU based ones and synchronization primitives can be used. For the physical network according separation needs to be ensured externally where needed as means for machines to be able to access each other.

QEMU User Networking

With QEMU user networking each jobs gets its 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 is `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 defined, it is 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 ...
...
```

The MAC address of each virtual NIC is controlled by the `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 as described in the next section.

Multi-machine test setup

The complete multi-machine test setup can be provided from the script `os-autoinst-setup-multi-machine` provided by "os-autoinst". The script can be also found online on https://github.com/os-autoinst/os-autoinst/blob/master/script/os-autoinst-setup-multi-machine

The configuration is applicable for openSUSE and will use *Open vSwitch* for virtual switch, *firewalld* (or *SuSEfirewall2* for older versions) for NAT and *wicked* or *NetworkManager* as network manager. Keep in mind that a firewall is not strictly necessary for operation. The operation without firewall is not covered in all necessary details in this documentation.

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

> **NOTE** Alternatively [salt-states-openqa](#) contains necessities to establish such a setup and configure it for all workers with the `tap` worker class. They also cover GRE tunnels (that are explained in the next section).

The script `os-autoinst-setup-multi-machine` can be run like this:

```
# specify the number of test VMs to run on this host
instances=30 bash -x $(which os-autoinst-setup-multi-machine)
```

What os-autoinst-setup-multi-machine does

Set up Open vSwitch

The script will install and configure Open vSwitch as well as a service called *os-autoinst-openvswitch.service*.

NOTE	*os-autoinst-openvswitch.service* is a support service that sets the vlan number of Open vSwitch ports based on `NICVLAN` variable - this separates the groups of tests from each other. The `NICVLAN` variable is dynamically assigned by the openQA scheduler.

The name of the bridge (default: `br1`) will be set in `/etc/sysconfig/os-autoinst-openvswitch`.

Configure virtual interfaces

The script will add the bridge device and the tap devices for every multi-machine worker instance.

NOTE	The bridge device will also call a script at `/etc/wicked/scripts/gre_tunnel_preup.sh` on *PRE_UP*. This script needs **manual** touch if you want to set up multiple multi-machine worker hosts. Refer to the GRE tunnels section below for further information.

Configure NAT with firewalld

The required firewall rules for masquerading (NAT) and zone configuration for the trusted zone will be set up. The bridge devices will be added to the zone. IP-Forwarding will be enabled.

```
# show the firewall configuration
firewall-cmd --list-all-zones
```

What is left to do after running os-autoinst-setup-multi-machine

GRE tunnels

By default all multi-machine workers have to be on a single physical machine. You can join multiple physical machines and its OVS bridges together by a 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 a gre_tunnel_preup script (change the `remote_ip` value correspondingly on both hosts):

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

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"
```

Ensure to make gre_tunnel_preup.sh executable.

NOTE	When using GRE tunnels keep in mind that virtual machines 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.

Configure openQA workers

Allow worker instances to run multi-machine jobs:

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

NOTE	The number of tap devices should correspond to the number of the running worker instances. For example, if you have set up 3 worker instances, the same number of tap devices should be configured.

Enable worker instances to be started on system boot:

```
systemctl enable openqa-worker@{1..3}
```

Verify the setup

Simply run a MM test scenario. For openSUSE, you can find many relevant tests on o3, e.g. look for networking-related tests like `ping_server`/`ping_client` or `wicked_basic_ref`/`wicked_basic_sut`.

To test GRE tunnels, you may want to change the jobs worker classes so the different jobs are

executed on different workers. So you could call `openqa-clone-job` like this:

```
openqa-clone-job \
    --skip-chained-deps \                                # assuming assets are present
    --max-depth 0 \                                      # clone the entire parallel cluster
    --within-instance                                    # create new jobs on the same
instance
    https://openqa.opensuse.org/tests/3886213 \          # arbitrary job in cluster to clone
    _GROUP=0 BUILD+=test-mm-setup \                       # avoid interfering with production
jobs
    WORKER_CLASS:wicked_basic_ref+=,worker_foo \ # ensure one job runs on `worker_foo`
    WORKER_CLASS:wicked_basic_sut+=,worker_bar   # ensure other job runs on
`worker_bar`
```

Also be sure to reboot the worker host to make sure the setup is actually persistent.

Start test VMs manually

You may also start VMs manually to verify the setup.

First, download a suitable image and launch a VM in the same way `os-autoinst` would do for MM jobs:

```
wget http://download.opensuse.org/tumbleweed/appliances/openSUSE-Tumbleweed-Minimal-
VM.x86_64-Cloud.qcow2
qemu-system-x86_64 -m 2048 -enable-kvm -vnc :42 -snapshot \
  -netdev tap,id=qanet0,ifname=tap40,script=no,downscript=no \
  -device virtio-net,netdev=qanet0,mac=52:54:00:13:0b:4a \
  openSUSE-Tumbleweed-Minimal-VM.x86_64-Cloud.qcow2
```

The image used here is of course just an example. You need to make sure to assign a unique MAC address (e.g. by adjusting the last two figures in the example; this will not conflict with MAC addresses used by os-autoinst) and use a tap device not used at the same time by a SUT-VM.

Within the VM configure the network **like** this (you may need to adjust concrete IP addresses, subnets and interface names):

```
ip link set dev eth0 up mtu 1380
ip a add dev eth0 10.0.2.15/24
ip r add default via 10.0.2.2
echo 'nameserver 8.8.8.8' > /etc/resolv.conf
```

The MTU is chosen in accordance with what the openSUSE test distribution uses for MM tests and should be below the MTU set on the Open vSwitch bridge device (e.g. via `os-autoinst-setup-multi-machine`).

After this it should be possible to reach other hosts. You may also launch a 2nd VM to see whether

the VMs can talk to each other. You may conduct ping tests similar to the `ping_client` test mentioned in the previous section (see the utility function in openSUSE tests for details). When running ping you can add/remove machines to/from the GRE network to bisect problematic hosts/connections (via `ovs-vsctl add-port` ⋯ and `ovs-vsctl del-port` ⋯).

