# Meta Test Family Documentation

**Petr Hracek** **Jan Scotka**

**Sep 27, 2018**

# Contents

Welcome to the Meta Test Family documentation!

CHAPTER 1

# About

**M**eta **T**est **F**amily (MTF) is a tool to test components of a modular Fedora.

Using MTF you can:

- write tests for RPMs, modules and Docker containers
- write multiline Bash snippet tests in YAML definition file
- write multihost tests
- write Bash tests
- write Python tests
- schedule tests with Jenkins and Taskotron
- run tests on a local host or in Vagrant environment

MTF has a presence on the following websites:

- Documentation is available on ReadTheDocs.
- A Package repository is available on Fedora Copr.
- MTF's code and the issue tracker for sharing bugs and feature ideas are stored on GitHub.

CHAPTER 2

# Content

## 2.1 Installation

There are two ways to install and use MTF: to set up it locally or alternatively on a virtual machine via the Vagrant tool.

**Topics**



### 2.1.1 Vagrant

Vagrant is a tool to aid developers in quickly deploying development environments. There is a Vagrantfile in the meta-test-family git repository on GitHub that can automatically deploy a virtual machine on your host with a MTF environment configured.

The MTF tool has been made available for use of two providers: `libvirt` (for Linux host only) and `virtualbox` (for MAC OS, Windows and Linux hosts), where `libvirt` is a default one. See more about Vagrant providers here.

This document assumes that you are running a recent version of Fedora although these steps should be roughly the same on other distributions, just be aware that package managers and names can differ if you are not using Fedora as

your host. Consult Vagrant installation documentation to set up Vagrant for a different platform and adjust the steps of this document accordingly.

**Note:** Before you start using Vagrant-libvirt, please make sure your libvirt and qemu installation is working correctly and you are able to create qemu or kvm type virtual machines with virsh or virt-manager.

### Prerequisites for Vagrant

1. Install Vagrant. Ensure that `vagrant-libvirt` is among pulled dependencies.

```
# install Vagrant
$ sudo dnf -y install vagrant
```

2. Start `libvirtd` service

```
# start libvirtd service
$ sudo systemctl start libvirtd
```

### Creating the Vagrant environment

After preparing the libvirt prerequisites using the instructions above:

1. You are now prepared to check out the MTF code into your preferred location.

```
# cd to your prefered location
$ cd $HOME/  # Season to taste.
$ git clone https://github.com/fedora-modularity/meta-test-family.git
```

2. Next, enter into the `meta-test-family` directory.

```
# cd in meta-test-family
$ cd meta-test-family
```

3. The MTF tool provides a configuration Vagrantfile that you can use to configure the Vagrant environment as given or open the Vagrantfile in your favorite editor and modify it to better fit your development preferences. This step is entirely optional as the default Vagrantfile should work for most users.

```
# vim Vagrantfile
$ vim Vagrantfile
```

4. If you've happy with the Vagrantfile, you can begin provisioning your Vagrant environment. Finish by running `vagrant reload` to reboot machine after provisioning and apply the latest kernel updates.

```
# Provision the Vagrant environment:
$ sudo vagrant up --provider=libvirt # or just `sudo vagrant up` as libvirt is a␣
↪default one
# Alternatively, set the TARGET envvar to test another target defined in examples/
↪testing-module/Makefile
$ sudo TARGET=check-pure-docker vagrant up
# The above will run for a while while it provisions your development environment.
$ sudo vagrant reload  # Reboot the machine at the end to apply kernel updates, etc.
```

5. Once you have followed the steps above, you should have a running deployed MTF development machine. Log into your Vagrant environment:

```
# ssh into the Vagrant environment
$ sudo vagrant ssh
```

### 2.1.2 Local installation

#### Requirements

MTF installer pulls its latest dependencies: `python-devel`, `python-setuptools` and `python-netifcaes`, `docker`, avocado, `yaml` and `json`.

