cfme_tests Documentation

Release 0

RedHat QE

March 07, 2016

Contents

Contents:

CHAPTER 1

Getting Started

1.1 Setup

- Create a virtualenv from which to run tests
  - Execute one of the following commands:
    * `pip install virtualenv`
    * `easy_install virtualenv`
    * `yum install python-virtualenv`
  - Create a virtualenv: `virtualenv <name>`
  - To activate the virtualenv later: `source <name>/bin/activate`
- Fork and Clone this repository into the new virtualenv
- Set the `PYTHONPATH` to include `cfme_tests`. Edit your virtualenv's `bin/activate` script, created with the virtualenv. At the end of the file, export a PYTHONPATH variable with the path to the repository clone by adding this line (altered to match your repository locations):
  - `export PYTHONPATH='/path/to/virtualenv/cfme_tests'`
- Also add this line at the end of your virtualenv to prevent .pyc files polluting your folders:
  - `export PYTHONDONTWRITEBYTECODE="yes"`
- Make sure you set the shared secret for the credentials files encryption. There are two ways:
  - add `export CFME_TESTS_KEY="our shared key"` into the activate script
  - create `.yaml_key` file in project root containing the key
- Ensure the following devel packages are installed (for building python dependencies):
  - `gcc`
  - `postgresql-devel`
  - `libxml2-devel`
  - `libxslt-devel`
  - `zeromq3-devel`
  - yum users: `sudo yum install gcc postgresql-devel libxml2-devel libxslt-devel zeromq3-devel`

- Install python dependencies:
  - `pip install -Ur /path/to/virtualenv/cfme_tests/requirements.txt`
- Copy template files in cfme_tests to the same file name without the `.template` extension
  - Example: `cp file.name.template file.name`
  - Bash script example: `for file in *.template; do cp -n $file ${file/.template}; done`
  - Edit these files as needed to reflect your environment.
- Do the same for the config yamls in the conf directory, using Configuration YAMLs
  - Example: `cd conf/; cp env.local.template env.local`
  - Then edit `conf/env.local.yaml` to override `base_url`
- Set up a local selenium server that opens browser windows somewhere other than your desktop by running *Selenium over VNC*
- Test! Run py.test. (This takes a long time, Ctrl-C will stop it)

Note: In the past, the pytest_mozwebqa package was used to help manage the web browser and selenium session. We've recently done away with it, so you can safely `pip uninstall pytest_mozwebqa`. pytest_mozwebqa provided many commandline options (for example: `--driver`, `--baseurl`, `--credentials`, `--untrusted`). These will all need to be removed from the py.test invocation (or addopts line in pytest.ini) if mozwebqa is uninstalled.

1.2 Activating the virtualenv

The virtualenv is activated on creation. To reactivate the virtualenv in subsequent sessions, the `bin/activate` script must be sourced.

```
#Bash example:
`cd /path/to/virtualenv'
source bin/activate or . bin/activate
```

1.3 Testing Framework

The testing framework being used is py.test

1.4 Browser Support

We support any browser that selenium supports, but tend to run Firefox or Chrome.

For detailed instructions on setting up different browsers, see *Browser Configuration*.

CHAPTER 2

Guides

2.1 Abbreviations and Naming Conventions

2.1.1 Abbreviations

In order to save line space and aid in quick pass reading, we have defined some abbreviations which we propose to be used throughout the code base.

Common Terms

Abbreviation	Meaning
cfg, config	Configuration
prov	Provider
pg	Page
db	Database
img	Image
vm	Virtual Machine
creds	Credentials

Locator Terms

Abbreviation	Meaning
btn	button
sel	select
txt	text
pwd	password
chk	checkbox
tarea	textarea

2.2 Browser Configuration

All browser configuration is done by editing `conf/env.yaml`, or creating a local override in `conf/env.local.yaml`. Local overrides are preferred. For more information about configuration yamls, see `utils.conf`.

All yaml examples in this document are snippets from `env.yaml`.

2.2.1 Local vs. Remote

Most WebDrivers can operate in two modes, as a local WebDriver or through a Remote WebDriver. The local WebDriver will launch a browser in the calling environment (such as your desktop), while the Remote WebDriver will connect to a remote selenium server (hence the name) and attempt to run the browser there.

Examples for each mode will be provided, where appropriate. Note that capitalization is extremely important when specifying either `webdriver` or `browserName`, as indicated in the examples below.

Some help for setting up the remote selenium server can be found in the *Selenium over VNC* document.

2.2.2 WebDriver Wharf

A variant of the Remote webdriver, WebDriver Wharf will spawn docker containers running the selenium standalone server on request.

Remote desired_capabilities

All `Remote` drivers take a "desired_capabilities" dictionary. Details on what keys and expected value types can be used in this dictionary can be found in the selenium documentation:

> https://code.google.com/p/selenium/wiki/DesiredCapabilities

Selenium, by default, looks for the selenium server on localhost port 4444. If the selenium server is running on a different machine, you'll need to add a `command_executor` option to `webdriver_options` in the examples below to the machine running the selenium server.

`command_exector` must be a URL to a selenium server hub, which by default is at the `/wd/hub` path on the selenium server.

For example:

```
browser:
     webdriver: Remote
     webdriver_options:
         command_executor: http://selenium-server-hostname:port/wd/hub
         desired_capabilities:
             browserName: browser
```

Note:

- Each browser has its own set of capabilities, and those capabilities will usually not apply from one browser to another.
- While most selenium identifiers have been translated from `JavaIdentifiers` to `python_identifiers`, the keys of `desired_capabilities` are not altered in any way. No name translation should have to be done for `desired_capabilities` keys (e.g. `browserName` does not become `browser_name`).

2.2.3 base_url

Regardless of which Webdriver you use, `base_url` must be set. It is assumed that the website at the `base_url` will be a working CFME UI.

Note: `base_url` is not solely used by the browser. Other functionality, such as the SSH and SOAP clients, derive their destination addresses from the `base_url`.

2.2.4 Firefox

Firefox has built-in support for selenium (and vice-versa). No additional configuration should be required to use the Firefox browser.

Local

```
browser:
    webdriver: Firefox
```

Remote

```
browser:
    webdriver: Remote
    webdriver_options:
        desired_capabilities:
            browserName: firefox
```

WebDriver Wharf

```
browser:
    webdriver: Remote
    webdriver_options:
        desired_capabilities:
            browserName: firefox
    webdriver_wharf: http://wharf.host:4899/
```

2.2.5 Chrome

In order to use Chrome with selenium, you must first install the `chromedriver` executable. This executable should be somewhere on your `PATH`.

- Download chromedriver. Use the latest available release for your architecture.
- `chromedriver` documentation: https://sites.google.com/a/chromium.org/chromedriver/getting-started

Local

```
browser:
    webdriver: Chrome
```

Remote

```
browser:
   webdriver: Remote
   webdriver_options:
       desired_capabilities:
           browserName: chrome
```

WebDriver Wharf

```
browser:
    webdriver: Remote
    webdriver_options:
        desired_capabilities:
            browserName: chrome
    webdriver_wharf: http://wharf.host:4899/
```

2.2.6 Safari

Like Firefox, Safari is natively supported by selenium. Usage is equally simple, with the exception that you'll probably need to be running selenium on OS X.

Local

```
browser:
    webdriver: Safari
```

Remote

```
browser:
    webdriver: Remote
    webdriver_options:
        # If selenium is running remotely, remember to update command_executor
        #command_executor: http://safari_host/wd/hub
        desired_capabilities:
            browserName: safari
```

2.2.7 Internet Explorer

Like Chrome & `chromedriver`, Internet Explorer needs a separate executable to work with selenium, `InternetExplorerDriver`. `InternetExplorerDriver` is a server that only runs in Windows, and should be running before starting selenium in either Local or Remote mode.

- For more information, visit https://code.google.com/p/selenium/wiki/InternetExplorerDriver

Local

```
browser:
    webdriver: Ie
```

Remote

```
browser:
    webdriver: Remote
    webdriver_options:
        # If selenium is running remotely, remember to update command_executor
        #command_executor: http://windows_host/wd/hub
        desired_capabilities:
```

```
        browserName: internet explorer
        # platform must be WINDOWS for IE
        platform: WINDOWS
```

2.2.8 Sauce Labs

By providing selenium servers on a multitude of platforms, Sauce Labs is able to help us test in "exotic" environments. In order to test against appliances behind firewalls, sauce-connect must be used:

https://saucelabs.com/docs/connect

sauce-connect tunnels are used by default if they're running, so the same `command_executor` can be used to use the sauce labs service whether sauce-connect is running or not:

```
command_executor: http://username:apikey@ondemand.saucelabs.com:80/wd/hub
```

Internet Explorer Sauce

The following example is our "worst-case scenario", which is running a very recent release of Internet Explorer in a very recent release of Windows:

```
browser:
    webdriver: Remote
    webdriver_options:
        command_executor: http://username:apikey@ondemand.saucelabs.com:80/wd/hub
        desired_capabilities:
            browserName: internet explorer
            platform: Windows 8.1
            version: 11
            screen-resolution: 1280x1024
```

The above configuration, at the time of this writing, ran our test suite with no issues.

More information on sauce-specific options allowed in desired_capabilities can be found in the sauce labs documentation:

- https://saucelabs.com/platforms
- https://saucelabs.com/docs/additional-config#desired-capabilities

---

Note: Python values for the browser constants used in the sauce labs "platform" page can be found here: https://code.google.com/p/selenium/source/browse/py/selenium/webdriver/common/desired_capabilities.py

---

2.2.9 Troubleshooting

If errors are encountered while launching a selenium browser, check the selenium website to make sure that your version of selenium matches the latest version. If not, upgrade.

https://code.google.com/p/selenium/downloads/list

2.3 Style Guide

2.3.1 General Guidelines

Contributing

- Own your pull requests; **you** are their advocate.
  - If a request goes unreviewed for two or three days, ping a reviewer to see what's holding things up.
  - Follow up on open pull requests and respond to any comments or questions a reviewer might have.
- Keep the contents of the pull request focused on one idea. Smaller pull requests are easier to review, and thus will be merged in more quickly.
- After submitting a request, be ready to work closely with a reviewer to get it tested and integrated into the overall test suite.
- Follow the Code Style guidelines to make your pull request as easy to review as possible.
- If your request requires the use of private information that can't be represented in the data file templates (probably cfme_data.yaml), please state that in the test module docstring or the individual test docstring, along with information on where that data can be found.
- Similar to the last point, any data files used by a test module should be clearly documented in that module's docstring.
- Any data required in a sensitive data file should be reflected in the template for that file.
- Standards may change over time, so copying older code with similar functionality may not be the most productive action. If in doubt, refer back to this document and update the copied code according to the current guidelines.
- Please keep large lint changes separate from new features, though this point should become less relevant over time.
- All pull requests should be squashed down to logical blocks of distinctive functionality that work by themselves and do not result in brokenness of master
  - As an example, if you were working on a test which required new pages, utilities and tests, it would be OK to split the page, utility and test changes into separate requests or commits, providing they were in the correct order of dependency.

Reviewers

Reviewers will be looking to make sure that the Contributing guidelines are being met. Some of the things that go into the review process:

- Assign the PR to the reviewer
- Pull request branches will be rebased against current master before testing.
- Newly added tests will be run against a clean appliance.
- Adherence to code style guidelines will be checked.

If tests fail, reviewers *WILL*:

- ...give you a complete traceback of the error.
- ...give you useful information about the appliance against which tests were run, such as the appliance version.

- ...give you insight into any related data files used.

If tests fail, reviewers *WILL NOT*:

- ...thoroughly debug the failing test(s).

All requests require 2 approvals from two reviewers, after which time, the contributor may, permissions allowing, merge the commit him/herself.

Reviewers must never approve their own pull requests.

