usmqe-tests Documentation

Release 0.0.1

usmqe team

Apr 03, 2019

Contents

USM QE tests are concerned with automated integration testing of Tendrl project.

This documentation provides all details needed for setting up and running automated integration tests, as well as instruction for test automation development.

CHAPTER 1

Contents

1.1 USM QE Team

The USM QE Team members are (in alphabetical order):

- Daniel Horák (github.com/dahorak)
- Elena Bondarenko (github.com/ebondare)
- Filip Balák (github.com/fbalak)
- Martin Bukatovič (gitlab.com/mbukatov, github.com/mbukatov)

1.2 Overview of USM QE repositories

This document contains a brief overview of structure and purpose of all USM QE repositories.

1.2.1 usmqe-tests

Upstream: https://github.com/usmqe/usmqe-tests

This repository is most important, because it contains python code of automated integration test cases and sphinx based documentation which covers tests and it's setup (you are reading it right now).

1.2.2 usmqe-setup

Upstream: https://github.com/usmqe/usmqe-setup

This repository contains test setup automation implemented via Ansible.

1.2.3 usmqe-centos-ci

Upstream: https://github.com/usmqe/usmqe-centos-ci

Machine deployment for Tendrl project and Jenkins jobs in CentOS CI.

1.3 Setup of QE Server

author mbukatov

date 2018-06-08

QE Server is machine where `usmqe-tests` (along with all it's dependencies) are installed and where the integration tests are executed from.

1.3.1 Requirements

QE Server must be a RHEL/CentOS 7 machine.

1.3.2 QE Server Playbook

To unify and automate the deployment of QE Server machines, usmqe team maintains qe_server.yml playbook in the usmqe-setup repository. You should use this playbook so that the same qe enviroment is used across all qe machines.

1.3.3 Quick Example of QE Server deployment

You need a RHEL 7 or CentOS 7 machine for the QE Server. For the purpose of this example, we are going to quickly create one virtual machine via virt-builder tool (to use this tool as shown below, you need to install at least `libguestfs-tools-c` and `libguestfs-xfs` packages, see also libguestfs Fedora package page).

First we build a vm image (uploading ssh authorized keys like this would make the machine accessible for everyone who has keys on the machine you are running this command):

```
$ virt-builder centos-7.5 -o mbukatov.qe-server.centos7.qcow2 --size 15G --format␣
↪qcow2 --mkdir /root/.ssh  --chmod 0700:/root/.ssh  --upload /root/.ssh/authorized_
↪keys:/root/.ssh/authorized_keys --selinux-relabel --update
```

Note that `virt-builder` sets random root password, which you may like to write down in your password manager so that you can connect to the machine directly via console (using `virsh console`) later in case of problems. That said, under normal circumstances you will connect to the machine via ssh using key based authentication.

Then we import the new image into libvirt creating new virtual machine (aka guest) and booting it for the first time:

```
# virt-install --import --name mbukatov --ram 2048 --os-variant rhel7 --disk path=/
↪var/lib/libvirt/images/mbukatov.qe-server.centos7.qcow2,format=qcow2 --network␣
↪default --noautoconsole
```

If you need change default network bridge or MAC address of the virtual machine, update `--network` option of `virt-install`, eg.: `--network bridge=br0_vlan4,mac=52:54:00:59:15:04`.

When the new machine is ready, specify an ip address or fqdn of the new qe server in the inventory file:

```
$ cat qe.hosts
[qe_server]
10.34.126.60
```

And make sure you have ssh configured properly (this includes ssh keys and local ssh client configuration) so that ansible can work with the machine:

```
$ ansible -i qe.hosts -m ping qe_server
10.34.126.60 | SUCCESS => {
        "changed": false,
        "ping": "pong"
}
```

Then you can run the `qe_server.yml` playbook:

```
$ ANSIBLE_ROLES_PATH=~/projects/tendrl.org/tendrl-ansible/roles ansible-playbook -i␣
↪qe.hosts qe_server.yml
```

Note that we have to reference tendrl-ansible here as the qe server playbook uses tendrl-ansible.gluster-gdeploy-copr role, which installs *upstream* build of gdeploy.

