fmf

May 18, 2021

Contents

The `fmf` Python module and command line tool implement a flexible format for defining metadata in plain text files which can be stored close to the source code. Thanks to hierarchical structure with support for inheritance and elasticity it provides an efficient way to organize data into well-sized text documents.

CHAPTER 1

Table of Contents

1.1 fmf

Flexible Metadata Format

1.1.1 Description

The `fmf` Python module and command line tool implement a flexible format for defining metadata in plain text files which can be stored close to the source code and structured in a hierarchical way with support for inheritance.

Although the proposal initially originated from user stories centered around test execution, the format is general and thus can be used in broader scenarios, e.g. test coverage mapping.

Using this approach it's also possible to combine both test execution metadata and test coverage information. Thanks to elasticity and hierarchy it provides ability to organize data into well-sized text documents while preventing duplication.

1.1.2 Synopsis

Command line usage is straightforward:

```
fmf command [options]
```

There are following commands available:

```
fmf ls      List identifiers of available objects
fmf show    Show metadata of available objects
fmf init    Initialize a new metadata tree
fmf clean   Remove cache directory and its content
```

1.1.3 Examples

List names of all objects stored in the metadata tree:

```
fmf ls
```

Show all test metadata (with 'test' attribute defined):

```
fmf show --key test
```

Show metadata for all tree nodes (not only leaves):

```
fmf show --key test --whole
```

List all attributes for the `/recursion` tests:

```
fmf show --key test --name /recursion
```

Show all covered requirements:

```
fmf show --key requirement --key coverage
```

Search for all tests with the `Tier1` tag defined and show a brief summary of what was found:

```
fmf show --key test --filter tags:Tier1 --verbose
```

Use arbitrary Python expressions to access deeper objects and create more complex conditions:

```
fmf show --condition "execute['how'] == 'shell'"
```

Initialize a new metadata tree in the current directory:

```
fmf init
```

Check help message of individual commands for the full list of available options.

1.1.4 Options

Here is the list of the most frequently used options.

Select

Limit which metadata should be listed.

--key=KEYS	Key content definition (required attributes)
--name=NAMES	List objects with name matching regular expression
--filter=FLTRS	Apply advanced filter when selecting objects
--condition=EXPR	Use arbitrary Python expression for filtering
--whole	Consider the whole tree (leaves only by default)

For filtering regular expressions can be used as well. See `pydoc fmf.filter` for advanced filtering options.

Format

Choose the best format for showing the metadata.

--format=FMT Custom output format using the {} expansion

--value=VALUES Values for the custom formatting string

See online documentation for details about custom format.

Utils

Various utility options.

--path PATHS Path to the metadata tree (default: current directory)

--verbose Print additional information standard error output

--debug Turn on debugging output, do not catch exceptions

Check help message of individual commands for the full list of available options.

1.1.5 Install

The fmf package is available in Fedora and EPEL:

```
dnf install fmf
```

Install the latest version from the Copr repository:

```
dnf copr enable psss/fmf
dnf install fmf
```

or use PIP:

```
pip install fmf
```

See documentation for more details about installation options.

1.1.6 Variables

Here is the list of environment variables understood by fmf:

FMF_CACHE_DIRECTORY Directory used to cache git clone calls for fmf identifiers.

1.1.7 Links

Git: https://github.com/psss/fmf

Docs: http://fmf.readthedocs.io/

Issues: https://github.com/psss/fmf/issues

Releases: https://github.com/psss/fmf/releases

Copr: http://copr.fedoraproject.org/coprs/psss/fmf

PIP: https://pypi.org/project/fmf/

Travis: https://travis-ci.org/psss/fmf

Coveralls: https://coveralls.io/github/psss/fmf

1.1.8 Authors

Petr Šplíchal, Miro Hrončok, Jakub Krysl, Jan Ščotka, Alois Mahdal, Cleber Rosa, Miroslav Vadkerti, Lukáš Zachar and František Nečas.

1.1.9 Copyright

1.2 Concept

In order to keep test execution efficient when number of test cases grows, it is crucial to maintain corresponding metadata, which define some aspects of how the test coverage is executed.

This tool implements a flexible format for defining metadata in plain text files which can be stored close to the test code and structured in a hierarchical way with support for inheritance.

Although the proposal initially originated from user stories centered around test execution, the format is general and thus can be used in broader scenarios, e.g. test coverage mapping.

Using this approach it's also possible to combine both test execution metadata and test coverage information. Thanks to elasticity and hierarchy it provides ability to organize data into well-sized text documents while preventing duplication.

1.2.1 Stones

These are essential corner stones for the design:

- Text files under version control
- Keep common uses cases simple
- Use hierarchy to organize content
- Prevent duplication where possible
- Metadata close to the test code
- Solution should be open source
- Focus on essential use cases

1.2.2 Stories

Important user stories to be covered:

- As a tester or developer I want to easy read and modify metadata and see history.
- As a tester I want to select a subset of test cases for execution by specifying a tag.

- As a tester I want to define a maximum time for a test case to run.
- As a tester I want to specify which environment is relevant for testing.
- As a user I want to easily define common metadata for multiple cases to simplify maintenance.
- As a user I want to provide specific metadata for selected tests to complement common metadata.
- As an individual tester and test contributor I want to execute specific single test case.
- As an automation tool I need a metadata storage with good api, extensible, quick for reading.