Code Style

We adhere to Python's PEP 8 style guide , occasionally allowing exceptions for the sake of readability. This is covered in the Foolish Consistency section of PEP 8. Information on using linting tools to help with this can be found on the *flake8* page.

We also do a few things that aren't explicitly called out in PEP 8:

- The github pull request pane is our primary code review medium, and has a minimum width of 100 characters. As a result, our maximum line length is 100 characters, rather than 80.
- When wrapping blocks of long lines, indent the trailing lines once, instead of indenting to the opening bracket. This helps when there are large blocks of long lines, to preserve some readability:

```
_really_really_long_locator_name = (True, 'div > tr > td > a[title="this \
    is just a little too long"]')
_another_really_super_long_locator_name = (True, 'div > tr > td > \
    a[title="this is getting silly now"]')
```

- When wrapping long conditionals, indent trailing lines twice, just like with function names and any other block statement (they usually end with colons):

```
if this_extremely_long_variable_name_takes_up_the_whole_line and \
        you_need_to_wrap_your_conditional_to_the_next_line:
    # Two indents help clearly separate the wrapped conditional
    # from the following code.
```

- When indenting a wrapping sequence, one indent will do. Don't try to align all of the sequence items at an arbitrary column:

```
a_good_list = [
    'item1',
    'item2',
    'item3'
]

a_less_good_list = [ 'item1',
                     'item2',
                     'item3'
]
```

- According to PEP 8, triple-quoted docstrings use double quotes. To help differentiate docstrings from normal multi-line strings, consider using single-quotes in the latter case:

```
"""This is a docstring.

It follows PEP 8's docstring guidelines.

"""
```

```
paragraph = '''This is a triple-quoted string, with newlines captured.
PEP 8 and PEP 257 guidelines don't apply to this. Using single quotes here
makes it simple for a reviewer to know that docstring style doesn't apply
to this text block.'''
```

- On the subject of docstrings (as well as comments) +++use them+++. Python is somewhat self-documenting, so use docstrings and comments as a way to explain not just what code is doing, but why it's doing what it is, and what it's intended to achieve.

  We have decided to use the following docstring format and use the Cartouche Sphinx plugin to generate nice docs. Details on the format can be found above, but an example is described below:

```
def my_function(self, locator):
    """Runs the super cool function on a locator

    Seriously, you have to try this

    Note: You don't actually have to try it

    Args:
        locator: The name of a locator that can be described by using
            multiple lines.

    Returns:
        Nothing at all.

    Raises:
        CertainQuestionsError: Raises certain questions about the authors sanity.
    """
```

- In addition to being broken up into the three sections of standard library, third-party, and the local application, imports should be sorted alphabetically. 'import' lines within those sections still come before 'from ... import' lines:

```
import sys
from os import environ
from random import choice
```

General Notes

- Avoid using `time.sleep()` as much as possible to workaround quirks in the UI. There is a `utils.wait.wait_for()` utility that can be used to wait for arbitrary conditions. In most cases there is some DOM visible change on the page which can be waited for.
- Avoid using `time.sleep()` for waiting for changes to happen outside of the UI. Consider using tools like mgmt_system to probe the external systems for conditions for example and tie it in with a `utils.wait.wait_for()` as discussed above.
- If you feel icky about something you've written but don't know how to make it better, ask someone. It's better to have it fixed before submitting it as a pull request ;)

Other useful code style guidelines:

- PEP 20 - The Zen of Python
- PEP 257 - Docstring Conventions

2.3.2 cfme_tests

For page development, please refer to *Page Development*.

Layout

`cfme_tests/`

- `cfme/` Page modeling and tests
  - `web_ui/` The new web framework
  - `fixtures/` The new fixtures
  - `tests/` Tests container
- `conf/` Place for configuration files
- `data/` Test data. The structure of this directory should match the structure under `cfme/tests/`, with data files for tests in the same relative location as the test itself.
  - For example, data files for `cfme/tests/dashboard/test_widgets.py` could go into `data/dashboard/test_widgets/`.
- `fixtures/` py.test fixtures that can be used by any test. Modules in this directory will be auto loaded.
- `markers/` py.test markers that can be used by any test. Modules in this directory will be auto loaded.
- `metaplugins/` Plugins loaded by `@pytest.mark.meta`. Further informations in `markers.meta`
- `utils/` Utility functions that can be called inside our outside the test context. Generally, util functions benefit from having a related test fixture that exposes the utility to the tests. Modules in this directory will be auto loaded.
  - `tests/` Unit tests for utils
- `scripts/` Useful scripts for QE developers that aren't used during a test run
- `sprout/` Here lives the Sprout appliance tool.

Writing Tests

Tests in *cfme_tests* have the following properties:

- They pass on a freshly deployed appliance with no configuration beyond the defaults (i.e. tests do their own setup and teardown).
- Where possible, they strive to be idempotent to facilitate repeated testing and debugging of failing tests. (Repeatable is Reportable)
- Where possible, they try to clean up behind themselves. This not only helps with idempotency, but testing all of the CRUD interactions helps to make a thorough test.
- Tests should be thoroughly distrustful of the appliance, and measure an action's success in as many ways as possible. A practical example:
  - Do not trust flash messages, as they sometimes tell lies (or at least appear to). If you can go beyond a flash message to verify a test action, do so.

Some points when writing tests:

- When naming a test, do not use a common part of multiple test names as a test name itself. In the example below, trying to run a single test called `test_provider_add`, not only runs that test, but also `test_provider_add_new` and `test_provider_add_delete`, as pytest uses string matching for test names. `test_provider_add` should have a suffix making it unique. In this way a tester can choose the run just the single test on its own, or the group of tests, whose names all begin the same way.
  - test_provider_add - Adds a provider (**Bad naming**)
  - test_provider_add_new - Adds a new provider type
  - test_provider_add_delete - Adds a provider and then deletes it
- Where a clean-up is required, it should be carried out in a Finalizer. In this way we prevent leaving an appliance dirty if the test fails as the clean up will happen regardless.
- Keep all properties, fixtures and functions together

Fixtures

Fixtures are not only responsible for setting up tests, but also cleaning up after a test run, whether that test run succeeded or failed. addfinalizer is very powerful. finalizer functions are called even if tests fail.

When writing fixtures, consider how useful they might be for the overall project, and place them accordingly. Putting fixtures into a test module is rarely the best solution. Instead, try to put them in the nearest conftest.py. If they're generic/useful enough consider putting them into one of the *fixtures/* directory for use in *cfme_tests* or the *plugin/* directory for use in both projects.

2.3.3 This Document

This page is subject to change as our needs and policies evolve. Suggestions are always welcome.

2.4 Documenting cfme_tests

2.4.1 Overview

In addition to PEP 257, inline documentation of the cfme_tests code adheres to the Google Python Style Guide. The Google-recommended docstring format is very easy to both read and write, and thanks to the cartouche library, it's parseable by sphinx, which we use to generate our documentation.

The documentation is built and hosted by the excellent readthedocs service, but should be built locally before making a pull request.

2.4.2 docstrings

The napoleon library parses our docstrings and turns them into nicely rendered API docs in the sphinx output. As such, we should follow napoleon's usage guidelines when writing docstrings:

> https://pypi.python.org/pypi/sphinxcontrib-napoleon

According to PEP 257, docstrings should use triple double-quotes, not triple single-quotes (“”” vs. ‘’‘).

Example:

```
"""This is a docstring."""

'''This is not a docstring.'''
```

2.4.3 Documenting Tests

Tests are documented slightly differently to modules, in that they require certain extra information that isn't required for a module/class/function. If a test uses the testgen library it **must** also specify a test_flag in the metadata section. An example of this is shown below.

```
"""Tests provisioning via PXE

Metadata:
    test_flag: pxe, provision
"""
```

These flags are also defined in the `cfme_data.yaml` file, under the `test_flags:` key. A provider in the `cfme_data.yaml` can opt out of collection for a particular *test_flag* by including the flag in the list of `excluded_test_flags:` key in the providers stanza. All of the flags listings are listed in comma separated format. This was chosen to cut down on syntax characters and all values are whitespace stripped.

For a test to be collected for a provider:

- the *test_flag* must be listed in the `cfme_data.yaml` file `test_flags:` key
- the *test_flag* must be listed in the metadata section of the test's docstring
- the *test_flag* must **NOT** appear in the list of `excluded_test_flags:` for a particular provider

2.4.4 Linking new modules

As new modules are created, they'll need to be added to the documentation tree. This starts in the `toctree` directive in `docs/index.rst`. Each entry in that tree references other .rst files in the docs/ directory, which can in turn reference documentation sources in their own `toctree` directives (ad infinitum).

Once the rst file has been inserted into the `toctree` (assuming one had to be created), sphinx needs to be told to generate documentation from the new code. We use sphinx's autodoc feature to do this, and it looks like this:

```
.. automodule:: packagename.modulename
```

The paramater passed to the `automodule` should be the importable name of the module to be documented, `cfme.login` for example.

There is no hard and fast rule for where things should go in the toctree, but do try to keep the docs well-organized.

2.4.5 Building the Docs

Prior to pushing up new code, preview any new documentation by building to docs locally. You can do this using the sphinx-build command. From the `cfme_tests` directory:

```
sphinx-build -b html docs/ docs/build/
```

This will build html documentation based on the sources in the docs/ directory, and put them in the docs/build/ directory, which can then be opened in a browser:

```
google-chrome docs/build/index.html
# or...
firefox docs/build/index.html
```

2.4.6 Old and busted

The "legacy" code (contained mainly in the `pages/` and `tests/` directories) will not be documented here. Time spent documenting that code is better spent converting it to the new page style, in the `cfme/` directory.

2.5 Setting up editors

2.5.1 Sublime

The "supported" editor of choice for working on this project is Sublime Text 2 (sublime), though these instructions will likely also work for Sublime Text 3. Of course you're free to use whichever editor helps you be the most productive, but the preponderance of Sublime users on the team make it the most useful target for our development environment setup documentation.

Getting Started

Get sublime

To begin, sublime must be installed. It is distributed via a tarball from the sublime download page. This tarball can be extracted anywhere. A likely place is in your home folder. Once extracted, run the `sublime_text` executable in the new directory to start the editor.

Configure sublime for Python

By default, sublime will attempt to autodetect indentation. When this autodetection fails, it will fall back to using 4-space tab stops, but using tabs instead of spaces. To easily address this, open any .py in the editor, and then select `Preferences > Settings - More > Syntax Specific - User` from the menu. This should open up `Python.sublime-settings`. In this file, enter the following options and save:

```
{
    "detect_indentation": false,
    "rulers": [100],
    "tab_size": 4,
    "translate_tabs_to_spaces": true,
    "use_tab_stops": true
}
```

This will force python files to match our code style guidelines, which use spaces instead of tabs with an indentation of 4 spaces.

The `rulers` option will also draw a vertical line at 100 characters as a visual aid to keep lines from getting too long. Additional integer values can be added to the `rulers` list; it might be useful to also have a rule at 80 columns as a "soft limit", for example.

Package Control

Once sublime is up and running, we'll need to install some package management, which we'll be using hereafter to bring in sublime extensions. Follow the installation instructions here. Be sure to follow the instructions for Sublime Text 2, unless you're beta testing Sublime Text 3.

Note: When installing packages, it is sometimes necessary to restart sublime for the installed packages to initialize. For simplicity, it is probably easiest to restart sublime after installing any package. Restarting sublime after changing configuration files should not be necessary.

SublimeCodeIntel

Install the SublimeCodeIntel package. Select `Preferences > Package Control` from the program menu, then choose "Install Package". Enter "SublimeCodeIntel". Once installed, SublimeCodeIntel will provide autocompletion for imports and function/method calls.