MTF supports Gherkin-based testing in Python. To write tests in a natural language style, backed up by Python code, install the BBD tool behave . Execute the following command to install behave with pip:

```
# install behave
$ sudo pip install behave
```

#### Installing MTF

Install MTF rpm from Fedora Copr repo.

```
# add meta-test-family yum repo
$ sudo dnf copr enable phracek/meta-test-family
$ sudo dnf install -y meta-test-family
```

MTF scripts, examples and documentation will be installed into `/usr/share/moduleframework`

### 2.1.3 Source code

You may also wish to follow the GitHub MTF repo if you have a GitHub account. This stores the source code and the issue tracker for sharing bugs and feature ideas. The repository should be forked into your personal GitHub account where all work will be done. Any changes should be submitted through the pull request process. Please see Contributing Guidelines for more information.

**See also:**

***User Guide*** User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

## 2.2 User Guide

1. In a module's root directory create a directory `tests` and place there a module configuration file `config.yaml` described in detail in section Configuration file. If you would like to use MTF without your own `config.yaml`. It is possible. It uses default minimal config. Then you have to set `URL` envvar to set *test subject*, otherwise it causes traceback. It is usefull for example for module what does not provide any service (no own `start/stop/status/etc` action defined.) or for testing with modulelint.
2. Optionally write multiline Bash snippet tests directly in the `tests/config.yaml` file as described in section Multiline Bash snippet tests.
3. Check the list of Environment variables.

4. Write your tests, for example see sanity tests and various tests examples in `/usr/share/moduleframework/examples/testing-module/`. All tests methods are listed in section API Index and alphabetically in genindex section.

5. In the directory `tests` create a `Makefile` as below.

   Mind to keep the `mtf-generator` line only if there are multiline Bash snippet tests in the `tests/config.yaml` file. The `mtf-generator` command will convert those multiline Bash snippet tests from the `tests/config.yaml` file into Python tests and stores them in the `tests/generated.py` file, which will be processed further by avocado.

```
MODULE_LINT=/usr/share/moduleframework/tests/generic/*.py
TESTS=*.py
CMD=avocado run $(MODULE_LINT) $(TESTS)

#
all:
    mtf-generator
    $(CMD)
```

6. In a module's root directory create a `Makefile`, which contains a secton **test**. For example:

```
.PHONY: build run default

IMAGE_NAME = debugging-tools
MODULEMDURL=file://debugging-tools.yaml

all: run
default: run

build:
    docker build --tag=$(IMAGE_NAME) .

run: build
    docker run -it --name $(IMAGE_NAME) --privileged --ipc=host --net=host --
↪pid=host -e HOST=/host -e NAME=$(IMAGE_NAME) -e IMAGE=$(IMAGE_NAME) -v /
↪run:/run -v /var/log:/var/log -v /etc/machine-id:/etc/machine-id -v /etc/
↪localtime:/etc/localtime -v /:/host $(IMAGE_NAME)

test: build
    cd tests; MODULE=docker MODULEMD=$(MODULEMDURL) URL="docker=$(IMAGE_NAME)
↪" make all
    cd tests; MODULE=nspawn MODULEMD=$(MODULEMDURL) make all
    cd tests; MODULE=openshift OPENSHIFT_IP="127.0.0.1" OPENSHIFT_USER=
↪"developer" OPENSHIFT_PASSWORD="developer" make all
```

7. Prepare the environment to run tests in.

8. Execute tests from the module root directory by running

```
#run tests from a module root directory
$ make test
```

   or from the `tests` directory by running

```
#run Python tests from the tests/ directory
$ sudo MODULE=docker mtf ./*.py
```

   or

```
#run Bash tests from the tests/ directory
$ sudo MODULE=docker mtf ./*.sh
```

9. Clean up the environment after test execution.

Contents:

### 2.2.1 Configuration file

To test a module create its configuration file `config.yaml` similar to an example configuration file described further. If the tested module doesn't represent any service, the minimal configuration file structure can be used.

An example of `config.yaml` header:

```
document: modularity-testing
version: 1
```

An example of module general description:

```
name: memcached
modulemd-url: http://raw.githubusercontent.com/container-images/memcached/master/
↪memcached.yaml
compose-url: https://kojipkgs.fedoraproject.org/compose/latest-Fedora-Modular-26/
↪compose/Server/x86_64/os/Packages/m/memcached-1.4.36-1.module_b2e063be.x86_64.rpm
service:
    port: 11211
packages:
    rpms:
        - memcached
        - perl-Carp
testdependencies:
    rpms:
        - nc
```