Debugging Open vSwitch Configuration

Boot sequence with wicked (version 0.6.23 and newer):

1. openvswitch (as above)

2. wicked - creates the bridge `br1` and tap devices, adds tap devices to the bridge,

3. firewalld (or SuSEfirewall2 in older setups)

4. os-autoinst-openvswitch - installs openflow rules, handles vlan assignment

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

```
cat /proc/sys/net/ipv4/conf/{br1,eth0}/forwarding # check whether IP forwarding is
enabled
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
ovs-dpctl show # show basic info on all datapaths
ovs-dpctl dump-flows # displays flows in datapaths
ovs-appctl stp/show # show stp information
ovs-appctl fdb/show br1 # show MAC address table
```

When everything is ok and the machines are able to communicate, the ovs-vsctl should show something like the following:

```
Bridge "br0"
    Port "br0"
        Interface "br0"
            type: internal
    Port "tap0"
        Interface "tap0"
    Port "tap1"
        tag: 1
        Interface "tap1"
    Port "tap2"
        tag: 1
        Interface "tap2"
  ovs_version: "2.11.1"
```

NOTE Notice the tag numbers are assigned to tap1 and tap2. They should have the same number.

To list the rules which are effectively configured in the underlying netfilter (`nftables` or `iptables`) use one of the following commands depending on which netfilter is used.

> **NOTE** Whether firewalld is using `nftables` or `iptables` is determined by the setting `FirewallBackend` in `/etc/firewalld/firewalld.conf`. SuSEfirewall2 is always using `iptables`.

```
nft list tables          # list all tables
nft list table firewalld # list all rules in the specified table
```

```
iptables --list --verbose # list all rules with package counts
```

Check the flow of packets over the network:

- 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

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

> **NOTE** To list the package count when `nftables` is used one needed to use counters (which can be added to existing rules).

Debugging GRE tunnels and MTU sizes

Initial setup for all experiments

```
# Enable ip forwarding
sysctl -w net.ipv4.ip_forward=1
sysctl -w net.ipv6.conf.all.forwarding=1
# Install and enable openvswitch
zypper in openvswitch3
systemctl enable --now openvswitch
```

Host	Network address	Bridge address	Remote IP
A	192.0.2.1/24	192.168.42.1/24	192.0.2.2
B	192.0.2.2/24	192.168.43.1/24	192.0.2.1

Simple scenario

Two servers with a single bridge on each side connected with GRE tunnel.

```
# Create bridge and tunnel
nmcli con add type bridge con.int br0 bridge.stp yes ipv4.method manual ipv4.address
"$bridge_address" ipv4.routes 192.168.42.0/23
nmcli con add type ip-tunnel mode gretap con.int gre1 master br0 remote "$remote_ip"

# Test the tunnel with ping
#   -M do    -- prohibit fragmentation
#   -s xxxx -- set packet size

ping -c 3 -M do -s 1300 192.168.42.1
ping -c 3 -M do -s 1300 192.168.43.1
```

Scenario with openvswitch

Two servers with a one virtual bridge connected with GRE tunnel.

```
# Create bridge, port and interface
nmcli con add type ovs-bridge con.int br0 ovs-bridge.stp-enable yes
nmcli con add type ovs-port con.int br0 con.master br0
nmcli con add type ovs-interface con.int br0 con.master br0 ipv4.method manual
ipv4.address "$bridge_address" ipv4.routes 192.168.42.0/23

# Create GRE tunnel
nmcli con add type ovs-port con.int gre1 con.master br0
nmcli con add type ip-tunnel mode gretap con.int gre1 master gre1 remote "$remote_ip"

# Test the tunnel
ping -c 3 -M do -s 1300 192.168.42.1
ping -c 3 -M do -s 1300 192.168.43.1
```

```
#  ovs-vsctl show
de1f31e9-1b51-4cc3-954a-4e037191ac07
    Bridge br0
        Port br0
            Interface br0
                type: internal
        Port gre1
            Interface gre1
                type: system
    ovs_version: "3.1.0"
```

GRE tunnel made in openvswitch

openvswitch uses flow-based GRE tunneling, i.e. one interface `gre_sys` for all tunnels, the tunnel can be created by `ovs-vsctl`. After that, everything works as expected.

```
# Create bridge, port and interface
nmcli con add type ovs-bridge con.int br0 ovs-bridge.stp-enable yes
nmcli con add type ovs-port con.int br0 con.master br0
nmcli con add type ovs-interface con.int br0 con.master br0 ipv4.method manual
ipv4.address "$bridge_address" ipv4.routes 192.168.42.0/23

# Create GRE tunnel
ovs-vsctl add-port br0 gre1 -- set interface gre1 type=gre options:remote_ip=
"$remote_ip"

# Test the tunnel
ping -c 3 -M do -s 1300 192.168.42.1
ping -c 3 -M do -s 1300 192.168.43.1
```

```
#  ovs-vsctl show
de1f31e9-1b51-4cc3-954a-4e037191ac07
    Bridge br0
        Port br0
            Interface br0
                type: internal
        Port gre1
            Interface gre1
                type: gre
                options: {remote_ip="192.0.2.2"}
    ovs_version: "3.1.0"
```

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 the switch and user mode networking:

```
systemctl enable --now openqa-vde_switch
systemctl enable --now openqa-slirpvde
```

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

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.

Repository URLs

- os-autoinst: https://github.com/os-autoinst/os-autoinst
  - the "backend" (thing that executes tests and starts/controls the SUT e.g. using QEMU)
- openQA: https://github.com/os-autoinst/openQA
  - mainly the web UI and accompanying daemons like the scheduler
  - the worker (thing that starts the backend and uploads results to the web UI)
  - documentation
  - miscellaneous support scripts
- test distribution: e.g. https://github.com/os-autoinst/os-autoinst-distri-opensuse for openSUSE
  - the actual tests, in case of `os-autoinst-distri-opensuse` conducted on http://openqa.opensuse.org
- needles: e.g. https://github.com/os-autoinst/os-autoinst-needles-opensuse for openSUSE
  - reference images if not already included in the test distribution
- empty example test distribution: https://github.com/os-autoinst/os-autoinst-distri-example
  - 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 in CI as soon as you create a pull request, but you **should** run the tidy scripts locally, i.e. before every commit call:

```
make tidy
```

to ensure your Perl and JavaScript code changes are consistent with the style rules.