1.2.3 Choices

These choices have been made:

- Use git for version control and history of changes.
- Yaml format easily readable for both machines and humans.

1.2.4 Files

A dedicated file name extension `fmf` as an abbreviation of Flexible Metadata Format is used to easily find all metadata files on the filesystem:

- smoke.fmf
- main.fmf

Special file name `main.fmf` works similarly as `index.html`. It can be used to define the top level data for the directory.

1.2.5 Attributes

The format does not define attribute naming in any way. This is up to individual projects. The only exception is the special name `main` which is reserved for main directory index.

Attribute namespacing can be introduced as needed to prevent collisions between similar attributes. For example:

- test-description, requirement-description
- test:description, requirement:description
- test_description, requirement_description

1.2.6 Trees

Metadata form a tree where inheritance is applied. The tree root is defined by an `.fmf` directory (similarly as `.git` identifies top of the git repository). The `.fmf` directory contains at least a `version` file with a single integer number defining version of the format.

1.2.7 Names

Individual tree nodes are identified by path from the metadata root directory plus optional hierarchy defined inside yaml files. For example, let's have the metadata root defined in the `wget` directory. Below you can see node names for different files:

Location	Name
wget/main.fmf	/
wget/download/main.fmf	/download
wget/download/smoke.fmf	/download/smoke

1.2.8 Identifiers

Node names are unique across the metadata tree and thus can be used as identifiers for local referencing across the same tree. In order to reference remote fmf nodes from other trees a full `fmf identifier` is defined as a dictionary containing keys with the following meaning:

url Git repository containing the metadata tree. Use any format acceptable by the `git clone` command. Optional, if no repository url is provided, local files will be used.

ref Branch, tag or commit specifying the desired git revision. This is used to perform a `git checkout` in the repository. If not provided, the `default branch` is used.

path Path to the metadata tree root. Should be relative to the git repository root if `url` provided, absolute local filesystem path otherwise. Optional, by default `.` is used.

name Node name as defined by the hierarchy in the metadata tree. Optional, by default the parent node `/` is used, which represents the whole metadata tree.

Here's a full fmf identifier example:

```
url: https://github.com/psss/fmf
ref: 0.10
path: /examples/wget
name: /download/test
```

Use default values for `ref` and `path` to reference the latest version of the smoke plan from the default branch:

```
url: https://github.com/psss/fmf
name: /plans/smoke
```

If desired, it is also possible to write the identifier on a single line as supported by the `yaml` format:

```
{url: "https://github.com/psss/fmf", name: "/plans/smoke"}
```

Let's freeze the stable test version by using a specific commit:

```
url: https://github.com/psss/fmf
ref: f24ef3f
name: /tests/basic/filter
```

Reference a smoke plan from another metadata tree stored on the local filesystem:

```
path: /home/psss/git/tmt
name: /plans/smoke
```

Local reference across the same metadata tree is also supported:

```
name: /plans/smoke
```

1.3 Features

Let's demonstrate the features on a simple wget example with the following directory structure:

```
wget
├── download
├── protocols
│   ├── ftp
│   ├── http
│   └── https
├── recursion
└── smoke
```

1.3.1 Simple

The most common use cases super simple to read & write. Test metadata for a single test look like this:

```
description: Check basic download options
tester: Petr Šplíchal <psplicha@redhat.com>
tags: [Tier2, TierSecurity]
test: runtest.sh
time: 3 min
```

1.3.2 Hierarchy

Hierarchy is defined by directory structure (see example above) and explicit nesting using attributes starting with `/`. Defining metadata for several tests in a single file is straightforward:

```
/download:
    description: Check basic download options
    tester: Petr Šplíchal <psplicha@redhat.com>
    tags: [Tier2, TierSecurity]
    test: runtest.sh
    time: 3 min
/recursion:
    description: Check recursive download options
    tester: Petr Šplíchal <psplicha@redhat.com>
    tags: [Tier2, TierSecurity]
    test: runtest.sh
    time: 20 min
```

Content above would be stored in `wget/main.fmf` file.

1.3.3 Inheritance

Metadata is inherited from parent objects:

```
tester: Petr Šplíchal <psplicha@redhat.com>
tags: [Tier2, TierSecurity]
test: runtest.sh

/download:
```

(continues on next page)

(continued from previous page)

```
    description: Check basic download options
    time: 3 min
/recursion:
    description: Check recursive download options
    time: 20 min
```

This nicely prevents unnecessary duplication. Redefining an attribute in a child object will by default overwrite value inherited from the parent.

1.3.4 Merging

When inheriting values from the parent it is also possible to use special attribute suffixes to merge child value with parent data. Append a + sign to the attribute name to add given value:

```
time: 1
/download:
    time+: 3
```

This operation is possible only for attributes of the same type. Exception `MergeError` is raised if types are different. When the + suffix is applied on dictionaries `update()` method is used to merge content of given dictionary instead of replacing it.