- **name** defines module name
- **modulemd-url** contains a link to a moduleMD file
- **compose-url** links to a final compose Pungi build. **repo** or **repos** can be used instead, see further
- **service** stores a port if a module has any
- **packages** defines a module type (at the moment only *rpms* type is supported)
- **testdependencies** covers dependencies to be installed and used in tests

An example of module types specification:

```
default_module: docker
module:
    openshift:
        template: ./memcached.yaml
        docker_pull: True
        container: docker.io/modularitycontainers/memcached
    docker:
        setup: "docker run -it -e CACHE_SIZE=128 -p 11211:11211"
        cleanup:"echo Cleanup magic"
        labels:
            description: "memcached is a high-performance, distributed memory"
```

(continues on next page)

(continued from previous page)

```
            io.k8s.description: "memcached is a high-performance, distributed memory"
        source: https://github.com/container-images/memcached.git
        url: docker.io/phracek/memcached
    rpm:
        setup: /usr/bin/memcached -p 11211
        cleanup: echo Cleanup magic
        start: systemctl start memcached
        stop: systemctl stop memcached
        status: systemctl status memcached
        url: http://download.englab.brq.redhat.com/pub/fedora/releases/25/Everything/
↪x86_64/os/
    inheriteddocker:
        parent: docker
        start: "docker run -it -p 11211:11211"
```

- **default_module**, if specified, sets the default tested module type
- **setup** runs setup commands on a host machine, not in container, and prepares the environment for tests, for example changes selinux policy or hostname
- **cleanup**: similar to setup but done after test finished
- **start** defines how to start module service if there is any
- **stop** defines how to stop module service if there is any
- **status** defines how to check the status of module service if there is any
- **labels** contains docker labels to check if there is any
- **url** contains link to a container or repo (same meaning as container or repo)
- **container** contains a link to a container (docker.io or local tar.gz file) (obsolete)
- **repo** is used when **compose-url** is not set and contains a repo to be used for rpm module type testing (obsolete)
- **parent** if you would like to have more configs for same module type, it is possible to do it via inheritance. There will be used parent module + overwritten values with this one, you can rewrite whatever you want. You have to set parent (base) module type allowed are just **rpm/docker**
- **template** contains an URL link or a path to an OpenShift template. The template is added into OpenShift resources, like template and new application is created based on the template. The *template* is used to deploy your application inside OpenShift using command *oc new-app* ...
- **docker_pull** specifies if image is pulled by command *docker pull* or not before adding to an OpenShift registry. Disabling this prevents your local image being overwritten. If it is not present then default value is *True*. You can specify 'True' or 'False'.

### Multiline Bash snippet tests

A `config.yaml` file may contain multiline Bash snippet tests directly. Every Bash command has to finish with 0 return code otherwise it returns fail:

```
test:
    processrunning:
        - 'ls  /proc/*/exe -alh | grep memcached'
testhost:
    selfcheck:
        - 'echo errr | nc localhost 11211'
```

(continues on next page)

(continued from previous page)

```
        - 'echo set AAA 0 4 2 | nc localhost 11211'
        - 'echo get AAA | nc localhost 11211'
    selcheckError:
        - 'echo errr | nc localhost 11211 |grep ERROR'
```

- **test** defines a section of multiline bash snippet tests
- **processrunning** contains commands to run as tests and displayed as avocado output
- **testhost** is optional and similar to **test**. The difference is that it runs commands on host machine so that there could be more dependencies than there are just in a module.

**See also:**

***User Guide*** User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

### 2.2.2 Enviroment setup

To test a particular component (docker, rpm or nspawn) the test environment should be configured accordingly, e.g. certain dependencies should be installed or some services should be started. There is an option to do it manually or by using MTF scripts.

#### Manual Setup

**Docker**

- Install Docker if not installed
- Add insecure registry to config if not added for your testing images
- (Re)Start docker service

**Nspawn**

- Install systemd-nspawn
- Disable selinux if enabled. It is an issue in selinux-policy

**Rpm**

- No any configuration needed

**OpenShift**

- Install OpenShift if not installed and if environment variable `OPENSHIFT_LOCAL` is specified.
- if `OPENSHIFT_LOCAL` variable is specified, then it starts an OpenShift by command `oc cluster up` or stops it by command `oc cluster down`.

#### Automated Setup

The environment configuration scripts should be executed in the same directory where the tests are, otherwise the environment variable **CONFIG** should be set.