- All tests are passed. This is ensured by a CI system. You can also run local tests in your development environment to verify everything works as expected, see Conducting tests)
- For git commit messages use the rules stated on How to Write a Git Commit Message as a reference
- Every pull request is reviewed in a peer review to give feedback on possible implications and how we can help each other to improve

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

Code style suggestions

- In Perl files:
  - Sort the use statements in this order from top to bottom:
    - `strict`, `warnings` or other modules that provide static checks
    - All external modules and from "lib" folder
    - `use FindBin; use lib "$FindBin::Bin/lib";` or similar to resolve internal modules
    - Internal test modules which provide early checks before other modules
    - Other internal test modules
  - When using signatures try to follow these rules:
    - Activate the feature with modules we already use if possible, e.g. `use Mojo::Base 'Something', -signatures;`
    - Use positional parameters whenever possible, e.g. `sub foo ($first, $second) {`
    - Use default values when appropriate, e.g. `sub foo ($first, $second = 'some value') {`
    - Use slurpy parameters when appropriate (hash and array), e.g. `sub foo ($first, @more) {`
    - Use nameless parameters when appropriate (very uncommon), e.g. `sub foo ($first, $, $third) {`
    - Do **not** get too creative with computational default values, e.g. `sub foo ($first, $second = rand($first)) {`
    - Do **not** combine sub attributes with signatures (requires Perl 5.28+), e.g. `sub foo :lvalue ($first) {`

Getting involved into development

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'.

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, normally because they are out of the current scope of the development. Developers looking for a place to start contributing are encouraged to simply go to that list and assign any open issue to themselves.

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. See Dependencies below.

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::Most 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.

Folder structure

Meaning and purpose of the most important folders within openQA are:

public
    Static assets published to users over the web UI or API

t
    Self-tests of openQA

assets
    3rd party JavaScript and CSS files

docs
    Documentation, including this document

etc
    Configuration files including template branding specializations

lib
    Main perl module library folder

script
    Main applications and startup files

.circleci
    circleCI definitions

dbicdh
    Database schema startup and migration files

container
    Container definitions

profiles
    Apparmor profiles

systemd
    systemd service definitions

templates
    HTML templates delivered by web UI

tools
    Development tools

Development setup

For developing openQA and os-autoinst itself it makes sense to checkout the [Git repositories](#) and either execute existing tests or start the daemons manually.

Dependencies

Have a look at the packaged version (e.g. `dist/rpm/openQA.spec` within the root of the openQA repository) for all required dependencies. For development build time dependencies need to be installed as well. Recommended dependencies such as logrotate can be ignored. For openSUSE there is also the `openQA-devel` meta-package which pulls all required dependencies for development.

You can find all required Perl modules in form of a `cpanfile` that enables you to install them with a CPAN client. They are also defined in `dist/rpm/openQA.spec`.

Conducting tests

To execute all existing checks and tests simply call:

```
make test
```

for style checks, unit and integration tests.

To execute single tests call `make` with the selected tests in the `TESTS` variable specified as a white-space separated list, for example:

```
make test TESTS=t/config.t
```

or

```
make test TESTS="t/foo.t t/bar.t"
```

To run only unit tests without other tests (perltidy or database tests):

```
make test-unit-and-integration TESTS=t/foo.t
```

Or use `prove` after pointing to a local test database in the environment variable `TEST_PG`. Also, 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= LC_ALL=C.utf8
LANGUAGE= prove -v t/14-grutasks.t
```

In the case of wanting to tweak the tests as above, to speed up the test initialization, start

PostgreSQL using `t/test_postgresql` instead of using the system service. E.g.

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

To check the coverage by individual test files easily call e.g.

```
make coverage TESTS=t/24-worker-engine.t
```

and take a look into the generated coverage HTML report in `cover_db/coverage.html`.

We use annotations in some places to mark "uncoverable" code such as this:

```
# uncoverable subroutine
```

See the docs for details https://metacpan.org/pod/Devel::Cover

There are some ways to save some time when executing local tests:

- One option is selecting individual tests to run as explained above
- Set the make variable `KEEP_DB=1` to keep the test database process spawned for tests for faster re-runs or run tests with `prove` manually after the test database has been created.
- Run `tools/tidy --only-changed` to tidy up code before committing in git
- Set the environment variable `DIE_ON_FAIL=1` from `Test::Most` for faster aborts from failed tests.

For easier debugging of t/full-stack.t one can set the environment variable `OPENQA_FULLSTACK_TEMP_DIR` to a clean directory (relative or absolute path) to be used for saving temporary data from the test, for example the log files from individual test job runs within the full stack test.

Setting up the PostgreSQL database

One also needs to setup a PostgreSQL database for openQA manually owned by your regular user:

1. Install PostgreSQL - under openSUSE the following package are required: `postgresql-server postgresql-init`
2. Start the server: `systemctl start postgresql`
3. The next two steps need to be done as 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-local` where `openqa-local` is the name you want to use for the database
6. Configure openQA to use PostgreSQL as described in the section Database of the installation guide. User name and password are not required.

7. openQA will default-initialize the new database on the next startup.

The script `openqa-setup-db` can be used to conduct step 4 and 5.

Importing production data

Assuming you have already followed steps 1. to 4. above:

1. Create a separate database: `createdb -O your_username openqa-o3` where `openqa-o3+` is the name you want to use for the database

2. The next steps must be run as the user you start your local openQA instance with, i.e. the `your_username` user.

3. Import dump: `pg_restore -c -d openqa-o3 path/to/dump` Note that errors of the form `ERROR: role "geekotest" does not exist` are due to the users in the production setup and can safely be ignored. Everything will be owned by `your_username`.

4. Configure openQA to use that database as in step 7. above.

Manual daemon setup

This section should give you a general idea how to start up daemons manually for development.

For basic development it is recommended to start the openQA services from a regular user account using a local "openQA base directory" included within the source repository folder. For example to start the webserver for development together with the required test database for development within your working copy call:

```
t/test_postgresql /dev/shm/tpg
TEST_PG='DBI:Pg:dbname=openqa_test;host=/dev/shm/tpg' OPENQA_DATABASE=test
OPENQA_BASEDIR=t/data script/openqa daemon
```

The other daemons (mentioned in the architecture diagram) are started in the same way when required, e.g. `script/openqa-scheduler daemon`.

You can also have a look at the systemd unit files. Although it likely makes not much sense to use them directly you can have a look at them to see how the different daemons are started. They are found in the `systemd` directory of the openQA repository. You can substitute `/usr/share/openqa/` with the path of your openQA Git checkout.

Of course you can ignore the user specified in these unit files and instead start everything as your regular user as mentioned above. However, you need to ensure that your user has the permission to the "openQA base directory". That is not the case by default so it makes sense to customize it.

Note that the web UI daemon will pull required JavaScript/CSS libraries automatically when started the first time. This might take a while and requires an internet connection.

You do **not** need to setup an additional web server because the daemons already provide one. The port under which a service is available is logged on startup (the main web UI port is 9625 by

default). Local workers need to be configured to connect to the main web UI port (add `HOST = http://localhost:9526+ to `workers.ini`).

Also find more details in [Run tests without Container](#).

Further tips

- 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`

- In case you have problems with broken rendering of the web page it can help to delete the asset cache and let the webserver regenerate it on first startup. For this delete the subdirectories `.sass-cache/`, `assets/cache/` and `assets/assetpack.db`. Make sure to look for error messages on startup of the webserver and to force the refresh of the web page in your browser.

- If you get errors like "ERROR: Failed to build gem native extension." make sure you have all listed dependencies including the "sass" application installed.

- For a concrete example some developers use under openSUSE Tumbleweed have a look at the [openQA-helper repository](#).

Handling of dependencies

Javascript and CSS

Install third-party JavaScript and CSS files via their corresponding npm packages and add the paths of those files to `assets/assetpack.def`.

If a dependency is not available on npm you may consider adding those files under `assets/3rdparty`. Additionally, add the license(s) for the newly added third-party code to the root directory of the repository. Do **not** duplicate common/existing licenses; extend the `Files:`-section at the beginning of those files instead.

Perl and other packages

In openQA, there is a `dependencies.yaml` file including a list of dependencies, separated in groups. For example the openQA client does not need all modules required to run openQA. Edit this file to add or change a dependency and run `make update-deps`. This will generate the `cpanfile` and `dist/rpm/openQA.spec` files.

The same applies to `os-autoinst` where `make update-deps` will generate the `cpanfile`, `os-autoinst.spec` and `container/ci/Dockerfile`.

If changing any package dependencies make sure packages and updated packages are available in openSUSE Factory and whatever current Leap version is in development. New package dependencies can be submitted. Before merging the according change into the main openQA repo the dependency should be published as part of openSUSE Tumbleweed.

Remarks regarding CI

- The CI of os-autoinst uses the container made using `container/ci/Dockerfile`.
- The CI of openQA uses the container made using `container/devel:openQA:ci/base/Dockerfile` and further dependencies listed in `tools/ci/ci-packages.txt` (see CircleCI documentation).
- There is an additional check running using OBS to check builds of packages against openSUSE Tumbleweed and openSUSE Leap.

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. In doubt, just follow the instructions below. If an empty migration has been emitted (SQL file produced in step 3. does not contain any statements) you can just drop the migration again.

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 is 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. Custom 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 custom migration scripts are executed in addition to the automatically generated ones. If the name of the custom migration script comes before `001-auto.sql` in alphabetical order it will be executed **before** the automatically created migration script. That is most of the times **not** desired.

The above steps are only for preparing the required SQL statements for the migration.

The migration itself (which alters your database!) is done **automatically** the first time the web UI is (re)started. So be sure **to backup your database** before restarting 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 initialize or update the database manually before restarting the web UI you can run either `./script/initdb --init_database` or `./script/upgradedb --upgrade_database`.

Migrations that affect possibly big tables should be tested against a local import of a production database to see how much time they need. Checkout the Importing production data section for details.

A migration can cause the analyser to regress so it produces worse query plans leading to impaired performance. Checkout the Working on database-related performance problems section for how to tackle this problem.

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.

Adding new authentication module

openQA comes with two authentication modules providing authentication methods: OpenID and Fake (see User authentication).

All authentication modules reside in `lib/OpenQA/Auth` directory. During openQA start, the `[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, the module for the given method is imported or 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 (which is for instance used to store test results) by setting the environment variable `OPENQA_BASEDIR`. The default value is `/var/lib`. Be sure to clear that variable when running unit tests locally (see next section). Take into account that the test results and assets can need a big amount of disk space.

Running tests of openQA itself

Beside simply running the testsuite, it is also possible to use containers. Using containers, tests are executed in the same environment as on CircleCI. This allows to reproduce issues specific to that environment.

Run tests without container

Be sure to install all required dependencies. The package `openQA-devel` will provide them.

If the package is not available the dependencies can also be found in the file `dist/rpm/openQA.spec` in the openQA repository. In this case also the package `perl-Selenium-Remote-Driver` is required to run UI tests. You also need to install chromedriver and either chrome or chromium for the UI tests.

To execute the testsuite use `make test`. This will also initialize a temporary PostgreSQL database used for testing. To do this step manually run `t/test_postgresql /dev/shm/tpg` to initialize a temporary PostgreSQL database and export the environment variable as instructed by that script. It is also possible to run a particular test, for example `prove t/api/01-workers.t`. When using `prove` directly, make sure an English locale is set (e.g. `export LC_ALL=C.utf8 LANGUAGE=` before initializing the database and running `prove`).

To keep the test database running after executing tests with the `Makefile`, add `KEEP_DB=1` to the make arguments. To access the test database, use `psql --host=/dev/shm/tpg openqa_test`.

To watch the execution of the UI tests, set the environment variable `NOT_HEADLESS`.

Run tests within a container

The container used in this section of the documentation is not identical with the container used within the CI. To run tests within the CI environment locally, checkout the CircleCI documentation below.

To run tests in a container please be sure that a container runtime environment, for example podman, is installed. To launch the test suite first it is required to pull the container image:

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

This container image is provided by the OBS repository https://build.opensuse.org/package/show/devel:openQA/openqa_dev and based on the `Dockerfile` within the `container/ci` sub directory of the openQA repository.

Build the image using Makefile target:

```
make container-test-build
```

Note that the image created by that target is called `openqa:latest` while the raw container pulled

from OBS is called `openqa_dev:latest`.

Launch the tests using Makefile target:

```
make launch-container-to-run-tests-within
```

Run tests by spawning a container manually, e.g.:

```
podman run -v OPENQA_LOCAL_CODE:/opt/openqa -e VAR1=1 -e VAR2=1 openqa:latest make
run-tests-within-container
```

Replace `OPENQA_LOCAL_CODE` with the location where you have the openQA code.

The command line to run tests manually reveals that the Makefile target `run-tests-within-container` is used to run the tests **inside** the container. It does some preparations to be able to run the full stack test within a container and considers a few environment variables defining our test matrix:

CHECKSTYLE=1	
FULLSTACK=0	UITESTS=0
FULLSTACK=0	UITESTS=1
FULLSTACK=1	
HEAVY=1	
GH_PUBLISH=true	

So by replacing VAR1 and VAR2 with those values one can trigger the different tests of the matrix.

Of course it is also possible to run (specific) tests directly via `prove` instead of using the Makefile targets.

Tips

Commands passed to `podman run` will be executed after the initialization script (which does database creation and so on). So if there is the need to run an interactive session after it just do:

```
podman run -it -v OPENQA_LOCAL_CODE:/opt/openqa openqa:latest bash
```

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

There is 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:

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

In case there is the need to follow what is happening in the currently running container (the execution will terminate the session):

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

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

```
xhost `local:root
podman run --rm -ti --name openqa-testsuite -v /tmp/.X11-unix:/tmp/.X11-unix:rw -e
DISPLAY="$DISPLAY" -e NOT_HEADLESS=1 openqa:latest 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:

```
podman run ⋯ -e CUSTOM_OS_AUTOINST=1 -v /path/to/your/os-autoinst:/opt/os-autoinst
make run-tests-within-container
```

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, e.g. when the openQA repo contains a lot of profiling data because you enabled `Mojolicious::Plugin::NYTProf`.

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

Logging behavior

Logs are redirected to a logfile when running tests within the CI. The output can therefore not be asserted using `Test::Output`. This can be worked around by temporarily assigning a different `Mojo::Log` object to the application. To test locally under the same condition set the environment variable `OPENQA_LOGFILE`.

Note that redirecting the logs to a logfile only works for tests which run `OpenQA::Log::setup_log`. In other tests the log is just printed to the standard output. This makes use of `Test::Output` simple but it should be taken care that the test output is not cluttered by log messages which can be quite irritating.

Test runtime limits

The test modules use `OpenQA::Test::TimeLimit` to introduce a test module specific timeout. The timeout is automatically scaled up based on environment variables, e.g. `CI` for continuous integration environments, as well as when executing while test coverage data is collected as longer runtimes should be expected in these cases. Consider lowering the timeout value based on usual local execution times whenever a test module is optimized in runtime. If the timeout is hit the test module normally aborts with a corresponding message.

To disable the timeout globally set the environment variable `OPENQA_TEST_TIMEOUT_DISABLE=1`.

Please be aware of the exception when the timeout triggers after the actual test part of a test module has finished but not all involved processes have finished or END blocks are processed. In this case the output can look like

```
t/my_test.t .. All 1 subtests passed

Test Summary Report
-------------------
t/my_test.t (Wstat: 14 Tests: 1 Failed: 0)
  Non-zero wait status: 14
Files=1, Tests=1,  2 wallclock secs ( 0.03 usr  0.00 sys +  0.09 cusr  0.00 csys =
0.12 CPU)
Result: FAIL
```

where "Wstat: 14" and "Non-zero wait status: 14" mean that the test process received the "ALRM" signal (signal number 14).

In case of problems with timeouts look into `OpenQA::Test::TimeLimit` to find environment variables that can tweaked to disable or change timeout values or timeout scale factors. If you want to disable the timeout for indefinite manual debugging, set the environment variable `OPENQA_TEST_TIMEOUT_DISABLE=1`. The option `OPENQA_TEST_TIMEOUT_SCALE_CI` is only effective if the environment variable `CI` is set, which e.g. it is in circleCI and OBS but not in local development environments. When running with coverage analysis enabled the scaling factor of `OPENQA_TEST_TIMEOUT_SCALE_COVER` is applied to account for the runtime overhead.

In case of Selenium based UI tests timing out trying to find a local chromedriver instance the variable `OPENQA_SELENIUM_TEST_STARTUP_TIMEOUT` can be set to a higher value. See https://metacpan.org/pod/Selenium::Chrome#startup_timeout for details.

CircleCI workflow

The goal of the following workflow is to provide a way to run tests with a pre-approved list of dependencies both in the CI and locally.

Dependency artefacts

- ci-packages.txt lists dependencies to test against.

- autoinst.sha contains sha of os-autoinst commit for integration testing. The testing will run against the latest master if empty.

Managing and troubleshooting dependencies

`ci-packages.txt` and `autoinst.sha` are aimed to represent those dependencies which change often. In normal workflow these files are generated automatically by dedicated Bot, then go in PR through CI, then reviewed and accepted by human. So, in normal workflow it is guaranteed that everyone always works on list of correct and approved dependencies (unless they explicitly tell CI to use custom dependencies).

The Bot tracks dependencies only in master branch by default, but this may be extended in circleci config file. The Bot uses `tools/ci/build_dependencies.sh` script to detect any changes. This script can be used manually as well. Alternatively just add newly introduced dependencies into ci-packages.txt, so CI will run tests with them.

Occasionally it may be a challenge to work with ci-packages.txt (e.g. package version is not available anymore). In such case you can either try to rebuild ci-packages.txt using `tools/ci/build_dependencies.sh` or just remove all entries and put only openQA-devel into it Script `tools/ci/build_dependencies.sh` can be also modified when major changes are performed, e.g. different OS version or packages from forked OBS project, etc.

Run tests locally using a container

One way is to build an image using the `build_local_docker.sh` script, start a container and then use the same commands one would use to test locally.

Pull the latest base image (otherwise it may be outdated):

```
podman pull registry.opensuse.org/devel/openqa/ci/containers/base:latest
```

Create an image called `localtest` based on the contents of `ci-packages.txt` and `autoinst`:

```
tools/ci/build_local_docker.sh
```

Mount the openQA checkout under `/opt/testing_area` within the container and run tests as usual, e.g.:

```
podman run -it --rm -v $PWD:/opt/testing_area localtest bash -c 'make test
TESTS=t/ui/25*'
```

Alternatively, start the container and execute commands via podman exec, e.g.:

```
podman run --rm --name t1 -v $PWD:/opt/testing_area localtest tail -f /dev/null &
sleep 1
podman exec -it t1 bash -c 'make test TESTS=t/ui/25-developer_mode.t'
podman stop -t 0 t1
```

Run tests using the circleci tool

After installing the circleci tool the following commands will be available. They will build the container and use committed changes from current local branch.

```
circleci local execute --job test1
circleci local execute --job testui
circleci local execute --job testfullstack
circleci local execute --job testdeveloperfullstack
```

Changing config.cnf

Command to verify the YAML with the circleci tool:

```
circleci config process .circleci/config.yml
```

Building plugins

Not all code needs to be included in openQA itself. openQA also supports the use of 3rd party plugins that follow the standards for plugins used by the Mojolicious web framework. These can be distributed as normal CPAN modules and installed as such alongside openQA.