SublimeCodeIntel will autodetect python names from project directories (visible in the sidebar) for autocompletion, but it won't detect builtins or installed libraries. To enable this, SublimeCodeIntel needs to be given a hint. It looks for config files in `.codeintel` directories inside of project directories, so we'll be putting the hint there. The `cfme_tests` directory is the perfect place for the `.codeintel` directory, so ensure that the `cfme_tests` directory has been added to your current project. If not, `Project > Add Folder to Project...`, and select your `cfme_tests` directory.

Using your tool of choice (for example, a shell or sublime itself), make the `.codeintel` directory under `cfme_tests`. Inside that directory, create and edit the file `config` (`cfme_tests/.codeintel/config`). Like most sublime configuration files, the content of this file is a python dictionary. It looks very similar to JSON, which is used in most sublime configuration files, so be mindful of the different syntax.

Insert the following:

```
{
    "Python":
    {
        "codeintel_scan_files_in_project": True,
        "python": "/path/to/virtualenv/bin/python",
        "pythonExtraPaths":
        [
            "/path/to/virtualenv/lib/python2.7/site-packages"
        ]
    }
}
```

Remember to change the `/path/to/virtualenv` strings to be the actual path to your virtualenv. `python` should point to the virtualenv's python interpreter.

Relative paths can be used here, and will be relative to the project folder (in this case, `cfme_tests`), not the location of this config file. So, if `cfme_tests` is in the same directory as the virtualenv's `bin` and `lib` directory, The paths for `python` and `pythonExtraPaths` could start with `../bin` and `../lib`, respectively.

Flake8 Lint

Using Package Control, install the "Python Flake8 Lint" package. To apply our specific style exceptions to this package, edit the configuration. Via the menu, choose `Preferences > Package Settings > Python Flake8 Lint > Settings - User`. In the settings file that opens, enter our exceptions:

```
{
      "pep8_max_line_length": 100,
      "ignore": ["E128"]
}
```

Flake8 lint will pop up every time you save a file, and does an excellent job of keeping you linted while you code.

Trailing Spaces

Using Package Control, install the "Trailing Spaces" plugin. This highlights trailing spaces so you can clean them up before flake8 sees them.

2.5.2 Sublime Text 3

Sublime Text 3 is currently in beta, but it is perfectly usable for python development. I will show you my setup here (`mfalesni`). Prerequisities are the same as for ST2 (Package Control).

Recommended Extensions and Settings

SublimePythonIDE

It is a rewrite of SublimeRope for ST3. It is both Python Autocompletion and PEP8 checker. Install it from package manager the same way is described in chapter about ST2.

After installation, go to `Preferences -> Package Settings -> SublimePythonIDE -> User` and insert this code:

```
{
   "open_pydoc_in_view": true,
   "create_view_in_same_group": false,

   // Linter settings
   "python_linting": true,
   "python_linter_mark_style": "outline", // "none" or "outline"
   "python_linter_gutter_marks": true,
   "python_linter_gutter_marks_theme": "alpha", // see folder gutter_mark_themes
   "pep8": true,
   "pep8_ignore": ["E128"],
   "pep8_max_line_length": 100,
   "pyflakes_ignore": []
}
```

For the project file (`Project -> Edit Project`), use this code:

```
{
  "folders":
  [
    {
      "follow_symlinks": true,
      "path": "/home/mfalesni/sublime-workspace/cfme_tests",
    },

    {
      "follow_symlinks": true,
```

```
      "path": "/home/mfalesni/sublime-workspace/whatever_else_directory_you_need",
    },
  ],

  "settings":
  {
    "python_interpreter": "/home/mfalesni/sublime-workspace/.cfme_tests_ve/bin/python",
    "tab_size": 4,
  },
}
```

Of course, replace the paths according to your setup. `python_interpreter` is the path for your virtualenv python.

From now, Sublime will know about all modules that are in virtualenv/cfme_tests namespace.

When you right-click a symbol, you can view a documentation, or jump to the symbol definition.

GitGutter

Very good plugin, showing you lines that are added/modified/removed in your git repository in form of marks on left side of the editor window. (first suggested by jkrocil)

BracketHighlighter

Simple plugin that shows you location of brackets, parenthesis and others that you are in on left side of editor window.

Neon color scheme

You might find default colour theme a bit humdrum. I use Neon color scheme, which uses more colours and the colouring depends on the context so one has better view on the situation.

To install, simply install `Neon Color Scheme` package. Then open `Preferences -> Settings - User` and add this entry `"color_scheme": "Packages/Neon Color Scheme/Neon.tmTheme"` to the conf dict.

Python Improved

Together with Neon, this package makes python source code better readable. Install with package manager `C-P -> Install Package -> Python Improved`. Then after installation, open whatever python source file you like, click `View -> Syntax -> Open all with current extension as ...` and select PythonImproved.

2.5.3 emacs

So far the best emacs setup I've (`jweiss`) found is iPython notebook, combined with the ein emacs package (emacs iPython notebook).

Installing iPython and its Emacs client

iPython

See the install docs.

ein

Emacs iPython Notebook is the emacs client for iPython.

The official ein package does not work with the latest ipython. I built a package from the fork of ein that does work. You can get the package from the internal repository listed below. You should also add the Melpa repository.

```
(add-to-list 'package-archives
  '("melpa" . "http://melpa.milkbox.net/packages/") t)
(add-to-list 'package-archives
  '("jweiss" . "http://qeblade5.rhq.lab.eng.bos/isos/emacs-package-archive/") t)
```

You can then run `M-x package-install, ein` in emacs to install ein.

Then in a shell somewhere, you can start up iPython notebook process. This is the python process that will intepret all the code you will be sending it.

```
$ source ~/my-virtual-env/bin/activate
$ cd ~/my-project
$ ipython notebook
```

Then in emacs, run `M-x ein:notebooklist-open`. It will prompt you for a port (default 8888). This will bring up the EIN environment, where you can evaluate python snippets (and edit them and evaluate them again). You can also save the notebook to use your snippets again later. The outputs are also saved.

Starting iPython from within Emacs

I wrote a little bit of elisp to start a iPython notebook process for you from within emacs. It's easier than having to type shell commands every time. It requires the `magit` package, which I highly recommend (it is a git client for emacs).

```
(autoload 'magit-get-top-dir "magit" nil t)

(defun magit-project-dir ()
  (magit-get-top-dir (file-name-directory (or (buffer-file-name) default-directory))))

(defun start-ipython-current-project (virtualenv-dir)
  (interactive
   (let ((d (read-directory-name "VirtualEnv dir: " "~/.virtualenvs/" nil t)))
     (list d)))
  (save-excursion
    (let ((buf (get-buffer-create
                (generate-new-buffer-name (file-name-nondirectory
                                           (directory-file-name (file-name-directory (magit-project-d
      (shell buf)
      (process-send-string buf (format ". %s/bin/activate\n" virtualenv-dir))
      (process-send-string buf (format "cd %s;ipython notebook\n" (magit-project-dir))))))
```

To use the above snippet,

- Go to any buffer that's visiting any file in your project (or any buffer whose `pwd` is in your project)

- `M-x start-ipython-current-project`
- At the prompt, input the directory where your virtualenv lives

It will start ipython in emacs' shell buffer.

Autosave Notebooks

Unlike the iPython web interface, ein does not autosave notebooks by default. Here is a snippet that will enable autosave (notebooks are saved every time you execute a cell)

```
;; ein save worksheet after running cell
(eval-after-load 'ein-multilang
  '(defadvice ein:cell-execute (after ein:save-worksheet-after-execute activate)
     (ein:notebook-save-notebook-command)))
```

Flake8 Lint

Flycheck is recommended because it highlights the column where the problem occurs instead of just the line.

Run `M-x package-install`, `flycheck`, and see the Flycheck homepage.

You can use the global mode as described on the homepage, or to just enable flymake for python files

```
(autoload 'flycheck "flycheck-mode")
(eval-after-load 'python
  '(add-hook 'python-mode-hook 'flycheck-mode))
```

Recommended

Magit Emacs client for git and a huge time saver. All git commands are a single keypress, pretty views of diffs, branches, remotes, etc. Package is `magit`.

Ido and Smex `ido` package (now built into emacs) for filename and buffer name completion, `smex` for `M-x` command completion.

Smartparens Inserts parens, brackets, quotes, etc in pairs. Keeps parens balanced, allows you to edit paren-delimited structures logically instead of as plain text (designed for lisp but also works on html, xml, json, etc). Replaces paredit, an older and more well-known tool that does the same thing. Package `smartparens`.

Autocomplete Code completion for emacs. Package is called `autocomplete`, see `ein` docs for how to enable in python buffers.

Undo Tree Edit with confidence! Keeps track of all your buffer changes, even stuff you undid and re-did on top of. Package is called `undo-tree`.

yagist Create a github gist (paste) from a region or buffer with a single keypress, and the link to the gist is automatically inserted into the clipboard so you can easily paste it into IRC.

Multiple cursors Extremely powerful editing tool, best described with this video. Package is `multiple-cursors`.

2.6 Gotchas

Selenium has a few quirks which have caused us immense amounts of debugging time. If you are facing strange issues with Selenium that you can't explain and this usually boils down to "Selenium is lying to me", please check this page first before spending vast amounts of time debugging .

2.6.1 Selenium is not clicking on the element it says it is

Sometimes, under certain circumstances, Selenium doesn't click on the element you tell it to. The symptoms of this include having a WebElement that gives a certain value when queried with `.text()` and then Selenium actually clicking on the wrong element. This has been observed happening when there is a frame or some other element where horizontal scrolling has been introduced. A typical example would be in the left hand tree items in the System Image Type under the `Infrastructure > PXE` menu. If one system image name is 256 characters, this causes the problem to manifest.

2.6.2 Selenium is not sending the keys I tell it to, or is filling the box with junk

If you have a file in the root of your project which has the same name as an item of text that you are trying to send to an input element. Selenium tries to be clever and replaces this text with a path, representing that file as it believes you are trying to upload it. Currently we do not have a way to disable this with the Python bindings, so just be wary of naming files that have the same name as text you may want to use further down the line. This was first discovered when a menu.php file used to exist in the root dir for checking PXE. When the menu filename box was due to be filled in with `menu.php` it was instead filled in with `/tmp/288762525-2350923r09u2-29u2o3ur23/23982986498264928file/menu.php` which caused the PXE refreshes to fail every time ;)

2.7 flake8

There are many handy tools that can be used to check your code against established python style. A tool called *flake8* exists to combine these tools into one easy-to-use package. *flake8* is used by reviewers on pull requests for style compliance, so it's a good idea to run *flake8* before submitting code for review.

Note: All new content in pull requests is expected to pass flake8 linting.

2.7.1 Manual Invocation

To use flake8 in our project, first install it: `pip install flake8` or `easy_install flake8`.

Some flags are required to deal with our specific alterations to python style:

- We allow lines up to 100 characters in length; add `--max-line-length=100`
- We indent block statement line continuations twice, even in function defs; add `--ignore=E128`

Then, aim it at the python file (or files) being edited:

```
flake8 --max-line-length=100 --ignore=E128 path/to/python_module.py
flake8 --max-line-length=100 --ignore=E128 path/to/python/package/`
```

These settings can be stored as defaults in a config file. By default, flake8 looks in `~/.config/flake8`. Here is an example file that adheres to our style guidelines:

```
[flake8]
ignore = E128
max-line-length = 100
```

2.7.2 IDE Integration

Sublime Text 2 & 3

The excellent Flake8 Lint for the sublime text editor will do automatic linting using the flake8 tool. To configure it to follow our guidelines, Add the following options to your `Flake8Lint.sublime-settings` file:

```
"pep8_max_line_length": 100
"ignore": ["E128"]
```

Emacs

See flymake-python-pyflakes.el.

If you have Melpa or Marmalade package repos already set up, you can install the package by `M-x package-install`, `flymake-python-pyflakes`.