In a similar way, appending a − sign will reduce or remove parent value from parent's attribute (which has to be defined):

```
time-: 5
tags-: [Tier2]
desc-: details.*
vars-: [z]
```

Numbers are subtracked, list items are removed from the parent attribute, matching regular expressions are replaced by an empty string. For dictionaries it's possible to provide list of keys which should be removed.

1.3.5 Elasticity

Use a single file or scatter metadata across the hierarchy, whatever is more desired for the project.

File `wget/main.fmf`:

```
tester: Petr Šplíchal <psplicha@redhat.com>
tags: [Tier2, TierSecurity]
test: runtest.sh
```

File `wget/download/main.fmf`:

```
description: Check basic download options
time: 3 min
```

File: `wget/recursion/main.fmf`:

```
description: Check recursive download options
time: 20 min
```

This allows reasonable structure for both small and large projects.

1.3.6 Scatter

Thanks to elasticity, metadata can be scattered across several files. For example `wget/download` metadata can be defined in the following three files:

File `wget/main.fmf`:

```
/download:
    description: Check basic download options
    test: runtest.sh
```

File `wget/download.fmf`:

```
description: Check basic download options
test: runtest.sh
```

File `wget/download/main.fmf`:

```
description: Check basic download options
test: runtest.sh
```

Parsing is done from top to bottom (in the order of examples above). Later/lower defined attributes replace values defined earlier/higher in the structure.

1.3.7 Leaves

When searching, **key content** is used to define which leaves from the metadata tree will be selected. For example, every test case to be executed must have the `test` attribute defined, every requirement to be considered for test coverage evaluation must have the `requirement` attribute defined. Otherwise object data is used for inheritance only:

```
description: Check basic download options
test: runtest.sh
time: 3 min
```

The key content attributes are not supposed to be hard-coded in the Flexible Metadata Format but freely configurable. Multiple key content attributes (e.g. script & backend) could be used as well.

1.3.8 Virtual

Using a single test code for testing multiple scenarios can be easily implemented using leaves inheriting from the same parent:

```
description: Check basic download options
test: runtest.sh

/fast:
    description: Check basic download options (quick smoke test)
    environment: MODE=fast
    tags: [Tier1]
    time: 1 min
/full:
    description: Check basic download options (full test set)
    environment: MODE=full
```

(continues on next page)

(continued from previous page)

```
    tags: [Tier2]
    time: 3 min
```

In this way we can efficiently create virtual test cases.

1.3.9 Adjust

It is possible to adjust attribute values based on the current *Context*, for example disable test if it's not relevant for given environment:

```
enabled: true
adjust:
    enabled: false
    when: distro ~< fedora-33
    because: the feature was added in Fedora 33
```

Note that this functionality reserves the following attributes for its usage:

when The condition to be evaluated in order to decide if the metadata should be merged. This is a **required** key.

continue By default, all provided rules are evaluated. When set to `false`, the first successful rule finishes the evaluation and the rest is ignored.

because An optional comment with justification of the adjustment. Should be a plain string.

Name of the attribute which contains rules to be evaluated can be arbitrary. In the example the default key `adjust` is used.

1.3.10 Format

When investigating metadata using the `fmf` command line tool, object identifiers and all associated attributes are printed by default, each on a separate line. It is also possible to use the `--format` option together with `--value` options to generate custom output. Python syntax for expansion using `{}` is used to place values as desired. For example:

```
fmf --format 'name: {0}, tester: {1}\n' \
    --value 'name' --value 'data["tester"]'
```

Individual attribute values can be accessed through the `data` dictionary, variable `name` contains the object identifier and `root` is assigned to directory where metadata tree is rooted.

Python modules `os` and `os.path` as well as other python functions are available and can be used for processing attribute values as desired:

```
fmf --format '{}' --value 'os.dirname(data["path"])'
```

1.4 Context

1.4.1 Motivation

Imagine you have a test which can run only for Fedora 33 and newer. Or your tests' require depend on which distribution you are running. For these cases you need just a slight tweak to your metadata but you can't really use the *Virtual* cases as only one of them should run.

This is exactly where adjusting metadata based on the given Context will help you. Let's see some examples to demonstrate the usage on a real-life use case.

Disable test by setting the `enabled` attribute:

```
enabled: true
adjust:
    enabled: false
    when: distro < fedora-33
    because: The feature was added in Fedora-33
```

Tweak the `require` attribute for an older distro:

```
require:
  - procps-ng
adjust:
    require: procps
    when: distro ~= centos-6
```

1.4.2 Syntax

To get a better idea of the `when` condition syntax including supported operators consult the following grammar outline:

```
condition ::= expression (bool expression)*
bool ::= and | or
expression ::= dimension binary_operator values
expression ::= dimension unary_operator
dimension ::= [[:alnum:]]+
binary_operator ::= '==' | '!=' | '<' | '<=' | '>' | '>=' |
    '~=' | '~!=' | '~<' | '~<=' | '~>' | '~>='
unary_operator ::= 'is defined' | 'is not defined'
values ::= value (',' value)*
value ::= [[:alnum:]]+
```

Lazy Evaluation

Operator `and` takes precedence over `or` and rule evaluation is lazy. It stops immediately when we know the final result.

Boolean Operations

When a dimension or outcome of the operation is not defined, the expression is treated as `CannotDecide`.