- to setup environment run `MODULE=docker mtf-env-set`
- to execute tests run `MODULE=docker mtf your.test.py`

- to cleanup environment `MODULE=docker mtf-env-clean`

#### Test Creation

There is a script called mtf-init which generates easy template of test for module docker as example.

- to create template for module docker `mtf-init --name your_name --container path_to_your_container`

### 2.2.3 Environment variables

Environment variables allow to overwrite some values of a module configuration file `config.yaml`.

- **AVOCADO_LOG_DEBUG=yes** enables avocado debug output.
- **DEBUG=yes** enables debugging mode to test output.
- **CONFIG** defines the module configuration file. It defaults to `config.yaml`.
- **MODULE** defines tested module type, if **default-module** is not set in `config.yaml`.
  - **=docker** uses the **docker** section of `config.yaml`.
  - **=rpm** uses the **rpm** section of `config.yaml` and tests RPMs directly on a host.
  - **=nspawn** tests RPMs in a virtual environment of lightweight virtualization with systemd-nspawn.
- **URL** overrides the value of **module.docker.container** or **module.rpm.repo**. The **URL** should correspond to the **MODULE** variable, for example
  - **URL=docker.io/modularitycontainers/haproxy** if **MODULE=docker**
  - **URL=https://phracek.fedorapeople.org/haproxy-module-repo** if **MODULE=nspawn** or **MODULE=rpm**
- **MODULEMDURL** overwrites the location of a moduleMD file.
- **COMPOSEURL** overwrites the location of a compose Pungi build.
- **MTF_SKIP_DISABLING_SELINUX=yes** does not disable SELinux. In nspawn type on Fedora 25 SELinux should be disabled, because it does not work well with SELinux enabled, this option allows to not do that.
- **MTF_DO_NOT_CLEANUP=yes** does not clean up module after tests execution (a machine remains running).
- **MTF_REUSE=yes** uses the same module between tests. It speeds up test execution. It can cause side effects.
- **MTF_REMOTE_REPOS=yes** disables downloading of Koji packages and creating a local repo, and speeds up test execution.
- **MTF_DISABLE_MODULE=yes** disables module handling to use nonmodular test mode (see multihost tests as an example).
- **DOCKERFILE="<path_to_dockerfile"** overwrites the location of a Dockerfile.
- **HELPMDFILE="<path_to_helpmdfile"** overwrites the location of a HelpMD file, If not set, search for mdfile in same directory where is Dockerfile.
- **OPENSHIFT_LOCAL=yes** enables installing `origin` and `origin-clients` on local machine
- **OPENSHIFT_IP=openshift_ip_address** uses this IP address for connecting to an OpenShift environment.
- **OPENSHIFT_USER=developer** uses this `USER` name for login to an OpenShift environment.
- **OPENSHIFT_PASSWORD=developer** uses this `PASSWORD` name for login to an OpenShift environment.

- **MTF_ODCS=[yes|openIDCtoken_string]** enable ODCS for compose creation. Token has to be placed or it tries contact openIDC token via your web browser. **Experimental feature**

**See also:**

*User Guide* User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

### 2.2.4 Workflow integration

#### Testsuite of project

- **Upstream testsuite for project located in */usr/share/moduleframework/examples/testing-module/***
  - You can use it as an **inspiration** for your tests
  - It contains various types how to schedule tests
  - **CI** It contains info how it is scheduled in internal-ci or in taskotron or how to do
  - Examples of **Manual** running of tests on localhost.
  - Example how to run general **multi-host** tests
  - Every new feature should be covered here - by new Makefile target or by new test run inside every testing module

#### Taskotron Wokflow

- **Production instance: https://taskotron.fedoraproject.org/resultsdb/results?testcases=dist.modularity-testing-framework**
  - Triggered fedmsg via **module-stream-version** string
  - Triggered by **Module Build system** done message, list of all: https://apps.fedoraproject.org/datagrepper/raw?topic=org.fedoraproject.prod.mbs.module.state.change
  - **There is general *runtask.yml* taskotron trigger: https://pagure.io/taskotron/task-modularity-testing-framework**
    - There is just one for every module and it contains whole logic where to find tests for module.
    - Not needed to duplicate *runtash.yml* for each component. Scheduler is same (existing Makefile)
    - It run *tools/run-them.sh* script. It contains whole logic where are tests and how to find them.
  - **run-them.sh script for taskotron**
    - Test Subject: rpm repositories (tagged koji builds of packages) via *systemd-nspawn*
    - Located in: */usr/share/moduleframework/tools/run-them.sh*
    - Scheduled as: *./run-them.sh testmodule testmodule-master-20170407121558 pdc*
    - Example targets: *check-run-them-pdc-testmodule, check-run-them-pdc-baseruntime*
    - **Internal logic**
      - Contact *PDC* (Product definition center) for info about module like *koji tags, moduleMD file*
      - **Try dowload package from *modules* namespace in *dist-git* via *fedpkg clone***