To activate on all Python files, add this to your emacs configuration:

```
(autoload 'flymake-python-pyflakes-load "flymake-python-pyflakes" nil t)
(eval-after-load 'python
  '(add-hook 'python-mode-hook 'flymake-python-pyflakes-load))
```

To use flake8 and our particular rules:

- `M-x customize-group`, `flymake-python-pyflakes`
- Set `Flymake Python Pyflakes Executable` to `flake8`
- Add to `Flymake Python Pyflakes Extra Arguments:` `* --max-line-length=100 * --ignore=E128`

Others

If your IDE isn't listed here, feel free to add instructions above!

2.8 Page Development

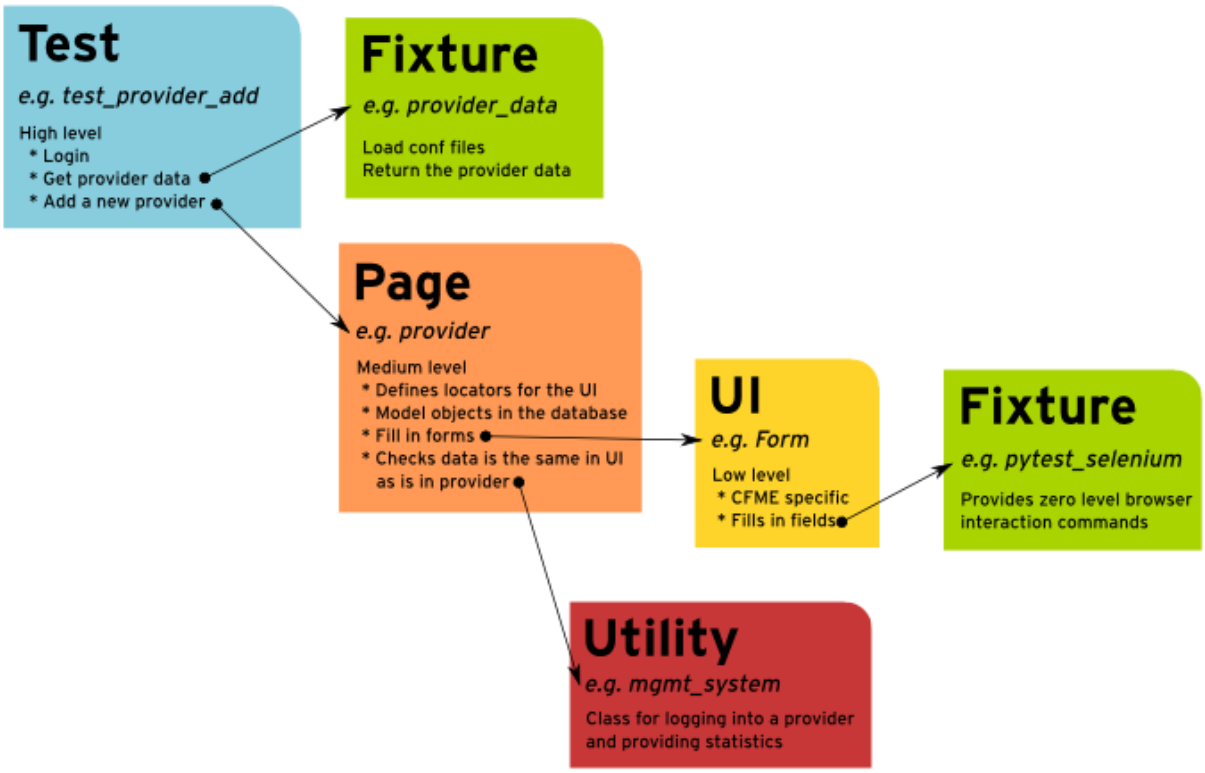
2.8.1 Introduction

This file is intended to explain how pages should be developed and specify what functionality should exist in them. Refering to the image below, a page model should contain the following:

- Locators to elements which appear only on that page or set of pages related to the object in question.
- Menu items to graft onto the main menu for use inside the pages and operations.
- Classes that are related to objects in the database which are acted upon with CRUD operations on that page or set of pages.

- Methods which are used to CRUD operations, with an aim that any of these can use a different backend where possible.

Anything else should be added as part of a web_ui component, a fixture or a utility.



Pages are read-only python modeling of the CFME UI, allowing the functional tests of the UI to be ignorant of the underlying page structure. As such, UI elements (pages, regions, forms, etc.) modeled in *cfme_pages* must provide helper methods and properties to expose a usable interface to *cfme_tests*. This is explained in more detail in the section on *Writing Tests* in *Style Guide*.

Pages should be modeled as a part of writing tests. Code in *cfme_pages* must never depend on code in *cfme_tests*.

When writing pages, a few points should be noted:

- Follow the standard naming convention for locators
  - Name of element, followed by type from the type list
- Type list: button, select, text, textbox, radio, option
- Ensure that your element is presented in an expected way. As an example, presenting a div containing and unordered list in one place and simply the unordered list in another, causes an unpredictable inconsistency as to how to handle the locator.
- Try to avoid using localized text as part of a locator where possible

2.8.2 Example

Since the majority of development happens in tests and page models, we will deconstruct a page to see how it is composedW.e will now look at a specific example using the `cloud/provider.py` file. Note that there may be subtle differences between the file and this documentation as the content evolves. It is expected that this document will be updated with any significant changes.

Imports

To begin with we have the imports, we have added comments after each to specify their use:

```
from functools import partial                    # Standard library
from selenium.webdriver.common.by import By      # Convenience functions for locators, ID etc.
import ui_navigate as nav                        # Navigation library
import cfme                                      # Core cfme module
import cfme.web_ui.menu                          # Standard menu for grafting additional menus onto
from cfme.web_ui import Region, Quadicon, Form   # Loads the Region, Quadicon and Form UI elements
import cfme.web_ui.flash as flash                # Flash message handler
import cfme.fixtures.pytest_selenium as browser  # The selenium zero-level functions
import utils.conf as conf                        # Loads all configuration from the yamls
from utils.update import Updateable              # Updatable class to give update capabilities
import cfme.web_ui.toolbar as tb                 # Toolbar UI element for clicking Center Toolbar
```

Locators

Now that we have the tools we need to begin crafting a page, we can start to define the locators. Locators are used to point to elements on a page. They can reference any html element and will typically be used:

- to pull out text for comparision, *e.g. making sure a flash message matches what is expected.*
- to send text to, *e.g. for inputting data into forms.*
- to click, *e.g. a button.*

Below is an excerpt from the set of locators on the page:

```
page = Region(
    locators={
        'add_submit': "//img[@alt='Add this Cloud Provider']",
        'creds_validate_btn': "//div[@id='default_validate_buttons_on']"
                              "/ul[@id='form_buttons']/li/a/img",
        'creds_verify_disabled_btn': "//div[@id='default_validate_buttons_off']"
                                     "/ul[@id='form_buttons']/li/a/img",
        'cancel_button': "//img[@title='Cancel']",
        'save_button': "//img[@title='Save Changes']",
    },
    title='CloudForms Management Engine: Cloud Providers')
```

Locators are usually supplied as a Python dict. The key is a name which should conform to the *Abbreviations and Naming Conventions* guidelines. The value is one of three:

- an XPATH string, as depicted above
- a tuple, containing a CSS selector, e.g. `(By.CSS_SELECTOR, "div.dhtmlxInfoBarLabel-2 > ul")`
- a tuple, containing an ID selector, e.g. `(By.ID, "text_button")`

These elements can then be used to perform actions as shown later in the file by:

```
if cancel:
    browser.click(page.cancel_button)
```

Forms

A recent edition to the codebase has been the introduction of Forms using the `cfme.web_ui.Form` web_ui component. Forms allow the defining of a set of locators which correspond to fields. Data can then be sent to the form object to fill in the fields automatically, without worrying about field type. We begin by defining a Form:

```
form = Form(
    fields=[
        ('type_select', "//*[@id='server_emstype']"),
        ('name_text', "//*[@id='name']"),
        ('hostname_text', "//*[@id='hostname']"),
        ('ipaddress_text', "//*[@id='ipaddress']"),
        ('amazon_region_select', "//*[@id='hostname']"),
        ('api_port', "//*[@id='port']"),
    ])
```

Notice that a Form is very similar to a Region. In fact, a Form inherits a Region so as above when we clicked on the cancel button by referencing it as an attribute of the page object. We can do the same here. `browser.set_text(form.api_port, "6000")`, for example, would set the text of the locator described by key value `api_port` to `6000`.

The details to fill in the form are loaded into a variable inside the management object called `OpenStackDetails` in this case:

```
def __init__(self, hostname=None, ip_address=None, api_port=None):
    self.details = {'hostname_text': hostname,
                    'ipaddress_text': ip_address,
                    'api_port': api_port,
                    'type_select': 'OpenStack'}
```

These details are then passed to the Forms fill_fields function:

```
details.details.update({'name_text': self.name})
form.fill_fields(details.details)
```

Notice that there has been an amendment to the `details` dictionary when it has been passed into the `_fill_details` function, and a new key/value called `name_text` has been added.

The `cfme.web_ui.Form.fill_fields()` Form method then takes these values, does an inspection of the element types to find out how to handle them (you couldn't set text on a select box for example), and then sets the values in the most appropriate way.

Toolbar

A Toolbar button can be accessed by simple using it in the following way:

```
tb.select('Configuration', 'Add a New Cloud Provider')
```

but in cases where we may have several `Configuration` buttons, we can make things a little simpler to type by making use of `partial`. Which takes a function and some arguments to create a shortened form of the function call. In the example below, we define this:

```
cfg_btn = partial(tb.select, 'Configuration')
```

We can now use the toolbars by doing something like the following:

```
cfg_btn('Add a New Cloud Provider')
```

Navigation Menu

In our provider page we are going to hook in the toolbar button presses to the navigation tree. This means we are able to do something the code below and have the page execute the toolbar button clicks to navigate to the page in question. We could simply use the `cfme.web_ui.toolbar.select()` function, but to make it clearer that we expect to navigate away from the current page, using the `nav.goto` function is better:

```
nav.go_to('cloud_provider_new')
```

We need to add a few buttons to the center menu to handle "Add a New Cloud Provider", "Discover Cloud Providers" and a special case.

The `ui_navigate` module handles all tree based navigation. It it prepopulated with all of the main navigation and is then extendable. This is called grafting. In the example below we add three more elements onto the tree:

```
nav.add_branch('clouds_providers',
               {'cloud_provider_new': lambda: cfg_btn('Add a New Cloud Provider'),
                'cloud_provider_discover': lambda: cfg_btn('Discover Cloud Providers'),
                'cloud_provider': [lambda ctx: browser.click(Quadicon(ctx['provider'].name)),
                                   {'cloud_provider_edit':
                                    lambda: cfg_btn('Edit Selected Cloud Provider')}]})
```

The first two elements are added by defining the name of the navigation leaf and a lambda function determining what to do to get to that page.

2.9 Development Tips and Tricks

2.9.1 Introduction

This document is intended to explain some of the extra bits of the framework that are there to make your life easier. Not everything is included here and we encourage people to add new tricks as they are developed and rediscovered.

2.9.2 Version Picking

Dealing with multiple releases, it's obvious that some things change from version to version. A lot of the time, these changes are simple, such as a string change. So that we can continue using the same codebase for any version, we define the idea of version picking. Version picking essentially returns an object depending on the version of an appliance. It's particularly useful for things like locator changes because most of the element handling routines are version picking away. This means if they receive a dict as an argument, they will automatically try to resolve it using the version picking tool. To use version picking is easy:

```
from utils import version

version.pick({'5.4': "Houses",
              '5.3': "House",
              version.LOWEST: "Boat"})
```

In this example, if the version is below 5.3, the `Boat` will be returned. Anything between 5.3 and 5.4 will return `House` and anything over 5.4 will return `Houses`. There is also a `version.LATEST` which points to upstream appliances.