Boolean operations with `CannotDecide`:

```
CannotDecide  and  True          ==  CannotDecide
CannotDecide  and  False         ==  False
CannotDecide  or   True          ==  True
CannotDecide  or   False         ==  CannotDecide
CannotDecide  and  CannotDecide  ==  CannotDecide
CannotDecide  or   CannotDecide  ==  CannotDecide
```

1.4.3 Dimensions

Each Dimension is a view on the Context in which metadata can be adjusted. For example it can be arch, distro, component, product or pipeline in which we run tests and so on.

Each value is treated as if it was a component with version. Name of the dimension doesn't matter, all are treated equally.

The characters `:` or `.` or `-` are used as version separators and are handled in the same way. The following examples demonstrate how the `name` and `version` parts are parsed:

```
centos-8.3.0
    name: centos
    version: 8, 3, 0

python3-3.8.5-5.fc32
    name: python3
    version: 3, 8, 5, 5, fc32

x86_64
    name: x86_64
    version: no version parts
```

1.4.4 Comparison

Value on the left always comes from dimension, it describes what is known about the context and should be as specific as possible (this is up to the calling tool). Value on the right comes from the rule and the creator of this rule sets how precise they want to be.

When the left side is not specific enough its missing version parts are treated as if they were lower than the right side. However, the left side needs to contain at least one version part:

```
git-2.3.4 < git-3    # True
git-2 < git-3.2.1    # True
git < git-3.2.1      # CannotDecide
```

Equality vs Comparison

It is always possible to evaluate whether two values are (not) equal. When the name and common version parts requested by the right side match then the two values are equal:

```
git-2.3.4 == git-2.3.4
git-2.3.4 == git-2.3
git-2.3.4 == git-2
git-2.3.4 == git
git-2.3.4 != git-1
git-2.3.4 != fmf
```

However, comparing order of two values is defined only if they match by name. If names don't match then values cannot be compared and the expression has `CannotDecide` outcome:

```
git-2.3.4 >= git-2      # True
git-2.3.4 >= git-3      # False
git-2.3.4 >= fmf-2      # CannotDecide
```

Major Version

Comparing distributions across their major versions can be tricky. One cannot easily say that e.g. `centos-8.0 > centos-7.9`. In this case `centos-8.0` was released sooner than `centos-7.9` so is it really newer?

Quite often new features are implemented in given minor version such as `centos-7.9` or `centos-8.2` which does not mean they are available in `centos-8.1` so it is not possible to apply a single rule such as `distro >= centos-7.9` to cover this case.

Another usage for this operators is to check for features specific to a particular major version or a module stream.

The following operators make it possible to compare only within the same major version:

```
'~=' | '~!=' | '~<' | '~<=' | '~>' | '~>='
```

If their major versions are different then their minor versions cannot be compared and as such are skipped during evaluation. The following example shows how the special less than operator ~< would be evaluated for given *centos* versions. Note that the right side defines if the minor comparison is evaluated or not.

~<	centos-7.9	centos-8.2	centos-8
centos-7.8	True	CannotDecide	True
centos-7.9	False	CannotDecide	True
centos-7	CannotDecide	CannotDecide	True
centos-8.1	CannotDecide	True	False
centos-8.2	CannotDecide	False	False
centos-8	CannotDecide	CannotDecide	False

Here is a couple of examples to get a better idea of how the comparison works for some special cases:

```
fedora < fedora-33 ---> cannot (left side has no version parts)
fedora-33 == fedora ---> True (right side wants only name)
fedora-33 < fedora-rawhide ---> True (rawhide is newer than any number)

centos-8.4.0 == centos ---> True
centos-8.4.0 < centos-9 ---> True
centos-8.4.0 ~< centos-9 ---> True (no minor comparison requested)
centos-8.4.0 ~< centos-9.2 ---> cannot (minor comparison requested)
```

1.5 Examples

Let's have a look at a couple of real-life examples!

1.5.1 Coverage

Test coverage information can be stored in a single file, for example `wget/requirements.fmf`:

```
/protocols:
    priority: high
    /ftp:
        requirement: Download a file using the ftp protocol.
        coverage: wget/protocols/ftp
    /http:
        requirement: Download a file using the http protocol.
```

(continues on next page)

(continued from previous page)

```
        coverage: wget/protocols/http
    /https:
        requirement: Download a file using the https protocol.
        coverage: wget/protocols/https

/download:
    priority: medium
    /output-document-pipe:
        requirement: Save content to pipe.
        coverage: wget/download
    /output-document-file:
        requirement: Save content to a file.
        coverage: wget/download

/upload:
    priority: medium
    /post-file:
        requirement: Upload a file to the server
        coverage: wget/protocols/http
    /post-data:
        requirement: Upload a string to the server
        coverage: wget/protocols/http
```

Or split by functionality area into separate files as desired, for example `wget/download/requirements.fmf`:

```
priority: medium
/output-document-pipe:
    requirement: Save content to pipe.
    coverage: wget/download
/output-document-file:
    requirement: Save content to a file.
    coverage: wget/download
```

Or integrated with test case metadata, e.g. `wget/download/main.fmf`:

```
description: Check basic download options
tags: [Tier2, TierSecurity]
test: runtest.sh
time: 3 min

/requirements:
    requirement: Various download options working correctly
    priority: low
    /get-file:
        coverage: wget/download
    /output-document:
        coverage: wget/download
    /continue:
    /timestamping:
    /tries:
    /no-clobber:
        coverage: wget/download
    /progress:
    /quota:
    /server-response:
    /bind-address:
    /spider:
```

In the example above three requirements are already covered, the rest still await for test coverage (attributes value is null).