checkout to proper version found by PDC (scmurl)

Try to find tests there ( if exist *Makefile* in *tests* directory)

- If None: Try to find module dir in MTF project tests in */usr/share/moduleframework/examples* directory
- If None: Run at least general ModuleLinter (*/usr/share/moduleframework/tests/modulelint*) with general minimal config.yaml located in *docs* directory

#### Arbitrary Jenkins Instance

- **Production instance:** ***hidden***
  - **Triggered via** ***fedmsg file***
    - Used **tools/run-them.sh** script, for same behaviour as Taskotron
  - **run-them.sh script for Jenkins based on whole fedmsg**
    - Test Subject: Same as *Taskotron Workflow*
    - Located in: Same as *Taskotron Workflow*
    - Scheduled as: *run-them.sh testmodule /usr/share/moduleframework/tools/example_message_module.yaml fedmsg*
    - Example targets: *check-run-them-fedmsg-testmodule*
    - **Internal logic**
      - Same as *Taskotron Workflow*

### 2.2.5 Linters

MTF provides a set of linters for checking containers, help files and Dockerfiles.

#### Dockerfile linters

Dockerfile linters are divided into two python files: dockerlint.py and dockerfile_lint.py.

dockerlint.py performs these checks on an existing container image:

- **test_all_nodocs** checks if documentation files shipped by installed RPM packages have been removed. They are usually installed in the `base image` and inherited by child layer or installed via the `RUN` instruction. This is only `WARN` check.
- **test_installed_docs** checks if RPM packages installed by the `RUN dnf` command also install documentation files. The `base image` is an exception.
- **test_clean_all** checks if `dnf/yum clean all` is present in Dockerfile.

dockerfile_lint.py these checks are performed on a Dockerfile:

- **test_from_is_first_directive** checks if the FROM instruction is really first in the Dockerfile.
- **test_from_directive_is_valid** checks if the FROM instruction has proper format.
- **test_chained_run_dnf_commands** checks if dnf/yum commands are chained or not.
- **test_checked_run_rest_commands** checks if the RUN instructions, except dnf/yum, are chained or not.
- **test_helpmd_is_present** checks if the help file is present for this container.

- **test_architecture_label_exists** checks if the architecture label is present in the Dockerfile.
- **test_name_in_env_and_label_exists** checks if the name label is present in the Dockerfile and NAME is present as ENV variable.
- **test_maintainer_label_exists** checks if the maintainer label is present in the Dockerfile.
- **test_release_label_exists** checks if the release label is present in the Dockerfile.
- **test_version_label_exists** checks if the version label is present in Dockerfile.
- **test_com_redhat_component_label_exists** checks if the com.redhat.component label is present in the Dockerfile.
- **test_summary_label_exists** checks if the summary label is present in the Dockerfile.
- **test_run_or_usage_label_exists** check if the run or usage label is present in the Dockerfile.

### Help file linter

Help file linter checks if the help.md file contains important sections. Help file linter is helpmd_lint.py.

Example of such help.md file is:

```
% MEMCACHED(1) Container Image Pages
% Petr Hracek
% February 6, 2017
# NAME
# DESCRIPTION
# USAGE
# SECURITY IMPLICATIONS
```

helpmd_lint.py contains those checks inside help.md file:

- **test_helpmd_image_name** checks if the help file contains an image name. The correct format is e.g. `% MEMCACHED(1)`.
- **test_helpmd_maintainer_name** checks if the help file contains a maintainer name. The correct format is e.g. `% USER NAME`.
- **test_helpmd_name** checks if the help file contains a section called `# NAME`. The section describes name of the container and short description.
- **test_helpmd_description** checks if the help file contains a section called `# DESCRIPTION`. This sections describes how to use image, etc.
- **test_helpmd_usage** checks if the help file contains a section called `# USAGE`.
- **test_helpmd_environment_variables** checks if the help file contains a section called `# ENVIRONMENT VARIABLES`. The check is valid only if ENV variable are present in the Dockerfile. There is no heuristic if the variable is the same as specified in the helper file.
- **test_helpmd_security_implications** checks if the help file contains a section called `# SECURITY IMPLICATIONS`. The check is valid only if container exposes a port. There is no heuristic if the exposed port is the same as specified in the help file.