2.9.3 Defining blockers

Sometimes we know a test fails due to a bug in the codebase. In order to make sure the test isn't run and attributing an extra fail that doesn't need to be investigated, we mark it with a meta marker. The meta marker is incredibly useful and integrates with our Bugzilla implementation to ensure that if a bug is still on DEV, or hasn't even been assigned yet, that the test won't run. The syntax is really easy:

```
@pytest.mark.meta(blockers=[12345, 12346])
def test_my_feature():
    # Test the new feature
    pass
```

Note the two bug numbers 12345 and 12346. More information can be found in the `fixtures.blockers` fixture.

2.9.4 Using blockers in tests

On the odd occasion, you don't want to disable an entire test, but just a part of it, until a bug is fixed. To do this, we can specify a bug object and ask the framework to skip if a certain bug exists and is not closed. The syntax is pretty simple:

```
def my_test(provider, bug):
    ui_bug = bug(12234)
    if not ui_bug:
        # Do something unless the bug is still present in which case, it will be skipped
```

2.9.5 Uncollecting tests

There are times when conditions dictate that we don't need to run a test if a certain condition is true. Imagine you don't want to run a test if the appliance version is below a certain value. In these instances, you can use `uncollectif` which is a pytest marker:

```
@pytest.mark.uncollectif(lambda: version.current_version() < '5.3')
def test_my_feature():
    # Test the new feature
    pass
```

Now if the version of the appliance is less than 5.3. Then the test will not be skipped, it will never even try to be run. This is ONLY to be used when a certain test is not valid for a certain reason. it is NOT to be used if there is a bug in the code. See the *Defining blockers* section above for skipping because of a bug.

2.9.6 Running commands on another appliance

We implement a small appliance stack in the framework. When a test first starts it loads up the base_url appliance as the first appliance in the stack. From then on, all the browsing operations, database operations and ssh commands are run on the top appliance in the stack. From time to time it becomes necessary to run commands on another appliance. Let's say you were trying to get two appliances to talk to each other, in this case, you would use the context manager for appliances.

By default, even if you add a new appliance onto the stack, the browser operations will keep happening on the last appliance that was used, however, there is a simple way to steal the browsers focus, and this is detailed in the example below:

```
appl1.ipapp.browser_steal = True
with appl1.ipapp:
    provider_crud.create()
```

In the example we have already created a new `utils.appliance.Appliance` object and called it `appl1`. Then we have set it to steal the browser focus. After this, we enter the context manager `appl1.ipapp` and are able to run operations like provider creates.

This is also why you should use `ssh_client` and `db` access from the `store.current_appliance` and not from the modules directly. If someone else uses your code and is inside an appliance context manager, the commands could be run against the wrong appliance.

2.9.7 Logging in as another user

In a similar way to the *Running commands on another appliance* section above, we implement a context manager for user operations. This allows the test developer to execute a section of code as a different user and then return to the original user once complete.

A major advantage of this, is that the User object used for the CM operations is the same as the `cfme.configure.access_control` object. This means that you can *create* a new user using the `cfme.configure.access_control.User` object and straight after use it as the context manager object:

```
cred = Credential(principal='uid', secret='redhat')
user = User(name='user' + fauxfactory.gen_alphanumeric(),
    credential=cred)
with user:
    sel.force_navigate('dashboard')
```

The `User` object stores the previous `User` object in a cache inside itself and on exiting the context, returns this to the pytest store as the *current* user so that future operations are performed with the original user.

2.9.8 Invalidating cached data

In order to speed things up, we cache certain items of data, such as the appliances version and configuration details. When these get changed, the cache becomes invalid and we must invalidate the cache somehow. It's not as tricky as it sounds. We have created a signals module to help with this. You can find the list of used signals in the `utils.signals` file. An example of this would be the server name. If the server name is changed. We need to invalidate the cache. To do this, we do the following:

```
def update(self):
    """ Navigate to a correct page, change details and save.

    """
    sel.force_navigate("cfg_settings_currentserver_server")
    fill(self.basic_information, self.details)
    # Workaround for issue with form_button staying dimmed.
    if self.details["appliance_zone"] is not None and current_version() < "5.3":
        sel.browser().execute_script(
            "$j.ajax({type: 'POST', url: '/ops/settings_form_field_changed/server',"
            " data: {'server_zone':'%s'}})" % (self.details["appliance_zone"]))
    sel.click(form_buttons.save)
    # TODO: Maybe make a cascaded delete on lazycache?
    fire('server_details_changed')
```

Notice the last line in this snippet which fires off the `server_details_changed` signal. You as the user don't need to care how to invalidate the cache, you just need to let the system know you've done it. Any time any one

updates the server details using the `cfme.configure.configuration.BasicInformation` class from the configuration module, this signal will automatically be fired, so unless you are doing something out of the ordinary, you shouldn't have to worry about it. However the signals are there if you need to. Note that the cache invalidation happens on the `current_appliance` in the stack. See the *Running commands on another appliance* section for more details.

2.9.9 pytest store

The pytest store provides access to common pytest data structures and instances that may not be readily available elsewhere. It can be found in `fixtures.pytest_store`, and during a test run is exposed on the pytest module in the store namespace as `pytest.store`.

2.9.10 Test generation (testgen)

We try to consolidate common test generation functions in the `utils.testgen` module. When parametrizing tests with the `pytest_generate_tests` hook, check the testgen module to see if there are functions available that already parametrize on the axis you want (usually by provider, but there are some other helpers in there).

2.9.11 Working with file paths

For any path in the project root, there are several helper functions that can be used. Look at the `utils.path` module for the complete list of pre-configured directories and available functions.

2.9.12 Appliance object SSH gremlins

If you get seemingly random SSH errors coming from `utils.appliance`, you might be facing the problem that some of the methods inside of the class does some version picking, or database connection outside of the object scope or whatever that is supposed to touch the target appliance but does not go through the object that you are in, but the `utils.appliance.IPAppliance` object itself is not pushed to the appliance stack in `fixtures.pytest_store`. So instead of using the IP address of the appliance the object is pointed to, it uses whatever was set before, either the `base_url` one or something that was pushed before. The solution is to wrap that in a `with` block, like this (presuming we call this code inside `utils.appliance.Appliance`):

```
with self.ipapp as ipapp:
    ipapp.wait_for_ssh()

    self._i_do_verpicking("and fail randomly when not in with block")

    success("!")
```

Until we come with a better solution, this will bite us from time to time when we forget about it.

2.10 Selenium over VNC

2.10.1 Purpose

The goal of this page is to explain how to set up a remote display that can run selenium tests, and manage/contain test-related web browser windows.

Note: This document assumes that you're running a recent Fedora release, and already have a working selenium

setup for cfme_pages as explained in the cfme_pages README.

While these instructions are specific to tigervnc, available in Fedora 11 onward, they can be easily adapted to use other VNC packages.

2.10.2 Install requirements

We'll need a VNC server (tigervnc-server), a lightweight window manager to run inside that VNC server (fluxbox), and a terminal emulator that can run inside the lightweight window manager (xterm):

```
# yum install tigervnc-server fluxbox xterm
```

We'll also need the standalone selenium server, which will run inside the VNC server. You'll have to download the Selenium Server jar from their download page (the file should start with `selenium-server-standalone`):

- Standalone Selenium Server

You'll want to put this somewhere relatively safe (e.g. not /tmp), and remember where you put it for later.

2.10.3 Configure the VNC server

If it isn't already there, create a `.vnc` directory in your home directory:

```
$ mkdir ~/.vnc
```

Set a password

Using the `vncpasswd` utility, enter your desired vnc password and save it to a file:

```
$ vncpasswd ~/.vnc/passwd
```

The `~/.vnc/passwd` file stores an obfuscated version of the password entered, so you'll either want to use a memorable password or write the password down. Also, passwords longer than 8 characters will be truncated. More on this Security).

Configure the startup script

Create or modify `~/.vnc/xstartup`. This script is run inside the VNC server, and bootstraps the environment. It must be executable, and needs to do the following things:

- If using chrome/chromdriver, configure the `$PATH` environment variable so that the selenium server can find the `google-chrome` and `chromedriver` binaries
- Start the window manager (fluxbox)
- Start the selenium server in a terminal window (xterm, selenium-server-standalone-VERSION.jar)

Here's an example script that does those things:

```
#!/bin/sh

# Set up the environment so selenium can find everything it might want
# (namely chrome and chromedriver)
export PATH="/path/to/google/chrome/directory/:/path/to/chromdriver/directory:$PATH"
```

```
# Start the window manager
fluxbox &

# Start the selenium server
xterm -maximized -e java -jar /path/to/selenium-server-standalone-VERSION.jar -ensureCleanSession -t
```

Important things: * The script **MUST** start with *#!/bin/sh* (or your shell shebang of choice). * The script **MUST** be executable (*chmod +x ~/.vnc/xstartup*)

Start the server

```
$ vncserver :99
```

This will start a local VNC server, listening on display 99 and port 5999. The string ':99' is all you should need to enter into connection prompts to connect to VNC display 99. This example uses :99, but any other reasonable display number can be used throughout this guide. This server will use the password stored in `~/.vnc/passwd`.

View your new desktop

To connect to the server, there are a few tools that you can use. GNOME has a built-in VNC viewer called `vinagre`, and tigervnc also provides one. Make sure at least one of these is installed (package names are `vinagre` and `tigervnc`), and then connect to the VNC server. Both tools have graphical and command-line interfaces.

To connect using either command-line tool, pass the display number as the first argument:

```
$ vncviewer :99
# -or-
$ vinagre :99
```

Enter the VNC password that you set [above](Selenium-over-VNC#set-a-password). Once connected, you should see your selenium server running in a maximized xterm window.

Help for the graphical interfaces to these tools is provided by the tools themselves, but they're pretty straightforward.

2.10.4 Configuring the selenium client

In your existing test environment, have a `env.yaml` file, with a `webdriver` key in the `browser` root key. This should be set to `Remote`, which is the default from the `env.yaml.template` it informs the test suite to use the remote selenium server now running inside your VNC server.

We also need to set the **Remote** options, by setting the `desired_capabilities` key to have the `platform` and `browsername` For Fedora, the platform would be `LINUX`, but selenium recognizes any of the following (possibly more).

- WINDOWS
- XP
- VISTA
- MAC
- LINUX
- UNIX

An example of the yaml is below:

```
base_url: https://10.11.12.13
browser:
    webdriver: Remote
    webdriver_options:
        desired_capabilities:
            platform: LINUX
            browserName: 'chrome'
```

2.10.5 Security

Simply put, VNC isn't very secure. Its connections aren't encrypted, and its passwords can only be a max of 8 characters long. For this reason, I recommend having the VNC server bind to the loopback interface. Fortunately, this is easily done by passing the `-localhost` flag to vncserver, like this:

```
$ vncserver :99 -localhost
```

No changes need to be made in the way clients are told to connect to support this change, but it prevents other users from connecting to and interacting with this VNC session remotely.

2.10.6 Recording

The `recordmydesktop` utility can be used to record test interactions for demonstration or review. Continuing with display `:99` for this example, recordmydesktop can be invoked like this:

```
$ recordmydesktop --display :99 --fps 60 -o outfile.ogv
```

In addition to specifying `--display :99`, `--fps 60` is passed to ensure no steps are missed in the recording. rescordmydesktop's default framerate has shown to be a little too low to accurately capture all of the actions taken in a test run. Finally, `-o` is passed to specify the output file.