1.5.2 Strategist

Here's an example implementation of test-strategist data for openscap using the Flexible Metadata Format:

```
/probes:
    description: Probes
    /offline:
        description: Offline scanning
    /online:
        description: Online scanning
/scanning:
    description: Reading and understanding source datastreams
    /oval:
        influencers:
        - openscap/probes/offline
        - openscap/probes/online
    /ds:
        influencers:
        - openscap/scanning/oval
        - openscap/scanning/cpe
    /cpe:
        influencers:
        - openscap/scanning/oval
```

1.5.3 Setups

This example shows how to use Flexible Metadata Format to run tests with different storage setups including cleanup. This is simplified metadata, whole example including tools can be found at storage_setup:

```
/setups:
    description: Tests to prepare and clean up devices for tests
    setup: True
    /setup_local:
        test: setup_local.py
        requires_cleanup: setups/cleanup_local
    /cleanup_local:
        test: cleanup_local.py
    /setup_remote:
        test: setup_remote.py
        requires_cleanup: setups/cleanup_remote
    /cleanup_remote:
        test: cleanup_remote.py
    /setup_vdo:
        test: setup_vdo.py
        requires_cleanup: setups/cleanup_vdo
    /cleanup_vdo:
        test: cleanup_vdo.py
/tests:
    description: Testing 'vdo' command line tool
    requires_setup: [setups/setup_vdo]
    /create
        description: Testing 'vdo create'
```

(continues on next page)

(continued from previous page)

```
        /ack_threads
        /activate
    /modify
        description: Testing 'vdo modify'
        requires_setup+: [setups/setup_remote]
        /block_map_cache_size
```

You can find here not only how to use FMF for setup/cleanup and group tests based on that, but also installing requirements, passing values from metadata to tests themself and much more.

1.5.4 Format

Custom format output using `--format` and `value`.

List object name and selected attribute:

```
fmf examples/wget --format '{0}: {1}\n' \
    --value 'name' --value 'data["tester"]'
```

Show missing attributes in red:

```
fmf examples/wget/ --format '{}: {}\n' --value 'name' \
    --value 'utils.color(str(data.get("priority")),
    "red" if data.get("priority") is None else "green")'
```

List all test scripts with full path:

```
fmf examples --key test --format "{}/{}/{}\n" \
    --value "os.getcwd()" \
    --value "data.get('path') or name" \
    --value "data['test']"
```

1.6 Modules

Flexible Metadata Format

class fmf.**Tree**(*data*, *name=None*, *parent=None*)

Metadata Tree

adjust(*context*, *key='adjust'*, *undecided='skip'*)

Adjust tree data based on provided context and rules

The 'context' should be an instance of the fmf.context.Context class describing the environment context. By default, the key 'adjust' of each node is inspected for possible rules that should be applied. Provide 'key' to use a custom key instead.

Optional 'undecided' parameter can be used to specify what should happen when a rule condition cannot be decided because context dimension is not defined. By default, such rules are skipped. In order to raise the fmf.context.CannotDecide exception in such cases use undecided='raise'.

child(*name*, *data*, *source=None*)

Create or update child with given data

climb(*whole=False*)

Climb through the tree (iterate leaf/all nodes)

commit
Commit hash if tree grows under a git repo, False otherwise

Return current commit hash if the metadata tree root is located under a git repository. For metadata initialized from a dict or local directory with no git repo 'False' is returned instead.

copy()
Create and return a deep copy of the node and its subtree

It is possible to call copy() on any node in the tree, not only on the tree root node. Note that in that case, parent node and the rest of the tree attached to it is not copied in order to save memory.

find(*name*)
Find node with given name

get(*name=None*, *default=None*)
Get attribute value or return default

Whole data dictionary is returned when no attribute provided. Supports direct values retrieval from deep dictionaries as well. Dictionary path should be provided as list. The following two examples are equal:

tree.data['hardware']['memory']['size'] tree.get(['hardware', 'memory', 'size'])

However the latter approach will also correctly handle providing default value when any of the dictionary keys does not exist.

grow(*path*)
Grow the metadata tree for the given directory path

Note: For each path, grow() should be run only once. Growing the tree from the same path multiple times with attribute adding using the "+" sign leads to adding the value more than once!

inherit()
Apply inheritance

static init(*path*)
Create metadata tree root under given path

merge(*parent=None*)
Merge parent data

static node(*reference*)
Return Tree node referenced by the fmf identifier

Keys supported in the reference:

url .... git repository url (optional) ref .... branch, tag or commit (default branch if not provided) path ... metadata tree root ('.' by default) name ... tree node name ('/' by default)

See the documentation for the full fmf id specification: https://fmf.readthedocs.io/en/latest/concept.html#identifiers Raises ReferenceError if referenced node does not exist.