Plugins are a good choice especially for extensions to the UI and HTTP API, but also for notification systems listening to various events inside the web server.

If your plugin was named `OpenQA::WebAPI::Plugin::Hello`, you would install it in one of the include directories of the Perl used to run openQA, and then configure it in `openqa.ini`. The `plugins` setting in the `global` section will tell openQA what plugins to load.

```
# Tell openQA to load the plugin
[global]
plugins = Hello

# Plugin specific configuration (optional)
[hello_plugin]
some = value
```

The plugin specific configuration is optional, but if defined would be available in `$app→config→{hello_plugin}`.

To extend the UI or HTTP API there are various named routes already defined that will take care of authentication for your plugin. You just attach the plugin routes to them and only authenticated requests will get through.

```perl
package OpenQA::WebAPI::Plugin::Hello;
use Mojo::Base 'Mojolicious::Plugin';

sub register {
    my ($self, $app, $config) = @_;

    # Only operators may use our plugin
    my $ensure_operator = $app->routes->find('ensure_operator');
    my $plugin_prefix = $ensure_operator->any('/hello_plugin');

    # Plain text response (under "/admin/hello_plugin/")
    $plugin_prefix->get('/' => sub {
      my $c = shift;
      $c->render(text => 'Hello openQA!');
    })->name('hello_plugin_index');

    # Add a link to the UI menu
    $app->config->{plugin_links}{operator}{'Hello'} = 'hello_plugin_index';
}

1;
```

The `plugin_links` configuration setting can be modified by plugins to add links to the `operator` and `admin` sections of the openQA UI menu. Route names or fully qualified URLs can be used as link targets. If your plugin uses templates, you should reuse the `bootstrap` layout provided by openQA. This will ensure a consistent look, and make the UI menu available everywhere.

```
% layout 'bootstrap';
% title 'Hello openQA!';
<div>
   <h2>Hello openQA!</h2>
</div>
```

For UI plugins there are two named authentication routes defined:

1. `ensure_operator`: under `/admin/`, only allows logged in users with `operator` privileges

2. `ensure_admin`: under `/admin/`, only allows logged in users with `admin` privileges

And for HTTP API plugins there are four named authentication routes defined:

1. `api_public`: under `/api/v1/`, allows access to everyone

2. `api_ensure_user`: under `/api/v1/`, only allows authenticated users

3. `api_ensure_operator`: under `/api/v1/`, only allows authenticated users with `operator` privileges

4. `api_ensure_admin`: under `/api/v1/`, only allows authenticated nusers with `admin` privileges

To generate a minimal installable plugin with a CPAN distribution directory structure you can use the Mojolicious tools. It can be packaged just like any other Perl module from CPAN.

```
$ mojo generate plugin -f OpenQA::WebAPI::Plugin::Hello
...
$ cd OpenQA-WebAPI-Plugin-Hello/
$ perl Makefile.PL
...
$ make test
...
```

And if you need code examples, there are some plugins included with openQA.

Checking for JavaScript problems

One can use the tool jshint to check for problems within JavaScript code. It can be installed easily via npm.

```
npm install jshint
node_modules/jshint/bin/jshint path/to/javascript.js
```

Profiling the web UI

1. Install NYTProf, under openSUSE Tumbleweed: `zypper in perl-Devel-NYTProf perl-Mojolicious-Plugin-NYTProf`

2. Put `profiling_enabled = 1`+ in `openqa.ini`.

3. Optionally import production data like described in the official contributors documentation.

4. Restart the web UI, browse some pages. Profiling is done in the background.

5. Access profiling data via `/nytprof` route.

Note

Profiling data is extensive. Remove it if you do not need it anymore and disable the `profiling_enabled` configuration again if not needed anymore.

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.

Containerized setup

The installation guide already contains simple one-liners for starting the web UI and workers in the Container based setup section.

This chapter describes how to deploy the containers for the openQA web UI and the workers using `docker-compose` or `podman-compose`.

There is also an approach using Fedora-based images mentioned but it is not supported by upstream.

Get container images

You can either build the images locally or use Fedora images from Docker Hub.

For the `docker-compose` setup it is required to build the images locally. However, it is done via `docker-compose` and explained later so this section can be skipped.

Download Fedora-based images from the Docker Hub

```
podman pull fedoraqa/openqa_data
podman pull fedoraqa/openqa_webui
podman pull fedoraqa/openqa_worker
```

Build openSUSE-based images locally

```
podman build -t openqa_data ./openqa_data
podman build -t openqa_webui ./webui
podman build -t openqa_worker ./worker
```

Setup with Fedora-based images

Data storage and directory structure

Our intent was to create universal `webui` and `worker` containers and move all data storage and configurations to a third container, called `openqa_data`. `openqa_data` is a so called Data Volume Container and is used for the database and to store results and configuration. During development and in production, you could update `webui` and `worker` images but as long as `openqa_data` is intact, you do not lose any data.

To make development easier and to reduce the final size of the `openqa_data` container, this guide describes how to override `tests` and `factory` directories with directories from your host system. This is not necessary but recommended. This guide is written with this setup in mind.

It is also possible to use `tests` and `factory` from within the `openqa_data` container (so you do not have any dependency on your host system) or to leave out the `openqa_data` container altogether (so you have only `webui` and `worker` containers and data is loaded and saved completely into your host system). If this is what you prefer, checkout the sections Keeping all data in the Data Volume Container and Keeping all data on the host system respectively.