To record test runs in one shot, the following pattern can be followed (changing the py.test invocation as needed, of course):

```
$ recordmydesktop --display :99 --fps 60 -o test_label.ogv & py.test -k test_label --highlight; pkil
```

CHAPTER 3

Modules

3.1 cfme package

3.1.1 Subpackages

cfme.automate package

Submodules

cfme.automate.buttons module

cfme.automate.explorer module

class cfme.automate.explorer.**Class**(*name=None*, *display_name=None*, *description=None*, *inherits_from=None*, *namespace=None*, *setup_schema=None*)

Bases: cfme.automate.explorer.CopiableTreeNode, utils.update.Updateable

Represents a Class in the CFME ui.

Providing a setup_schema dict, creates the Class with teh specified schema

class **SchemaField**(*name=None*, *type_=None*, *data_type=None*, *default_value=None*, *display_name=None*, *description=None*, *sub=None*, *collect=None*, *message=None*, *on_entry=None*, *on_exit=None*, *on_error=None*, *max_retries=None*, *max_time=None*)

Bases: utils.update.Updateable

get_form(*blank=False*)

Gets a form for a field that already exists (by its name). Or if blank=True, get the form for a new field. Must be on the correct page before calling this.

Class.**create**(*cancel=False*)

Class.**delete**(*cancel=False*)

Class.**edit_schema**(*add_fields=None*, *remove_fields=None*)

Class.**form = <cfme.web_ui.Form fields=[('name_text', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('disp**

Class.**name_in_table**

The item is displayed differently with display_name

Class.**name_in_tree**

The item is displayed differently with display_name

Class.**parent**

Class.**path_str**()
Returns string path to this class, eg ns1/ns2/ThisClass

Class.**schema_edit_page = <cfme.web_ui.Region title=None>**

Class.**update**(*updates*, *cancel=False*)

class cfme.automate.explorer.**CopiableTreeNode**
Bases: cfme.automate.explorer.TreeNode

class_name
Used for gathering the object name from the class name. If the name is not same, you can set it manually. This exploits the fact that the classes are named exactly as it appears in the UI, so it will work unless someone changes ui/class name. Then you can set it manually, as it contains setter.

copy_button = FormButton('Copy')

copy_form = <cfme.web_ui.Form fields=[('domain', Select('select#domain', multi=False)), ('domain_text_only', {Versio

copy_to(*domain=None*)

class cfme.automate.explorer.**Domain**(*name=None*, *description=None*, *enabled=False*)
Bases: cfme.automate.explorer.TreeNode, utils.update.Updateable

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('ns_name',), _use_id=False>), ('description', <

is_enabled

is_locked

navigate_tree()

update(*updates*)

class cfme.automate.explorer.**Instance**(*name=None*, *display_name=None*, *description=None*, *values=None*, *cls=None*)
Bases: cfme.automate.explorer.CopiableTreeNode, utils.update.Updateable

Represents a Instance in the CFME ui.

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('name_text', "//input[contains(@name,'inst_name')]"), ('display_name_text', "//inpu

name_in_table
The item is displayed differently with display_name

name_in_tree
The item is displayed differently with display_name

parent

update(*updates*, *cancel=False*)

class cfme.automate.explorer.**InstanceFields**
Bases: object

Represents the table of fields defined for instance.

It uses web-scraping to determine what fields are available. It is maybe a slight slowdown, but no better solution with similar complexity (2 SLoC) exists.

Only real drawback is that you cannot use *form* when being somewhere else than on the page.

fields = {Version ('lowest'): "//div[@id='form_div']//table[@class='style3']//td[img]", '5.4': "//div[@id='form_div']//ta

form
Returns Form filled with fields. Scraps the webpage to determine the fields.

Requires to be on the page

class cfme.automate.explorer.**InstanceFieldsRow**(*row_id*)
Bases: utils.pretty.Pretty

Represents one row of instance fields.

Parameters **row_id** – Sequential id of the row (begins with 0)

columns = ('value', 'on_entry', 'on_exit', 'on_error', 'collect')

fields = ('inst_value_{}', 'inst_on_entry_{}', 'inst_on_exit_{}', 'inst_on_error_{}', 'inst_collect_{}')

form
Returns the form with fields targeted at our row_id.

Does not need to be on the page.

pretty_attrs = ['row_id']

table = <cfme.web_ui.Table _loc={Version ('lowest'): "//div[@id='form_div']//table[@class='style3']", '5.4': "//div[@id=

class cfme.automate.explorer.**Method**(*name=None*, *display_name=None*, *location=None*, *data=None*, *cls=None*)
Bases: cfme.automate.explorer.CopiableTreeNode, utils.update.Updateable

Represents a Method in the CFME ui. *Display Name* is not supported (it causes the name to be displayed differently in different places in the UI).

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('name_text', "//input[contains(@name,'method_name')]"), ('display_name_text', "//

parent

update(*updates*, *cancel=False*)

class cfme.automate.explorer.**Namespace**(*name=None*, *description=None*, *parent=None*, *domain=None*)
Bases: cfme.automate.explorer.TreeNode, utils.update.Updateable

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('name', "//*[@id='ns_name']"), ('description', "//*[@id='ns_description']")]>

classmethod make_path(**names*, ***kwargs*)
Make a set of nested Namespace objects with the given path.

Usage:

```
#eg.
    n = Namespace.make_path("foo", "bar")
#is equivalent to:
    n = Namespace(name="bar", parent=Namespace(name="foo"))
```

update(*updates*, *cancel=False*)

class cfme.automate.explorer.**TreeNode**
Bases: utils.pretty.Pretty

exists()

name_in_table

name_in_tree

nav_edit()

nav_path

navigate_tree()

path
Returns the path to this object as a list starting from the root

pretty_attrs = ['name']

cfme.automate.explorer.**datastore_checkbox**(*name*)

cfme.automate.explorer.**get_domain_order**()

cfme.automate.explorer.**open_order_dialog_func**(*_*)

cfme.automate.explorer.**set_domain_order**(*items*)

cfme.automate.explorer.**table_select**(*name*)

cfme.automate.explorer.**tree_item_not_found_is_leaf**(*e*)
Returns true if the given exception was while navigating a tree and the item in the path that was missing was the last item.

cfme.automate.provisioning_dialogs module

class cfme.automate.provisioning_dialogs.**DialogTypeSelect**(*loc*)
Bases: utils.pretty.Pretty

pretty_attrs = ['loc']

select

class cfme.automate.provisioning_dialogs.**ProvisioningDialog**(*type*, *name=None*, *description=None*, *content=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

ALLOWED_TYPES = set([('host_provision', 'Host Provision'), ('vm_provision', 'VM Provision'), ('vm_migrate', 'VM Mig

HOST_PROVISION = ('host_provision', 'Host Provision')

VM_MIGRATE = ('vm_migrate', 'VM Migrate')

VM_PROVISION = ('vm_provision', 'VM Provision')

change_type(*new_type*)
Safely changes type of the dialog. It would normally mess up the navigation

create(*cancel=False*)

delete(*cancel=False*)

exists

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('description', <cfm

pretty_attrs = ['name', 'description', 'content']

type_nav

update(*updates*)

cfme.automate.provisioning_dialogs.**get_dialog_name**(*o*)

cfme.automate.service_dialogs module

class cfme.automate.service_dialogs.**ServiceDialog**(*label=None*, *description=None*, *submit=False*, *cancel=False*, *tab_label=None*, *tab_desc=None*, *box_label=None*, *box_desc=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

create(**element_data*)

delete(*cancel=False*)

element(*element_data*)

element_type(*each_element*)

pretty_attrs = ['label', 'description']

reorder_elements(*box*, **element_data*)

update(*updates*)

cfme.automate.simulation module

class cfme.automate.simulation.**AVPForm**(*attr_prefix='attribute_'*, *val_prefix='value_'*, *start_number=1*, *end_number=5*)

Bases: object

Maps dictionary to Attribute/Value pair subform

fill(*data*)

num_fields

cfme.automate.simulation.**simulate**(***data*)

Runs the simulation of specified Automate object.

Parameters ****data** – See sim_form for keyword reference

Module contents

cfme.cloud package

Submodules

cfme.cloud.availability_zone module A page functions for Availability Zone

var list_page A cfme.web_ui.Region object describing elements on the list page.

var details_page A cfme.web_ui.Region object describing elements on the detail page.

cfme.cloud.flavor module Page functions for Flavor pages

var list_page A cfme.web_ui.Region object describing elements on the list page.

var details_page A cfme.web_ui.Region object describing elements on the detail page.

cfme.cloud.instance module

cfme.cloud.provider module

cfme.cloud.security_group module Page functions for Security Group page

var list_page A cfme.web_ui.Region object describing elements on the list page.

var details_page A cfme.web_ui.Region object describing elements on the detail page.

cfme.cloud.stack module

class cfme.cloud.stack.**Stack**(*name=None*)

Bases: utils.pretty.Pretty

delete()

edit_tags(*tag*, *value*)

get_tags()

nav_to_output_link()

nav_to_parameters_link()

nav_to_resources_link()

nav_to_security_group_link()

pretty_attrs = ['name']

wait_for_appear()

wait_for_delete()

cfme.cloud.tenant module Page functions for Tenant pages

var list_page A cfme.web_ui.Region object describing elements on the list page.

var details_page A cfme.web_ui.Region object describing elements on the detail page.

class cfme.cloud.tenant.**Tenant**(*name*, *description*, *provider_key*)

Bases: object

exists()

Module contents

cfme.common package

Submodules

cfme.common.provider module

class cfme.common.provider.**BaseProvider**
Bases: cfme.common.Taggable

class Credential(**kwargs)
Bases: cfme.Credential, utils.update.Updateable

Provider credentials

Parameters

- **type** – One of [amqp, candu, ssh, token] (optional)
- **domain** – Domain for default credentials (optional)

BaseProvider.**STATS_TO_MATCH = []**

BaseProvider.**add_provider_button = None**

BaseProvider.**category**

BaseProvider.**create**(*cancel=False*, *validate_credentials=False*)
Creates a provider in the UI

Parameters

- **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the Provider has been filled in the UI.
- **validate_credentials** (*boolean*) – Whether to validate credentials - if True and the credentials are invalid, an error will be raised.

BaseProvider.**data**

BaseProvider.**delete**(*cancel=True*)
Deletes a provider from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

BaseProvider.**delete_if_exists**(*args, **kwargs)
Combines .exists and .delete() as a shortcut for request.addfinalizer

BaseProvider.**detail_page_suffix = ''**

BaseProvider.**edit_page_suffix = ''**

BaseProvider.**exists**

BaseProvider.**get_detail**(*ident, **kwargs)
Gets details from the details infoblock

The function first ensures that we are on the detail page for the specific provider.

Parameters ***ident** – An InfoBlock title, followed by the Key name, e.g. "Relationships", "Images"

Keywords: use_icon: Whether to use icon matching

Returns: A string representing the contents of the InfoBlock's value.

BaseProvider.**get_mgmt_system**()
Returns the mgmt_system using the utils.providers.get_mgmt() method.

BaseProvider.**get_yaml_data**()
Returns yaml data for this provider.

BaseProvider.**load_details**(*refresh=False*)
To be compatible with the Taggable and PolicyProfileAssignable mixins.

BaseProvider.**mgmt**

BaseProvider.**page_name** = ''

BaseProvider.**properties_form** = None

BaseProvider.**quad_name** = None

BaseProvider.**refresh_provider_relationships**(*from_list_view=False*)
Clicks on Refresh relationships button in provider

BaseProvider.**save_button** = None

BaseProvider.**string_name** = ''

BaseProvider.**type**

BaseProvider.**update**(*updates*, *cancel=False*, *validate_credentials=False*)
Updates a provider in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

BaseProvider.**validate**(*db=True*)
Validates that the detail page matches the Providers information.

This method logs into the provider using the mgmt_system interface and collects a set of statistics to be matched against the UI. The details page is then refreshed continuously until the matching of all items is complete. A error will be raised if the match is not complete within a certain defined time period.