prune(*whole=False*, *keys=None*, *names=None*, *filters=None*, *conditions=None*)
Filter tree nodes based on given criteria

show(*brief=False*, *formatting=None*, *values=None*)
Show metadata

update(*data*)
Update metadata, handle virtual hierarchy

fmf.**filter**(*filter*, *data*, *sensitive=True*, *regexp=False*)
Return true if provided filter matches given dictionary of values

Filter supports disjunctive normal form with '|' used for OR, '&' for AND and '-' for negation. Individual values are prefixed with 'value:', leading/trailing white-space is stripped. For example:

```
tag: Tier1 | tag: Tier2 | tag: Tier3
category: Sanity, Security & tag: -destructive
```

Note that multiple comma-separated values can be used as a syntactic sugar to shorten the filter notation:

```
tag: A, B, C ---> tag: A | tag: B | tag: C
```

Values should be provided as a dictionary of lists each describing the values against which the filter is to be matched. For example:

```
data = {tag: ["Tier1", "TIPpass"], category: ["Sanity"]}
```

Other types of dictionary values are converted into a string. A FilterError exception is raised when a dimension parsed from the filter is not found in the data dictionary. Set option 'sensitive' to False to enable case-insensitive matching. If 'regexp' option is True, regular expressions can be used in the filter values as well.

1.6.1 base

Base Metadata Classes

class fmf.base.**Tree**(*data*, *name=None*, *parent=None*)
Metadata Tree

adjust(*context*, *key='adjust'*, *undecided='skip'*)
Adjust tree data based on provided context and rules

The 'context' should be an instance of the fmf.context.Context class describing the environment context. By default, the key 'adjust' of each node is inspected for possible rules that should be applied. Provide 'key' to use a custom key instead.

Optional 'undecided' parameter can be used to specify what should happen when a rule condition cannot be decided because context dimension is not defined. By default, such rules are skipped. In order to raise the fmf.context.CannotDecide exception in such cases use undecided='raise'.

child(*name*, *data*, *source=None*)
Create or update child with given data

climb(*whole=False*)
Climb through the tree (iterate leaf/all nodes)

commit
Commit hash if tree grows under a git repo, False otherwise

Return current commit hash if the metadata tree root is located under a git repository. For metadata initialized from a dict or local directory with no git repo 'False' is returned instead.

copy()
Create and return a deep copy of the node and its subtree

It is possible to call copy() on any node in the tree, not only on the tree root node. Note that in that case, parent node and the rest of the tree attached to it is not copied in order to save memory.

find(*name*)
Find node with given name

get(*name=None*, *default=None*)
Get attribute value or return default

Whole data dictionary is returned when no attribute provided. Supports direct values retrieval from deep dictionaries as well. Dictionary path should be provided as list. The following two examples are equal:

tree.data['hardware']['memory']['size'] tree.get(['hardware', 'memory', 'size'])

However the latter approach will also correctly handle providing default value when any of the dictionary keys does not exist.

grow(*path*)
Grow the metadata tree for the given directory path

Note: For each path, grow() should be run only once. Growing the tree from the same path multiple times with attribute adding using the "+" sign leads to adding the value more than once!

inherit()
Apply inheritance

static init(*path*)
Create metadata tree root under given path

merge(*parent=None*)
Merge parent data

static node(*reference*)
Return Tree node referenced by the fmf identifier

Keys supported in the reference:

url .... git repository url (optional) ref .... branch, tag or commit (default branch if not provided) path ... metadata tree root ('.' by default) name ... tree node name ('/' by default)

See the documentation for the full fmf id specification: https://fmf.readthedocs.io/en/latest/concept.html#identifiers Raises ReferenceError if referenced node does not exist.

prune(*whole=False*, *keys=None*, *names=None*, *filters=None*, *conditions=None*)
Filter tree nodes based on given criteria

show(*brief=False*, *formatting=None*, *values=None*)
Show metadata

update(*data*)
Update metadata, handle virtual hierarchy

fmf.base.**construct_yaml_str**(*self*, *node*)

fmf.base.**unique_key_constructor**(*loader*, *node*, *deep=False*)
YAML constructor that checks for duplicate keys

1.6.2 utils

Logging, config, constants & utilities

class fmf.utils.**Coloring**(*mode=None*)
Coloring configuration

MODES = ['COLOR_OFF', 'COLOR_ON', 'COLOR_AUTO']

enabled()
True if coloring is currently enabled

get()
Get the current color mode

set(*mode=None*)
Set the coloring mode

If enabled, some objects (like case run Status) are printed in color to easily spot failures, errors and so on. By default the feature is enabled when script is attached to a terminal. Possible values are:

```
COLOR=0 ... COLOR_OFF .... coloring disabled
COLOR=1 ... COLOR_ON ..... coloring enabled
COLOR=2 ... COLOR_AUTO ... if terminal attached (default)
```

Environment variable COLOR can be used to set up the coloring to the desired mode without modifying code.