When the ansible playbook run finishes, you can login to the usmqe account on the QE Server for the first time:

```
$ ssh root@10.34.126.6
[root@mbukatov ~]# su - usmqe
[usmqe@mbukatov ~]$ ls
tendrl-ansible  usmqe-setup  usmqe-tests
```

Note that `rh-python36` software collection is enabled by default in `~/.bashrc` file of usmqe user account and that all requirements (eg. pytest, mrglog, . . .) are already available:

```
[usmqe@qeserver ~]$ python --version
Python 3.6.3
[usmqe@qeserver ~]$ py.test --version
This is pytest version 3.6.1, imported from /home/usmqe/.local/lib/python3.6/site-
↪packages/pytest.py
setuptools registered plugins:
  pytest-ansible-playbook-0.3.0 at /home/usmqe/.local/lib/python3.6/site-packages/
↪pytest_ansible_playbook.py
[usmqe@qeserver ~]$ which mrglog_demo.py
~/.local/bin/mrglog_demo.py
```

Also note that even though the default python for usmqe user is `python3.6` from the software collection, one can still run other system utilities which are running on system default python2:

```
[usmqe@qeserver ~]$ ansible --version
ansible 2.5.3
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/usmqe/.ansible/plugins/modules', u'/usr/
↪share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /bin/ansible
  python version = 2.7.5 (default, Apr 11 2018, 07:36:10) [GCC 4.8.5 20150623 (Red␣
↪Hat 4.8.5-28)]
```

This is the case because all python tools packaged in Fedora/Red Hat/CentOS uses explicit shebang:

```
[usmqe@qeserver ~]$ head -1 /usr/bin/ansible
#!/usr/bin/python2
```

1.3.4 Related information

At this point, we have a fresh QE server machine. But for us to be able to run integration tests, we need to:

- Prepare fresh machines where Tendrl and Gluster will be installed. See *Setup of Test Enviroment*.
- Configure the tests, go into `~/usmqe-tests` directory and follow *Configuration before test run*

For full description and examples how to run integration tests, see *Test Execution*.

1.4 Setup of Test Enviroment

USM QE integrations tests are expected to be executed on virtual or bare metal machines so that for each storage role (eg. *gluster client*, *ceph monitor*, *tendrl console*, . . .) there is a dedicated machine (eg. storage client role should not be deployed on the same machine as ceph monitor) and if the role requires multiple machines, a minimal amount of machines needs to be available based on the role needs (eg. having a trusted storage pool on just 2 machines is not very useful for proper integration testing because it would prevent us from testing some important use cases).

For this reason, all post installation and test setup configuration steps are automated via ansible playbooks and stored in a separate usmqe-setup repository. You need to deploy test machines using playbooks from there.

1.5 Test Configuration

This article describes configuration of *USM QE integration tests*.

1.5.1 Configuration Scheme

Configuration for the tests is defined in:

- *YAML configuration files* which contain most of actual test configuration. File conf/main.yaml should contain list of other configuration files that are used as test configuration. File conf/defaults.yaml should be linked there at first place. After this there can be more configuration files that are specific to testing enviroment (for more see example configuration conf/usm_example.yaml). File conf/defaults.yaml should contain default configuration that is not environment specific.
- Ansible *host inventory file* (see an example in conf/usm_example.hosts), which is used both by ansible and by USM QE inventory module to organize machines into groups by it's role in test cluster. Actual path of this file is configured in one of the *YAML configuration files* (see conf/main.yaml).

1.5.2 Details for Test Development

To learn how to access configuration values from code of a test case, see *Reading Configuration Values* for more details.

1.5.3 Configuration before test run

We assume that:

- *QE Server machine* has been configured as described in *Setup of QE Server*
- You have *host inventory file* for the test cluster, which has been already deployed (our deployment automation should generate the inventory file in the end of the process).
- You are logged as *usmqe* user on the QE Server

Now, you need to:

- Check that `usmqe` user can ssh to all nodes with his ssh key stored in `~/.ssh`. This can be configured in `~/.ssh/config`. Public ssh key is deployed on all machines of test cluster.
- Store *host inventory file* in `conf/clustername.hosts` and specify this path in `inventory_file` option of conf/main.yaml.
- Verify that ssh and ansible are configured so that one can reach all machines from test cluster:

```
[usmqe@qeserver ~]$ ansible -i conf/clustername.hosts -m ping -u root all
```

- Provide all mandatory options in *usm config file*. This includes: `username`, `password`, `web_url`, `api_url` and `cluster_member`. The actual list depends on the test suite you are going to run (eg. api tests don't care about `web_url` while LDAP integration tests would need to know address of the LDAP server).