BaseProvider.**version**

BaseProvider.**wait_for_delete**()

class cfme.common.provider.**CloudInfraProvider**
Bases: cfme.common.provider.BaseProvider, cfme.common.PolicyProfileAssignable

get_assigned_policy_profiles()
Return a set of Policy Profiles which are available and assigned.

Returns: set of str of Policy Profile names

get_unassigned_policy_profiles()
Return a set of Policy Profiles which are available but not assigned.

Returns: set of str of Policy Profile names

load_all_provider_images()

load_all_provider_instances()

load_all_provider_templates()
Loads the list of images that are available under the provider.

If it could click through the link in infoblock, returns True. If it sees that the number of images is 0, it returns False.

load_all_provider_vms()
Loads the list of instances that are running under the provider.

If it could click through the link in infoblock, returns True. If it sees that the number of instances is 0, it returns False.

num_template(*db=True*)
Returns the providers number of templates, as shown on the Details page.

num_vm(*db=True*)
Returns the providers number of instances, as shown on the Details page.

wait_for_creds_ok()
Waits for provider's credentials to become O.K. (circumvents the summary rails exc.)

cfme.common.provider.**cleanup_vm**(*vm_name*, *provider*)

cfme.common.vm module

Module contents

class cfme.common.**PolicyProfileAssignable**
Bases: object

This class can be inherited by anything that provider load_details method.

It provides functionality to assign and unassign Policy Profiles

assign_policy_profiles(**policy_profile_names*)
Assign Policy Profiles to this object.

Parameters **policy_profile_names** – str with Policy Profile names. After Control/Explorer coverage goes in, PolicyProfile objects will be also passable.

assigned_policy_profiles

manage_policies_tree = <cfme.web_ui.CheckboxTree locator={Version ('lowest'): "//div[@id='treebox']/div/table",

unassign_policy_profiles(**policy_profile_names*)
Unssign Policy Profiles to this object.

Parameters **policy_profile_names** – str with Policy Profile names. After Control/Explorer coverage goes in, PolicyProfile objects will be also passable.

class cfme.common.**Taggable**
Bases: object

This class can be inherited by anything that provider load_details method.

It provides functionality to assign and unassign tags.

add_tag(*tag*, *single_value=False*)

get_tags(*tag='My Company Tags'*)

remove_tag(*tag*)

cfme.configure package

Subpackages

cfme.configure.configuration package

Submodules

cfme.configure.configuration.candu module

cfme.configure.configuration.candu.**disable_all**()
Enable all C&U metric collection for this region

cfme.configure.configuration.candu.**enable_all**()
Enable all C&U metric collection for this region

Module contents

class cfme.configure.configuration.**AmazonAuthSetting**(*access_key*, *secret_key*, *get_groups=False*, *timeout_h=None*, *timeout_m=None*)

Bases: cfme.configure.configuration.AuthSetting

Authentication settings via Amazon.

Parameters

- **access_key** – Amazon access key
- **secret_key** – Amazon secret key
- **get_groups** – Whether to get groups from the auth provider (default *False*)
- **timeout_h** – Timeout in hours
- **timeout_m** – Timeout in minutes

Usage:

```
amiauth = AmazonAuthSetting("AJSHDGVJAG", "IUBDIUWQBQW")
amiauth.update()
```

form = <cfme.web_ui.Form fields=[('timeout_h', Select('select#session_timeout_hours', multi=False)), ('timeout_m', Sele

pretty_attrs = ['access_key', 'secret_key', 'get_groups', 'timeout_h', 'timeout_m']

update(*updates=None*)

class cfme.configure.configuration.**AnalysisProfile**(*name*, *description*, *files=None*, *events=None*, *categories=None*, *registry=None*)

Bases: utils.pretty.Pretty, utils.update.Updateable

Analysis profiles. Do not use this class but the derived one.

Example

```
p = VMAnalysisProfile(name, description)
p.files = [
    "/somefile",
    {"Name": "/some/anotherfile", "Collect Contents?": True}
]
p.categories = ["check_system"]
p.create()
p.delete()
```

CREATE_LOC = None

copy(*name=None*)

create()

delete()

exists

form = <cfme.web_ui.tabstrip.TabStripForm fields=[('name', 'input#name'), ('description', 'input#description'), ('catego

pretty_attrs = ['name', 'description', 'files', 'events']

update(*updates=None*)

class cfme.configure.configuration.**AuthSetting**
Bases: utils.update.Updateable, utils.pretty.Pretty

form = <cfme.web_ui.Form fields=[('timeout_h', Select('select#session_timeout_hours', multi=False)), ('timeout_m', Sele

classmethod set_session_timeout(*hours=None*, *minutes=None*)
Sets the session timeout of the appliance.

class cfme.configure.configuration.**BasicInformation**(*company_name=None*, *appliance_name=None*, *appliance_zone=None*, *time_zone=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

This class represents the "Basic Info" section of the Configuration page.

Parameters

- **company_name** – Company name.
- **appliance_name** – Appliance name.
- **appliance_zone** – Appliance Zone.
- **time_zone** – Time Zone.

Usage:

```
basic_info = BasicInformation(company_name="ACME Inc.")
basic_info.update()
```

basic_information = <cfme.web_ui.Form fields=[('company_name', <cfme.web_ui.Input _names=('server_company

pretty_attrs = ['company_name', 'appliance_name', 'appliance_zone', 'time_zone']

update()
Navigate to a correct page, change details and save.

class cfme.configure.configuration.**Category**(*name=None*, *display_name=None*, *description=None*, *show_in_console=True*, *single_value=True*, *capture_candu=False*)
Bases: utils.pretty.Pretty

create(*cancel=False*)

delete(*cancel=True*)

pretty_attrs = ['name', 'display_name', 'description', 'show_in_console', 'single_value', 'capture_candu']

update(*updates*, *cancel=False*)

class cfme.configure.configuration.**DatabaseAuthSetting**(*timeout_h=None*, *timeout_m=None*)
Bases: cfme.configure.configuration.AuthSetting

Authentication settings for DB internal database.

Parameters

- **timeout_h** – Timeout in hours
- **timeout_m** – Timeout in minutes

Usage:

```
dbauth = DatabaseAuthSetting()
dbauth.update()
```

form = <cfme.web_ui.Form fields=[('timeout_h', Select('select#session_timeout_hours', multi=False)), ('timeout_m', Sele

pretty_attrs = ['timeout_h', 'timeout_m']

update(*updates=None*)

class cfme.configure.configuration.**DatabaseBackupSchedule**(*name*, *description*, *active=True*, *protocol=None*, *depot_name=None*, *uri=None*, *username=None*, *password=None*, *password_verify=None*, *run_type='Once'*, *run_every=None*, *time_zone=None*, *start_date=None*, *start_hour=None*, *start_min=None*)

Bases: cfme.configure.configuration.Schedule

Configure/Configuration/Region/Schedules - Database Backup type

Parameters

- **name** – Schedule name
- **description** – Schedule description
- **active** – Whether the schedule should be active (default *True*)
- **protocol** – One of `{'Samba', 'Network File System'}`
- **run_type** – Once, Hourly, Daily, ...
- **run_every** – If *run_type* is not Once, then you can specify how often it should be run
- **time_zone** – Time zone selection
- **start_date** – Specify start date (mm/dd/yyyy or datetime.datetime())
- **start_hour** – Starting hour
- **start_min** – Starting minute

Usage:

```
smb_schedule = DatabaseBackupSchedule(
    name="Bi-hourly Samba Database Backup",
    description="Everybody's favorite backup schedule",
    protocol="Samba",
    uri="samba.example.com/share_name",
    username="samba_user",
    password="secret",
    password_verify="secret",
    time_zone="UTC",
    start_date=datetime.datetime.utcnow(),
    run_type="Hourly",
    run_every="2 Hours"
)
smb_schedule.create()
smb_schedule.delete()

... or ...

nfs_schedule = DatabaseBackupSchedule(
    name="One-time NFS Database Backup",
    description="The other backup schedule",
    protocol="Network File System",
    uri="nfs.example.com/path/to/share",
    time_zone="Chihuahua",
    start_date="21/6/2014",
    start_hour="7",
    start_min="45"
)
nfs_schedule.create()
nfs_schedule.delete()
```

create(*cancel=False*, *samba_validate=False*)

Create a new schedule from the informations stored in the object.

Parameters

- **cancel** – Whether to click on the cancel button to interrupt the creation.
- **samba_validate** – Samba-only option to click the *Validate* button to check if entered samba credentials are valid or not

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('description', <cfm

update(*updates*, *cancel=False*, *samba_validate=False*)

Modify an existing schedule with informations from this instance.

Parameters

- **updates** – Dict with fields to be updated
- **cancel** – Whether to click on the cancel button to interrupt the editation.
- **samba_validate** – Samba-only option to click the *Validate* button to check if entered samba credentials are valid or not

class cfme.configure.configuration.**ExternalAuthSetting**(*get_groups=False*, *timeout_h='1'*, *timeout_m='0'*)

Bases: cfme.configure.configuration.AuthSetting

Authentication settings for authentication via httpd.

Parameters

- **timeout_h** – Timeout in hours
- **timeout_m** – Timeout in minutes
- **get_groups** – Get user groups from external auth source.

Usage:

```
dbauth = ExternalAuthSetting(get_groups=True)
dbauth.update()
```

form = <cfme.web_ui.Form fields=[('timeout_h', Select('select#session_timeout_hours', multi=False)), ('timeout_m', Sele

pretty_attrs = ['timeout_h', 'timeout_m', 'get_groups']

setup()

update(*updates=None*)

class cfme.configure.configuration.**HostAnalysisProfile**(*name*, *description*, *files=None*, *events=None*, *categories=None*, *registry=None*)

Bases: cfme.configure.configuration.AnalysisProfile

CREATE_LOC = 'host_analysis_profile_add'

class cfme.configure.configuration.**LDAPAuthSetting**(*hosts*, *user_type*, *user_suffix*, *base_dn=None*, *bind_dn=None*, *bind_password=None*, *get_groups=False*, *get_roles=False*, *follow_referrals=False*, *port=None*, *timeout_h=None*, *timeout_m=None*)

Bases: cfme.configure.configuration.AuthSetting

Authentication via LDAP

Parameters

- **hosts** – List of LDAP servers (max 3).
- **user_type** – "userprincipalname", "mail", ...
- **user_suffix** – User suffix.
- **base_dn** – Base DN.
- **bind_dn** – Bind DN.
- **bind_password** – Bind Password.
- **get_groups** – Get user groups from LDAP.
- **get_roles** – Get roles from home forest.
- **follow_referrals** – Follow Referrals.
- **port** – LDAP connection port.
- **timeout_h** – Timeout in hours
- **timeout_m** – Timeout in minutes

Usage:

```
ldapauth = LDAPAuthSetting(
    ["host1", "host2"],
    "mail",
    "user.acme.com"
)
ldapauth.update()
```

AUTH_MODE = 'LDAP'

form = <cfme.web_ui.Form fields=[('timeout_h', Select('select#session_timeout_hours', multi=False)), ('timeout_m', Sele

pretty_attrs = ['hosts', 'user_type', 'user_suffix', 'base_dn', 'bind_dn', 'bind_password']

update(*updates=None*)

class cfme.configure.configuration.**LDAPSAuthSetting**(*hosts*, *user_type*, *user_suffix*, *base_dn=None*, *bind_dn=None*, *bind_password=None*, *get_groups=False*, *get_roles=False*, *follow_referrals=False*, *port=None*, *timeout_h=None*, *timeout_m=None*)

Bases: cfme.configure.configuration.LDAPAuthSetting

Authentication via LDAPS

Parameters

- **hosts** – List of LDAPS servers (max 3).
- **user_type** – "userprincipalname", "mail", ...
- **user_suffix** – User suffix.
- **base_dn** – Base DN.
- **bind_dn** – Bind DN.
- **bind_password** – Bind Password.
- **get_groups** – Get user groups from LDAP.
- **get_roles** – Get roles from home forest.
- **follow_referrals** – Follow Referrals.
- **port** – LDAPS connection port.
- **timeout_h** – Timeout in hours
- **timeout_m** – Timeout in minutes

Usage:

```
ldapauth = LDAPSAuthSetting(
    ["host1", "host2"],
    "mail",
    "user.acme.com"
)
ldapauth.update()
```

AUTH_MODE = 'LDAPS'

class cfme.configure.configuration.**SMTPSettings**(*host=None*, *port=None*, *domain=None*, *start_tls=None*, *ssl_verify=None*, *auth=None*, *username=None*, *password=None*, *from_email=None*, *test_email=None*)

Bases: utils.update.Updateable

SMTP settings on the main page.