exception fmf.utils.**FetchError**
Fatal error in helper command while fetching

exception fmf.utils.**FileError**
File reading error

exception fmf.utils.**FilterError**
Missing data when filtering

exception fmf.utils.**FormatError**
Metadata format error

exception fmf.utils.**GeneralError**
General error

class fmf.utils.**Logging**(*name='fmf'*)
Logging Configuration

COLORS = {4: 'magenta', 7: 'cyan', 10: 'green', 20: 'blue', 30: 'yellow', 40: 'r

class ColoredFormatter(*fmt=None*, *datefmt=None*, *style='%'*)
Custom color formatter for logging

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

LEVELS = ['CRITICAL', 'DEBUG', 'ERROR', 'FATAL', 'INFO', 'NOTSET', 'WARN', 'WARNING']

MAPPING = {0: 30, 1: 20, 2: 10, 3: 7, 4: 4, 5: 1}

get()
Get the current log level

set(*level=None*)
Set the default log level

If the level is not specified environment variable DEBUG is used with the following meaning:

```
DEBUG=0 ... LOG_WARN (default)
DEBUG=1 ... LOG_INFO
DEBUG=2 ... LOG_DEBUG
DEBUG=3 ... LOG_CACHE
```

(continues on next page)

(continued from previous page)

```
DEBUG=4 ... LOG_DATA
DEBUG=5 ... LOG_ALL (log all messages)
```

exception fmf.utils.**MergeError**
Unable to merge data between parent and child

exception fmf.utils.**ReferenceError**
Referenced tree node cannot be found

exception fmf.utils.**RootError**
Metadata tree root missing

fmf.utils.**clean_cache_directory**()
Delete used cache directory if it exists

fmf.utils.**color**(*text*, *color=None*, *background=None*, *light=False*, *enabled='auto'*)
Return text in desired color if coloring enabled

Available colors: black red green yellow blue magenta cyan white. Alternatively color can be prefixed with "light", e.g. lightgreen.

fmf.utils.**dict_to_yaml**(*data*, *width=None*, *sort=False*)
Convert dictionary into yaml

fmf.utils.**evaluate**(*expression*, *data*, *_node=None*)
Evaluate arbitrary Python expression against given data

Expects data dictionary which will be used to populate local namespace. Used to provide flexible conditions for filtering.

fmf.utils.**fetch**(*url*, *ref=None*, *destination=None*, *env=None*)
Deprecated: Use *fetch_repo()* instead

fmf.utils.**fetch_repo**(*url*, *ref=None*, *destination=None*, *env=None*)
Fetch remote git repository and return local directory

Fetch git repository from provided url into a local cache directory, checkout requested ref and return path to the repo. If no ref is provided, the default branch from the origin is used. If destination directory is provided, it should not exist or needs to be empty. Use dictionary env to set environment variables for git calls.

Raises FetchError upon failure with the original exception included.

fmf.utils.**fetch_tree**(*url*, *ref=None*, *path='.'*)
Get initialized Tree from a remote git repository

url .... git repository url (required) ref .... branch, tag or commit (default branch if None) path ... metadata tree root (default to '.')

See *fmf.base.Tree.node()* to canonical default values.

Remote repository is cached locally (see *get_cache_directory()*), local directory with cache is locked during reading.

Raises GeneralError when lock couldn't be acquired.

fmf.utils.**filter**(*filter*, *data*, *sensitive=True*, *regexp=False*)
Return true if provided filter matches given dictionary of values

Filter supports disjunctive normal form with '|' used for OR, '&' for AND and '-' for negation. Individual values are prefixed with 'value:', leading/trailing white-space is stripped. For example:

```
tag: Tier1 | tag: Tier2 | tag: Tier3
category: Sanity, Security & tag: -destructive
```

Note that multiple comma-separated values can be used as a syntactic sugar to shorten the filter notation:

```
tag: A, B, C ---> tag: A | tag: B | tag: C
```

Values should be provided as a dictionary of lists each describing the values against which the filter is to be matched. For example:

```
data = {tag: ["Tier1", "TIPpass"], category: ["Sanity"]}
```

Other types of dictionary values are converted into a string. A FilterError exception is raised when a dimension parsed from the filter is not found in the data dictionary. Set option 'sensitive' to False to enable case-insensitive matching. If 'regexp' option is True, regular expressions can be used in the filter values as well.

fmf.utils.**get_cache_directory**(*create=True*)
Return cache directory, created by this call if necessary

Cache directory is (first existing): - Value of FMF_CACHE_DIRECTORY environment variable - Value set by the last call of *set_cache_directory()* - $XDG_CACHE_HOME/fmf - ~/.cache/fmf

Raise GeneralError if it is not possible to create it.

fmf.utils.**info**(*message*, *newline=True*)
Log provided info message to the standard error output

fmf.utils.**invalidate_cache**()
Force fetch next time cache is used regardless its age

fmf.utils.**listed**(*items*, *singular=None*, *plural=None*, *max=None*, *quote=''*, *join='and'*)
Convert an iterable into a nice, human readable list or description:

```
listed(range(1)) ..................... 0
listed(range(2)) ..................... 0 and 1
listed(range(3), join='or') ......... 0, 1 or 2
listed(range(3), quote='"') ......... "0", "1" and "2"
listed(range(4), max=3) ............. 0, 1, 2 and 1 more
listed(range(5), 'number', max=3) ... 0, 1, 2 and 2 more numbers
listed(range(6), 'category') ........ 6 categories
listed(7, "leaf", "leaves") ......... 7 leaves
```