1.5.4 Configuration options

- `log_level` - Log level. It can be one of [DEBUG, INFO, WARNING, ERROR, CRITICAL, FATAL]
- `username` - API and UI login
- `password` - API and UI password
- `web_url` - web UI url
- `api_url` - API url
- `etcd_api_url` - Etcd API url
- `ca_cert` - path to CA cert
- `cluster_member` - one of nodes from cluster which identifies cluster for re-use testing, see section *Structure of Functional Tests*.

1.6 Test Execution

This document briefly describes how to execute USM QE tests.

1.6.1 Preparation for running the tests

Before running USM QE tests, you need to prepare:

- QE Server machine as described in *Setup of QE Server*. The tests are executed under `usmqe` user account on this QE Server machine.
- Machines for Tendrl/Gluster have been deployed as described in *Setup of Test Enviroment*.

- Configuration of USM QE tests has been done as explained in *Configuration before test run*.

1.6.2 How to run the tests

This is step by step description how to run the tests. First of all, login on QE Server as `usmqe` user:

```
$ ssh mbukatov.usmqe.example.com
[root@mbukatov ~]# su - usmqe
[usmqe@mbukatov ~]$
```

Note that the setup playbook prepared both setup and test repositories there:

```
[usmqe@mbukatov ~]$ ls -ld usmqe-*
drwxr-xr-x. 10 usmqe usmqe 4096 Dec 18 04:13 usmqe-setup
drwxr-xr-x. 11 usmqe usmqe 4096 Dec 18 03:50 usmqe-tests
```

You may consider updating these repositories (eg. running `git pull`) before going on, but this depends on the task you are working on.

Go to `usmqe-tests` repository and check that you have already updated the config file to match your environment (as noted in previous section):

```
[usmqe@mbukatov ~]$ cd usmqe-tests
[usmqe@mbukatov usmqe-tests]$ git status -s conf
 M conf/main.yaml
[usmqe@mbukatov usmqe-tests]$ cat conf/main.yaml
# List of configuration files loaded in usmqe test framework.
# It is recomended to keep defaults.yaml configuration file and attach more
# configuration files.
#
# Each item in list overwrites previous configuration file.
#
configuration_files:
  - conf/defaults.yaml
  - conf/mbukatov-usm1.yaml
inventory_file:
  - conf/mbukatov-usm1.hosts
```

File `mbukatov-usm1.hosts` is ansible inventory file, which describes my testing machines, while `mbukatov-usm1.yaml` is usmqe config file, used by the tests. I mention this here only to make it clear how it all fits together, for full details about configuration of USM QE tests, see references in previous section.

The tests are written in pytest framework, and we use custom wrapper `pytest_cli.py` to execute them. The wrapper is used to simplify test configuration (so that one doesn't have to repeat the same config values multiple times, as eg. inventory file is used by both pytest ansible plugin and test code itself).

This means that command to run the tests looks like this:

```
[usmqe@mbukatov usmqe-tests]$ ./pytest_cli.py [pytest_options] usmqe_tests/[file_or_
↪dir] ...
```

Note that running all the tests (`./pytest_cli.py usmqe_tests`) is not a good idea (there are various types of tests, including demo, and it never makes sense to just run them all), always specify at least directory or marker there.

Useful pytest options one can use are:

- `-m MARKEXPR` only run tests matching given mark expression
- `--pdb` start the interactive Python debugger on errors

- `-v` verbose mode
- `-s` turns off per-test capture logging, all logs are immediately reported on console (which is useful when developing new test code and immediate feedback is needed)
- `--last-failed` reruns only test cases which failed in a previous run
- `--junit-xml=testrun_foo.xml -o junit_suite_name=foo` istructs pytest to create xml junit log file named `testrun_foo.xm` and to set `foo` as a value of `/testsuite/@name` attribute in this xml file (you can then process the xml file to compare test runs and generate reports using tools such as junit2html)

1.6.3 Examples

This section contains few basic examples how to run the tests.