**See also:**

***User Guide*** User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

### 2.2.6 Glossary

**Module** A set of packages tested and released together as a distinct unit, complete with the metadata needed to manage it as a unit. May depend on other modules.

### 2.2.7 Troubleshooting

#### First test takes so long time

It is expected behavior, because the first test run downloads all packages from Koji and creates a local repo. It is workaround because of missing composes for modules (on demand done by pungi). To make tests execute faster use environment variables:

- **MTF_DO_NOT_CLEANUP=yes** does not clean up module after tests execution (a machine remains running).
- **MTF_REUSE=yes** uses the same module between tests. It speeds up test execution. It can cause side effects.
- **MTF_REMOTE_REPOS=yes** disables downloading of Koji packages and creating a local repo, and speeds up test execution.

#### Unable to debug avocado output errors

If you see an error: `Avocado crashed: TestError: Process died before it pushed early test_status.`, add environment variables:

- **AVOCADO_LOG_DEBUG=yes**
- **DEBUG=yes**

**See also:**

***API Index*** API Index

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

## 2.3 Manual testing

- **Scheduled on host machine**
  - **docker, nspawn** *MODULE* type does not affect Host machine
  - **rpm** *MODULE* type test directly on host machine. It installs things there and may be **very dangerous**
- Intended for test debugging

### 2.3.1 modules dist-git integration

- **dist-git - Create Makefile in top directory, what contains build target and test target dependant on build taret and set prop**
  - Example: https://github.com/container-images/container-image-template/blob/master/Makefile
  - Makefile like: *cd tests; MODULEMDURL=$(MODULEMDURL) MODULE=docker URL="docker=$(IMAGE_NAME)" make all*

  - inside *tests* directory cretates just simple Makefile like https://github.com/container-images/container-image-template/blob/master/tests/Makefile

### 2.3.2 Docker

- Test Subject: docker images, https://fedoraproject.org/wiki/Docker
- **Scheduled as:** ***MODULE=docker avocado run *.py modulelint/*.py***
  - or *MODULE=docker CONFIG=./minimal.yaml avocado run *.py modulelint/*.py* when you use alternate configuration file
- Example targets: *check-docker, check-minimal-config-docker, check-behave-docker*
- **Internal logic of testing**
  - pull docker image
  - setup environment
  - start docker image via start section or default one (keep it running)
  - do test
  - cleanup enviroment
  - remove docker container

### 2.3.3 Nspanw

- Test Subject: rpm repository, inside systemd-nspawn virtualization https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html
- **Scheduled as:** ***MODULE=nspawn avocado run *.py modulelint/*.py***
  - or *MODULE=nspawn CONFIG=./minimal.yaml avocado run *.py modulelint/*.py* when you use alternate configuration file
- Example targets: *check-rpm, check-minimal-config-rpm*
- **Internal logic of testing**
  - install packages to *changeroot* with systemd
  - setup environment and *boot* nspawn machine (to keep it running)
  - start via start section or default one on *guest*
  - do test
  - cleanup enviroment
  - halt system and remove installed chroot dir

### 2.3.4 Rpm

- **Destructive** and **WIP**
- Test Subject: rpm repository, bare metal, intended for testing packages directly on machine (without any module)
- **Scheduled as:** ***MODULE=rpm avocado run *.py modulelint/*.py***
  - or *MODULE=rpm avocado run *.py modulelint/*.py* when you use alternate configuration file

- Example targets: *None* - cause changes on host
- **Internal logic of testing**
  - install packages to system
  - start via start section or default one
  - do test
  - cleanup enviroment if any

### 2.3.5 Multihost

- Test Subject: any of previous
- Could be used for general multihost testing not directly dependent on modules
- Scheduled as: *cd /usr/share/moduleframework/examples/multios_testing; MTF_DISABLE_MODULE=yes avocado run *.py*
- Example targets: *check-multihost-testing*
- **Internal logic of testing**
  - could be same as previous ones that there is one *Host* and one *Guest* machine what can cooperate togetger
  - **Or it could be used for general multihost testing with *N* machines where *N>1* via use backends directly in setU**
    - see example of test: https://github.com/fedora-modularity/meta-test-family/blob/master/examples/multios_testing/sanityRealMultiHost.py
    - this example creates 3 machines *(using nspawn)* with various fedora versions and gather data.