Otherwise, when you want to have the big files (isos and disk images, tests and needles) outside of the Volume container, you should create this file structure from within the directory you are going to execute the container:

```
mkdir -p data/factory/{iso,hdd} data/tests
```

It could be necessary to either run all containers in privileged mode or to setup SELinux properly. If you are having problems with it, run this command:

```
chcon -Rt svirt_sandbox_file_t data
```

Update firewall rules

There is a bug in Fedora with `docker-1.7.0-6` package that prevents containers to communicate with each other. This bug prevents workers to connect to the web UI. If you use docker, as a workaround, run:

```
sudo iptables -A DOCKER --source 0.0.0.0/0 --destination 172.17.0.0/16 -m conntrack
--ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A DOCKER --destination 0.0.0.0/0 --source 172.17.0.0/16 -j ACCEPT
```

on the host machine.

Run the data and web UI containers

```
podman run -d -h openqa_data --name openqa_data -v "$PWD"/data/factory:/data/factory
-v "$PWD"/data/tests:/data/tests fedoraqa/openqa_data
podman run -d -h openqa_webui --name openqa_webui --volumes-from openqa_data -p 80:80
-p 443:443 fedoraqa/openqa_webui
```

You can change the `-p` parameters if you do not want the openQA instance to occupy ports 80 and 443, e.g. `-p 8080:80 -p 8043:443`, but this will cause problems if you wish to set up workers on other hosts (see below). You do need root privileges to bind ports 80 and 443 in this way.

It is now necessary to create and store the client keys for openQA. In the next two steps, you will set an OpenID provider (if necessary), create the API keys in the openQA's web interface, and store the configuration in the Data Container.

Generate and configure API credentials

Go to https://localhost/api_keys, generate key and secret. Then run the following command substituting `KEY` and `SECRET` with the generated values:

```
exec -it openqa_data /scripts/client-conf set -l KEY SECRET
```

Run the worker container

```
podman run -d -h openqa_worker_1 --name openqa_worker_1 --link
openqa_webui:openqa_webui --volumes-from openqa_data --privileged
fedoraqa/openqa_worker
```

Check whether the worker connected in the web UI's administration interface.

To add more workers, increase the number that is used in hostname and container name, so to add worker 2 use:

```
podman run -d -h openqa_worker_2 --name openqa_worker_2 --link
openqa_webui:openqa_webui --volumes-from openqa_data --privileged
fedoraqa/openqa_worker
```

Enable services

Some systemd services are provided to start up the containers, so you do not have to keep doing it manually. To install and enable them:

```
sudo cp systemd/*.service /etc/systemd/system
sudo systemctl daemon-reload
sudo systemctl enable openqa-data.service
sudo systemctl enable openqa-webui.service
sudo systemctl enable openqa-worker@1.service
```

Of course, if you set up two workers, also do `sudo systemctl enable openqa-worker@2.service`, and so on.

Get tests, ISOs and create disks

You have to put your tests under `data/tests` directory and ISOs under `data/factory/iso` directory. For testing Fedora, run:

```
git clone https://bitbucket.org/rajcze/openqa_fedora data/tests/fedora
wget https://dl.fedoraproject.org/pub/alt/stage/22_Beta_RC3/Server/x86_64/iso/Fedora-
Server-netinst-x86_64-22_Beta.iso -O data/factory/iso/Fedora-Server-netinst-x86_64-
22_Beta_RC3.iso
```

And set permissions, so any user can read/write the data:

```
chmod -R 777 data
```

This step is unfortunately necessary with Docker because Docker can not mount a volume with specific user ownership in container, so ownership of mounted folders (uid and gid) is the same as on your host system (presumably 1000:1000 which maps into nonexistent user in all of the containers).

If you wish to keep the tests (for example) separate from the shared directory, for any reason (we do, in our development scenario) refer to the [Developing tests with Container setup] section at the end of this document.

Populate the openQA database:

```
podman exec openqa_webui /var/lib/openqa/tests/fedora/templates
```

Create all necessary disk images:

```
cd data/factory/hdd && createhdds.sh VERSION
```

where `VERSION` is the current stable Fedora version (its images will be created for upgrade tests) and createhdds.sh is in `openqa_fedora_tools` repository in `/tools` directory. Note that you have to have `libguestfs-tools` and `libguestfs-xfs` installed.

Setup openQA with openSUSE-based images and docker-compose

All relative paths in this section are relative to a checkout of openQA's Git repository.

Configuration

The web UI will be available under http://localhost and https://localhost. So it is using default HTTP(S) ports. Make sure those ports are not used by another service yet or change ports in the **nginx section** of `container/webui/docker-compose.yaml`.

If TLS is required, edit the certificates mentioned in the nginx section of `container/webui/docker-compose.yaml` to point it to your certificate. By default, a self-signed test certificate is used.

Edit `container/webui/conf/openqa.ini` to configure the web UI as needed, e.g. change `[auth] method = Fake` or `[logging] level = debug`. If the web UI will be exposed/accessed via a certain domain, set `base_url` in the `[global]` section accordingly so redirections for authentication work.

Edit `container/worker/conf/workers.ini` to configure the workers as needed.

Edit `container/webui/nginx.conf` to customize the NGINX configuration.

To set the number of web UI replicas set the environment variable `OPENQA_WEBUI_REPLICAS` to the desired number. If this is not set, then the default value is 2. Additionally, you can edit `container/webui/.env` to set the default value for this variable. This does not affect the websocket server, livehandler and gru.

All the data which normally ends up under `/var/lib/openqa` in the default setup will be stored under `container/webui/workdir/data`. The database will be stored under `container/webui/workdir/db`.

Build images

`docker-compose` will build images automatically. However, it is also possible to build images explicitly:

```
cd container/webui
docker-compose build       # build web UI images
docker-compose build nginx # build a specific web UI image
cd container/worker
docker-compose build       # build worker images
```

Run the web UI containers in HA mode

To start the containers, just run:

```
cd container/webui
docker-compose up
```

To rebuild the images, add `--build`.

It is also possible to run it in the background by adding `-d`. To stop it again, run:

```
docker-compose down
```

Further useful commands:

```
docker-compose top                      # show spawned containers and their status
docker-compose logs                     # access logs
docker-compose exec db psql openqa openqa  # open psql shell
```

Generate and configure API credentials

Go to https://localhost/api_keys and generate a key/secret and configure it in `container/webui/conf/client.conf` **and** `container/worker/conf/client.conf` in all sections.

The web UI services need the credentials as well for internal API requests. So it is required to restart the web UI containers to apply the changes:

```
cd container/webui
docker-compose restart
```

Run the worker container

Configure the number of workers to start via the environment variable `OPENQA_WORKER_REPLICAS`. By default, one worker is started.