Get familiar with logging and test reporting

To get basic idea how usm qe test runs and error reporting looks like, one can run `usmqe_tests/demo` test suite. This demo should work even with default example configuration committed in the repository.

First of all, you can use `--collect-only` option of pytest to get list of test cases in the demo test module:

```
[usmqe@mbukatov usmqe-tests]$ ./pytest_cli.py --collect-only usmqe_tests/demo/
=================================== test session starts␣
↪===================================
platform linux -- Python 3.6.3, pytest-3.6.1, py-1.5.3, pluggy-0.6.0
rootdir: /home/usmqe/usmqe-tests, inifile: pytest.ini
plugins: ansible-playbook-0.3.0
collected 17 items
<Module 'usmqe_tests/demo/test_logging.py'>
  <Function 'test_pass_one'>
  <Function 'test_pass_many'>
  <Function 'test_pass_parametrized[a-1]'>
  <Function 'test_pass_parametrized[a-2]'>
  <Function 'test_pass_parametrized[a-3]'>
  <Function 'test_pass_parametrized[b-1]'>
  <Function 'test_pass_parametrized[b-2]'>
  <Function 'test_pass_parametrized[b-3]'>
  <Function 'test_pass_parametrized_fixture[1]'>
  <Function 'test_pass_parametrized_fixture[2]'>
  <Function 'test_fail_one_check'>
  <Function 'test_fail_many_check'>
  <Function 'test_fail_one_exception'>
  <Function 'test_error_in_fixture'>
  <Function 'test_xfail_one'>
  <Function 'test_xfail_many'>
  <Function 'test_fail_anyway'>
```

Then the test execution of the demo:

```
[usmqe@mbukatov usmqe-tests]$ ./pytest_cli.py usmqe_tests/demo/
```

In this case, only short summary of the test run is reported, along with full logs for test cases which failed. The logs here are using mrglog module. The output itself is too long to be included there. Moreover for full understanding, one is expected to check source code of the demo test module.

Note: Because of commit Improve the error handling, error in the test is catched in mrglog statistics. However there could be a discrepancy with pytest results. If the error/exception originates from the test method code and there is no failed check, pytest says the test FAILED at the same time mrlog evaluates the result of the test as ERROR.

Run all tests for alering

Assuming we have the machines and configuration ready and that we want junit xml report from the test run:

```
[usmqe@mbukatov usmqe-tests]$ ./pytest_cli.py --junit-xml=logs/result.xml usmqe_tests/
↪alerting
```

The xml junit file with the full test report will be then placed in logs/result.xm even if the logs directory didn't exist before.

1.7 Test Development

This is an overview of style and rules we follow when automating system functional test cases.

1.7.1 Code style

All python code in this project must be python 3 compatible. Moreover no python 2 compatibility layers or extensions should be added into the code, with the exceptions of pytest plugins (code in plugin directory).

We follow PEP 8 with a single exception regarding the maximum line length: we use 80 character as a soft limit so that one could break this rule if readability is affected, assuming the line length doesn't go over 100 characters (the hard limit).

1.7.2 Reading Configuration Values

To access USM QE configuration, use UsmConfig object:

```
from usmqe.usmqeconfig import UsmConfig

CONF = UsmConfig()

username = CONF.config["usmqe"]["username"]
```

Obviously this assumes that the username option has been specified in a config file which is referenced in conf/main.yaml file. The minimal yaml file for the previous example to work would look like this:

```
usmqe:
  username: admin
```

To access data from the host inventory, use functions provided by class InventoryManager from ansible.inventory.manager module. If inventory_file option is specified correctly in conf/main.yaml file, then instance of this class is available after loading the configuration under inventory key, e.g. CONF.inventory.

1.7.3 Unit Tests

We have unit tests of `usmqe-tests` project itself, which covers some code in `usmqe` module and runs flake8 checks on this module and the test code. One is encouraged to add unit tests when adding code into `usmqe` module and to run the tests before submitting a pull request.