If singular form is provided but max not set the description-only mode is activated as shown in the last two examples. Also, an int can be used in this case to get a simple inflection functionality.

fmf.utils.**pluralize**(*singular=None*)
Naively pluralize words

fmf.utils.**repr_str**(*dumper*, *data*)

fmf.utils.**run**(*command*, *cwd=None*, *check_exit_code=True*, *env=None*)
Run command and return a (stdout, stderr) tuple

:command as list (name, arg1, arg2...) :cwd path to directory where to run the command :check_exit_code raise CalledProcessError if exit code is non-zero :env dictionary of the environment variables for the command

fmf.utils.**set_cache_directory**(*cache_directory*)
Set preferred cache directory

fmf.utils.**set_cache_expiration**(*seconds*)
Seconds until cache expires

fmf.utils.**split**(*values*, *separator=re.compile('[,]+')*)
Convert space-or-comma-separated values into a single list

Common use case for this is merging content of options with multiple values allowed into a single list of strings thus allowing any of the formats below and converts them into ['a', 'b', 'c']:

```
--option a --option b --option c ... ['a', 'b', 'c']
--option a,b --option c ............ ['a,b', 'c']
--option 'a b c' ................... ['a b c']
```

Accepts both string and list. By default space and comma are used as value separators. Use any regular expression for custom separator.

1.6.3 cli

This is command line interface for the Flexible Metadata Format.

Available commands are:

```
fmf ls      List identifiers of available objects
fmf show    Show metadata of available objects
fmf init    Initialize a new metadata tree
fmf clean   Remove cache directory and its content
```

See online documentation for more details and examples:

http://fmf.readthedocs.io/

Check also help message of individual commands for the full list of available options.

class fmf.cli.**Parser**(*arguments=None*, *path=None*)
Command line options parser

clean()
Remove cache directory

command_clean()
Clean cache

command_init()
Initialize tree

command_ls()
List names

command_show()
Show metadata

options_formatting()
Formating options

options_select()
Select by name, filter

options_utils()
Utilities

show(*brief=False*)
Show metadata for each path given

fmf.cli.**main**(*arguments=None*, *path=None*)
Parse options, do what is requested

1.7 Contribute

1.7.1 Introduction

Feel free and welcome to contribute to this project. You can start with filing issues and ideas for improvement in GitHub tracker. Our favorite thoughts from The Zen of Python:

- Beautiful is better than ugly.
- Simple is better than complex.
- Readability counts.

We respect the PEP8 Style Guide for Python Code. Here's a couple of recommendations to keep on mind when writing code:

- Comments should be complete sentences.
- The first word should be capitalized (unless identifier).
- When using hanging indent, the first line should be empty.
- The closing brace/bracket/parenthesis on multiline constructs is under the first non-whitespace character of the last line

1.7.2 Commits

It is challenging to be both concise and descriptive, but that is what a well-written summary should do. Consider the commit message as something that will be pasted into release notes:

- The first line should have up to 50 characters.
- Complete sentence with the first word capitalized.
- Should concisely describe the purpose of the patch.
- Do not prefix the message with file or module names.
- Other details should be separated by a blank line.

Why should I care?

- It helps others (and yourself) find relevant commits quickly.
- The summary line will be re-used later (e.g. for rpm changelog).
- Some tools do not handle wrapping, so it is then hard to read.
- You will make the maintainers happy to read beautiful commits :)

You can get some more context in the stackoverflow article.

1.7.3 Develop

In order to experiment, play with the latest bits and develop improvements it is best to use a virtual environment:

```
mkvirtualenv fmf
git clone https://github.com/psss/fmf
cd fmf
pip install -e .
```

Install `python3-virtualenvwrapper` to easily create and enable virtual environments using `mkvirtualenv` and `workon`. Note that if you have freshly installed the package you need to open a new shell session to enable the wrapper functions.

Install the `pre-commit` hooks to run all available checks for your commits to the project:

```
pip install pre-commit
pre-commit install
```

1.7.4 Makefile

There are several Makefile targets defined to make the common daily tasks easy & efficient:

make test Execute the test suite.

make smoke Perform quick basic functionality test.

make coverage Run the test suite under coverage and report results.

make docs Build documentation.

make packages Build rpm and srpm packages.

make tags Create or update the Vim `tags` file for quick searching. You might want to use `set tags=./tags;` in your `.vimrc` to enable parent directory search for the tags file as well.

make clean Cleanup all temporary files.

1.7.5 Tests

Run the default set of tests directly on your localhost:

```
tmt run
```

To run tests using pytest with the test coverage overview:

```
make coverage
```

Install pytest and coverage using dnf or pip:

```
dnf install python3-pytest python3-coverage
pip install pytest coveralls
```

1.7.6 Docs

For building documentation locally install necessary modules:

```
pip install sphinx sphinx_rtd_theme mock
```

Make sure docutils are installed in order to build man pages:

```
dnf install python3-docutils
```

Building documentation is then quite straightforward:

```
make docs
```

Find the resulting html pages under the `docs/_build/html` folder.

CHAPTER 2

Indices and Tables

- genindex
- modindex

Python Module Index

f

Index

N

O

P

R

S

T

U