To start a worker, just run:

```
cd container/worker
docker-compose up
```

The same `docker-compose` commands as shown for the web UI can be used for further actions. The worker should also show up in the web UI's workers table.

It is also possible to use a container runtime environment directly as shown by the script `container/worker/launch_workers_pool.sh` which allows spawning a bunch of workers with consecutive numbers for the `--instance` parameter:

It will launch the desired number of workers in individual containers using consecutive numbers

for the `--instance` parameter:

```
cd container/worker
./launch_workers_pool.sh --size=<number-of-workers>
```

Get tests, ISOs and create disks

You have to put your tests under `data/tests` directory and ISOs under `data/factory/iso` directory. For testing openSUSE, follow this guide.

The test distribution might have additional dependencies which need to be installed into the worker container before tests can run. To install those dependencies automatically on the container startup one can add a script called `install_deps.sh` in the root of the test distribution which would install the dependencies, e.g. via a `zypper` call.

Running jobs

After performing the "setup" tasks above - do not forget about tests and ISOs.

Then you can use `openqa-cli` as usual with the containerized web UI. It is also possible to use `openqa-clone-job`, e.g.:

```
cd container/webui
docker-compose exec webui openqa-clone-job \
    --host http://localhost:9526 \
    https://openqa.opensuse.org/tests/1896520
```

Further configuration options

Most of these options do **not** apply to the docker-compose setup.

Change the OpenID provider

https://www.opensuse.org/openid/user/ is set as a default OpenID provider. To change it, run:

```
podman exec -it openqa_data /scripts/set_openid
```

and enter the provider's URL.

Adding workers on other hosts

You may want to add workers on other hosts, so you do not need one powerful host to run the UI and all the workers.

Let's assume you are setting up a new 'worker host' and it can see the web UI host system with the hostname openqa_webui.

You must somehow share the data directory from the web UI host to each host on which you want to run workers. For instance, to use sshfs on the new worker host, run:

```
sshfs -o context=unconfined_u:object_r:svirt_sandbox_file_t:s0
openqa_webui:/path/to/data /path/to/data
```

Of course, the worker host must have an ssh key the web UI host will accept. You can add this mount to /etc/fstab to make it permanent.

Then check openqa_fedora_tools out on the worker host and run the data container, as described above:

```
podman run -d -h openqa_data --name openqa_data -v /path/to/data/factory:/data/factory
-v /path/to/data/tests:/data/tests fedoraqa/openqa_data
```

and set up the API key with podman exec -ti openqa_data /scripts/set_keys.

Finally create a worker container, but omit the use of --link. Ensure you use a hostname which is different from all other worker instances on all other hosts. The container name only has to be unique on this host, but it probably makes sense to always match the hostname to the container name:

```
podman run -h openqa_worker_3 --name openqa_worker_3 -d --volumes-from openqa_data
--privileged fedoraqa/openqa_worker
```

If the container will not be able to resolve the `openqa_webui` hostname (this depends on your network setup) you can use `--add-host` to add a line to `/etc/hosts` when running the container:

```
podman run -h openqa_worker_3 --name openqa_worker_3 -d --add
-host="openqa_webui:10.0.0.1" --volumes-from openqa_data --privileged
fedoraqa/openqa_worker
```

Worker instances always expect to find the server as `openqa_webui`; if this will not work you must adjust the `/data/conf/client.conf` and `/data/conf/workers.ini` files in the data container. You will also need to adjust these files if you use non-standard ports (see above).

Keeping all data in the Data Volume container

If you decided to keep all the data in the Volume container (`openqa_data`), run the following commands:

```
podman exec openqa_data mkdir -p data/factory/{iso,hdd} data/tests
podman exec openqa_data chmod -R 777 data/factory/{iso,hdd} data/tests
```

In the section about running the web UI and data container, use the `openqa_data` container like this instead:

```
podman run -d -h openqa_data --name openqa_data fedoraqa/openqa_data
```

And finally, download the tests and ISOs directly into the container:

```
podman exec openqa_data git clone https://bitbucket.org/rajcze/openqa_fedora
/data/tests/fedora
podman exec openqa_data wget
https://dl.fedoraproject.org/pub/alt/stage/22_Beta_RC3/Server/x86_64/iso/Fedora-
Server-netinst-x86_64-22_Beta.iso -O /data/factory/iso/Fedora-Server-netinst-x86_64-
22_Beta_RC3
```

The rest of the steps should be the same.

Keeping all data on the host system

If you want to keep all the data in the host system and you prefer not to use a Volume Container, run the following commands:

```
cp -a openqa_data/data.template data
chcon -Rt svirt_sandbox_file_t data
```

In the section about running the web UI and data container, do **not** run the `openqa_data` container and run the `webui` container like this instead:

```
podman run -d -h openqa_webui -v `pwd`/data:/data --name openqa_webui -p 443:443 -p
80:80 fedoraqa/openqa_webui:4.1-3.12
```

Change OpenID provider in `data/conf/openqa.ini` under `provider` in `[openid]` section and then put Key and Secret under both sections in `data/conf/client.conf`.

In the run worker container section, run the worker as:

```
podman run -h openqa_worker_1 --name openqa_worker_1 -d --link
openqa_webui:openqa_webui -v `pwd`/data:/data --volumes-from openqa_webui --privileged
fedoraqa/openqa_worker:4.1-3.12 1
```

Then continue with tests and ISOs downloading as before.

Developing tests with container setup

With this setup, the needles created from the web UI will almost certainly have a different owner and group than your user account. As we have the tests in Git, we still want to retain the original owner and permissions, even when we update/create needles from openQA. To accomplish this, we can use BindFS. An example entry in `/etc/fstab`:

```
bindfs#/home/jskladan/src/openQA/openqa_fedora
/home/jskladan/src/openQA/openqa_fedora_tools/docker/data/tests/fedora    fuse
create-for-user=jskladan,create-for-group=jskladan,create-with-perms=664:a+X,perms=777
0    0
```

Mounts the `openqa_fedora` directory to the `…/tests/fedora directory`. All files in the `tests/fedora` directory seem to have 777 permissions set, but new files are created (in the underlying `openqa_fedora` directory) with `jskladan:jskladan` user and group, and 664:a+X permissions.