## 2.4 MTF - Levels of testing

### 2.4.1 Component level testing

- **WIP**
- **Test Subject** - RPM packages build by koji
- See sections Manual testing - *Rpm* or *Multihost*
- MTF could be used for component level testing, it is **not primar purpose** of this project

### 2.4.2 Module level testing

- **Test Subject** - Module Build (rpm packages produced by MBS and tagged by koji or Docker container created manually or by OSBS or similar service)
- See sections Manual testing - *Docker Nspawn*
- **This is primar purpose of this framework**
  - tagged rpm packages are not final artifacts (Module Compose should be final artifact) - for now it supply Compose level testing

  - Docker image is final build artifacts

### 2.4.3 Compose level testing

- **WIP**
- **Test Subject:** Module compose (done by Pungi https://pagure.io/pungi-fedora)
- We are waiting for real module composes, what will be able to provide data about modules (modulemd files, repositories)
- It does not exist yet.
- There should be service for module builds on demand, not just composes for all modules together
- MTF is prepared for *Compose testing* somehow
- **How to:**
  - remove modulemd-url from config use COMPOSE env variable or compose-url inside config.yaml.
  - it gets all data from compose info
  - Scheduled as: *MODULE=nspawn COMPOSEURL=https://kojipkgs.stg.fedoraproject.org/compose/branched/jkaluza Fedora-Modular-26/compose/base-runtime/x86_64/os/ avocado run *.py*

## 2.5 How to Contribute

If you are interested in contributing to MTF, the best way is to report a bug or propose a new feature that interests you and submit a patch for it. We use GitHub Issues for sharing bugs and feature ideas. The MTF GitHub repository should be forked into your personal GitHub account where all work will be done. Any changes should be submitted through the pull request process. For more information on how to contibute, please read our Contributing Guidelines.

MTF is written in Python. While you do not need to be a Python expert to contribute, it would be advantageous to run through the Python tutorial if you are new to Python or programming in general.

Some knowledge of git and GitHub is useful as well. Documentation on both is available on the GitHub help page .

### 2.5.1 Contribution Checklist

1. Make sure that you choose the appropriate upstream branch.
2. Check your code with pylint and pyflakes
3. Please ensure that your code follows our style guide.
4. Please make sure that any new features are documented and that changes are reflected in existing docs.
5. Please squash your commits and use our commit message guidelines.
6. Make a PR!

Thank you!

**See also:**

***User Guide*** User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

## 2.6 API Index

This section contains the MTF API, auto generated from the source code.

### 2.6.1 Module Framework

### 2.6.2 Common

### 2.6.3 Exceptions

### 2.6.4 MTF Generator

### 2.6.5 Module for reading data from compose

### 2.6.6 PDC data

### 2.6.7 Docker

### 2.6.8 BASH Helper

### 2.6.9 Timeoutlib

**class** moduleframework.timeoutlib.**NOPTimeout**(**args*, ***kwargs*)

**class** moduleframework.timeoutlib.**Retry**(*attempts=1*, *timeout=None*, *exceptions=(<type 'exceptions.Exception'>, )*, *error=None*, *inverse=False*, *delay=None*)

**handle_failure**(*start_time*)

**class** moduleframework.timeoutlib.**Timeout**(*retry*, *timeout*)

**See also:**

***User Guide*** User Guide

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channel on freenode IRC.

## 2.7 License

MTF is released under the GPLv2+, see LICENSE file in the source code repository.

CHAPTER 3

# Index and Search

- genindex
- search

**See also:**

**webchat.freenode.net** Questions? Help? Ideas? Stop by the #fedora-modularity chat channelon freenode IRC.

# Python Module Index

## m

# Index