Code related to unit testing:

- usmqe/unit_tests directory which contains pytest configuration (`pytest.ini`, `conftest.py`) and the code of unit tests itself
- tox.ini file
- .travis.yml config for Travis CI integration, uses tox

Unit test execution

To execute the unit tests, just run `tox` command in root directory of `usmqe-tests` repo.

Moreover the unit tests are executed for each new pull request via Travis CI.

1.7.4 Integration Tests

We have integration tests of `usmqe-tests` project itself, which covers some code in `usmqe` module. One is encouraged to add integration tests when adding code into `usmqe` module and to run the tests before submitting a pull request. Among these tests can be tests that use global configuration via `UsmConfig`.

usmqe/integration_tests is directory that contains the code of integration tests.

To execute the integration tests, just run `pytest` command in usmqe/integration_tests directory.

1.7.5 Structure of Functional Tests

Setup of *Gluster trusted storage pool(s)* is done prior test execution, and is fully automated via gdeploy config files in usmqe-setup/gdeploy_config/.

No *pytest fixture* or *test case* creates or removes *Gluster trusted storage pool(s)* on it's own.

The test cases are implemented as pytest functions, stored in logical chunks in python source files (python modules).

Test cases which requires an imported cluster (aka trusted storage pool), uses pytest fixture `imported_cluster`, which:

- Doesn't create the cluster, but just checks if the cluster is already imported and tries to import it if it's not imported already. If it fails during the import or no suitable cluster is available for import, it raises an errror.
- Cluster suitable for import is identified using node defined by `usm_cluster_member` parameter in usmqe configuration file.
- Returns information about the imported cluster via value of the fixture passed to the test function (`cluster` object), which includes cluster name, cluster id, volumes in the cluster.
- Teardown of this fixture runs cluster unmanage if the cluster was imported during setup phase.

Test casess are tagged by tags:

- Positive test case: `@pytest.mark.happypath`
- Negative test case: `@pytest.mark.negative`

- TODO: marker for gluster related tests
- TODO: marker for volume type
- TODO: marker for status of gluster profiling
- TODO: marker for human readable name
- marker for working/stable test - currently `mark.testready`
- TODO: marker for wip test?

Note: Open questions, enhancements:

- fixture to read markers and change import accordingly
- fixture to read markers and check if the cluster matches the requirements (eg. do we have specified volume type there?)
- multiple clusters

Tagging makes it possible to run for example just tests related to particular volume which requires profiling to be enabled.

All tests should use a proper pytest fixture for setup and teardown, if setup or teardown is needed. All objects created during testing should be removed after test run. The same applies for the fixtures, if something is created during setup phase, it should be removed during teardown. There should not be any remains after test run.

Exceptions

There are only 2 exceptions from the rules listed above.

Test cases which test import or unamanage cluster operations itself should not use `imported_cluster` fixture, but handle the import itself in the code of the test case.

Such cases should be stored in separate module (python source file) so that it could be part of separate test runs.

The same would apply for **CRUD happy path tests**, which are stored in one python source file where they share object created and deleted during testing tests from file. These tests should run in same order like they are written in the file. Such cases are run at the beginning of testing because they left created/imported clusters for further testing. This exception exists because cluster creation have extremly big resource needs.

Note: Note that we don't have any CRUD happy path tests and are not going to have them untill we need to test day 1 or day 2 operations, which includes creating or deleting gluster clusters, volumes or other cluster components.

CHAPTER 2

Quick Introduction

USM QE tests are:

- ***integration tests* - which means that:**
  - all Tendrl components are tested together in production like configuration
  - separate machine for each machine role is needed, and when more machines are expected to be assigned to a cluster role, we need to have at least 2 or 4 such machines for the role (depends on the role itself)
  - user facing interfaces are tested (eg. we use Tendrl REST API, but don't interfere with internal structure of *Tendrl Data Store*)
- used in *fully automated* enviroment (including setup of all machines, which is automated via ansible)
- based on pytest framework, which is extended or changed via many pytest plugins
- written in Python 3.6 only
- expected to be executed from Red Hat (or CentOS) 7 machine
- maintained by *USM QE Team*

CHAPTER 3

License

Distributed under the terms of the GNU GPL v3.0 license, usmqe-tests is free and open source software.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search