Parameters

- **host** – SMTP Server host name
- **port** – SMTP Server port
- **domain** – E-mail domain
- **start_tls** – Whether use StartTLS
- **ssl_verify** – SSL Verification
- **auth** – Authentication type
- **username** – User name
- **password** – User password
- **from_email** – E-mail address to be used as the "From:"
- **test_email** – Destination of the test-email.

Usage:

```
smtp = SMTPSettings(
    host="smtp.acme.com",
    start_tls=True,
    auth="login",
    username="mailer",
    password="secret"
)
smtp.update()
```

buttons = <cfme.web_ui.Region title=None>

classmethod send_test_email(*to_address*)

Send a testing e-mail on specified address. Needs configured SMTP.

Parameters **to_address** – Destination address.

smtp_settings = <cfme.web_ui.Form fields=[('host', <cfme.web_ui.Input _names=('smtp_host',), _use_id=False>), ('p

update()

class cfme.configure.configuration.**Schedule**(*name*, *description*, *active=True*, *action=None*, *filter_type=None*, *filter_value=None*, *run_type='Once'*, *run_every=None*, *time_zone=None*, *start_date=None*, *start_hour=None*, *start_min=None*)

Bases: utils.pretty.Pretty

Configure/Configuration/Region/Schedules functionality

CReate, Update, Delete functionality. Todo: Maybe the row handling might go into Table class?

Parameters

- **name** – Schedule's name.
- **description** – Schedule description.
- **active** – Whether the schedule should be active (default *True*)
- **action** – Action type
- **filter_type** – Filtering type
- **filter_value** – If a more specific *filter_type* is selected, here is the place to choose hostnames, machines and so ...
- **run_type** – Once, Hourly, Daily, ...
- **run_every** – If *run_type* is not Once, then you can specify how often it should be run.
- **time_zone** – Time zone selection.
- **start_date** – Specify start date (mm/dd/yyyy or datetime.datetime()).
- **start_hour** – Starting hour
- **start_min** – Starting minute.

Usage:

```
schedule = Schedule(
    "My very schedule",
    "Some description here.",
    action="Datastore Analysis",
    filter_type="All Datastores for Host",
    filter_value="datastore.intra.acme.com",
    run_type="Hourly",
    run_every="2 Hours"
)
schedule.create()
schedule.disable()
schedule.enable()
schedule.delete()
# Or
Schedule.enable_by_names("One schedule", "Other schedule")
# And so.
```

create(*cancel=False*)

Create a new schedule from the informations stored in the object.

Parameters **cancel** – Whether to click on the cancel button to interrupt the creation.

delete(*cancel=False*)

Delete the schedule represented by this object.

Calls the class method with the name of the schedule taken out from the object.

Parameters **cancel** – Whether to click on the cancel button in the pop-up.

classmethod delete_by_name(*name*, *cancel=False*)

Finds a particular schedule by its name and then deletes it.

Parameters

- **name** – Name of the schedule.
- **cancel** – Whether to click on the cancel button in the pop-up.

disable()
Enable the schedule via table checkbox and Configuration menu.

classmethod disable_by_names(**names*)
Checks all schedules that are passed with *names* and then disables them via menu.

Parameters ***names** – Names of schedules to disable.

enable()
Enable the schedule via table checkbox and Configuration menu.

classmethod enable_by_names(**names*)
Checks all schedules that are passed with *names* and then enables them via menu.

Parameters ***names** – Names of schedules to enable.

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('description', <cfm

pretty_attrs = ['name', 'description', 'run_type', 'run_every', 'start_date', 'start_hour', 'start_min']

classmethod select_by_names(**names*)
Select all checkboxes at the schedules with specified names.

Can select multiple of them.

Candidate for DRY in Table class.

Parameters ***names** – Arguments with all schedules' names.

tab = {'Monthly': 'timer_months', 'Hourly': 'timer_hours', 'Daily': 'timer_days', 'Weekly': 'timer_weeks'}

update(*updates*, *cancel=False*)
Modify an existing schedule with informations from this instance.

Parameters

- **updates** – Dict with fields to be updated
- **cancel** – Whether to click on the cancel button to interrupt the editation.

class cfme.configure.configuration.**ServerLogDepot**
Bases: utils.pretty.Pretty

This class represents the 'Collect logs' for the server.

Usage:

```
log_credentials = ServerLogDepot.Credentials("nfs", "backup.acme.com")
log_credentials.update()
ServerLogDepot.collect_all()
ServerLogDepot.Credentials.clear()
```

class Credentials(*p_type*, *name*, *uri*, *username=None*, *password=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

This class represents the credentials for log depots.

Parameters

- **p_type** – One of ftp, nfs, or smb.

- **uri** – Hostname/IP address of the machine.
- **username** – User name used for logging in (ftp, smb only).
- **password** – Password used for logging in (ftp, smb only).

Usage:

```
log_credentials = ServerLogDepot.Credentials("nfs", "backup.acme.com")
log_credentials.update()
log_credentials = ServerLogDepot.Credentials(
    "smb",
    "foobar",
    "backup.acme.com",
    username="jdoe",
    password="xyz",
)
log_credentials.update()
```

classmethod clear(*cancel=False*)
Navigate to correct page and set <No Depot>.
Parameters **cancel** – If set to True, the Cancel button is clicked instead of saving.

p_types

pretty_attrs = ['p_type', 'name', 'uri', 'username', 'password']

save_button = {Version ('lowest'): FormButton('Save Changes'), '5.4': FormButton('Save changes')}

server_collect_logs = <cfme.web_ui.Form fields=[('type', Select('select#log_protocol', multi=False)), ('name',

update(*validate=True*, *cancel=False*)
Navigate to a correct page, change details and save.
Parameters
- **validate** – Whether validate the credentials (not for NFS)
- **cancel** – If set to True, the Cancel button is clicked instead of saving.

validate = FormButton('Validate the credentials by logging into the Server')

classmethod ServerLogDepot.**collect_all**()
Initiate and wait for collection of all logs to finish.

classmethod ServerLogDepot.**collect_current**()
Initiate and wait for collection of the current log to finish.

ServerLogDepot.**elements = <cfme.web_ui.Region title=None>**

classmethod ServerLogDepot.**get_last_collection**()
Returns the Last Log Collection that is displayed in the InfoBlock.

Returns: If it is Never, returns *None*, otherwise utils.timeutil.parsetime.

classmethod ServerLogDepot.**get_last_message**()
Returns the Last Message that is displayed in the InfoBlock.

class cfme.configure.configuration.**Tag**(*name=None*, *display_name=None*, *category=None*)
Bases: utils.pretty.Pretty

create()

delete(*cancel=True*)

pretty_attrs = ['name', 'display_name', 'category']

update(*updates*)

class cfme.configure.configuration.**VMAnalysisProfile**(*name*, *description*, *files=None*, *events=None*, *categories=None*, *registry=None*)

Bases: cfme.configure.configuration.AnalysisProfile

CREATE_LOC = 'vm_analysis_profile_add'

class cfme.configure.configuration.**Zone**(*name=None*, *description=None*, *smartproxy_ip=None*, *ntp_server_1=None*, *ntp_server_2=None*, *ntp_server_3=None*, *max_scans=None*, *user=None*, *password=None*, *verify=None*)

Bases: utils.pretty.Pretty

Configure/Configuration/Region/Zones functionality

Create/Read/Update/Delete functionality.

create(*cancel=False*)

Create a new Zone from the information stored in the object.

Parameters **cancel** – Whether to click on the cancel button to interrupt the creation.

delete(*cancel=False*)

Delete the Zone represented by this object.

Parameters **cancel** – Whether to click on the cancel button in the pop-up.

exists

classmethod go_to_by_description(*description*)

Finds and navigates to a particular Zone by its description.

This method looks for a Zone with the provided description. If it finds one (and only one) Zone with that description, it navigates to it. Otherwise, it raises an Exception.

Parameters **description** – description of the Zone.

Raises ZoneNotFound – If no single Zone is found with the specified description.

pretty_attrs = ['name', 'description', 'smartproxy_ip', 'ntp_server_1', 'ntp_server_2', 'ntp_server_3', 'max_scans', '

update(*updates*, *cancel=False*)

Modify an existing zone with information from this instance.

Parameters

- **updates** – Dict with fields to be updated
- **cancel** – Whether to click on the cancel button to interrupt the edit.

cfme.configure.configuration.**add_tag**(*cat_name*)

cfme.configure.configuration.**edit_tag**(*cat_name*, *tag_name*)

cfme.configure.configuration.**get_ntp_servers**()

cfme.configure.configuration.**get_replication_backlog**(*navigate=True*)

Gets replication backlog from Configure / Configuration pages.

Returns: int representing the remaining items in the replication backlog.

cfme.configure.configuration.**get_replication_status**(*navigate=True*)

Gets replication status from Configure / Configuration pages.

Returns: bool of whether replication is Active or Inactive.

cfme.configure.configuration.**get_server_roles**(*navigate=True*, *db=True*)
Get server roles from Configure / Configuration

Returns: `dict` with the roles in the same format as `set_server_roles()` accepts as kwargs.

cfme.configure.configuration.**get_workers_list**(*do_not_navigate=False*, *refresh=True*)
Retrieves all workers.

Returns a dictionary where keys are names of the workers and values are lists (because worker can have multiple instances) which contain dictionaries with some columns.

cfme.configure.configuration.**restart_workers**(*name*, *wait_time_min=1*)
Restarts workers by their name.

Parameters **name** – Name of the worker. Multiple workers can have the same name. Name is matched with *in*

Returns: bool whether the restart succeeded.

cfme.configure.configuration.**server_id**()

cfme.configure.configuration.**server_name**()

cfme.configure.configuration.**server_region**()

cfme.configure.configuration.**server_region_pair**()

cfme.configure.configuration.**server_roles_disabled**(**roles*)

cfme.configure.configuration.**server_roles_enabled**(**roles*)

cfme.configure.configuration.**server_zone_description**()

cfme.configure.configuration.**set_auth_mode**(*mode*, ***kwargs*)
Set up authentication mode

Parameters

- **mode** – Authentication mode to set up.
- **kwargs** – A dict of keyword arguments used to initialize one of the *AuthSetting classes - class type is mode-dependent.

Raises `AuthModeUnknown` – when the given mode is not valid

cfme.configure.configuration.**set_database_external_appliance**(*hostname*)
Set the database as an external from another appliance

Parameters **hostname** – Host name of the another appliance

cfme.configure.configuration.**set_database_external_postgres**(*hostname*, *database*, *username*, *password*)
Set the database as an external Postgres DB

Parameters

- **hostname** – Host name of the Postgres server
- **database** – Database name
- **username** – User name
- **password** – User password

cfme.configure.configuration.**set_database_internal**()
Set the database as the internal one.

cfme.configure.configuration.**set_ntp_servers**(**servers*)
Set NTP servers on Configure / Configuration pages.

Parameters ***servers** – Maximum of 3 hostnames.

cfme.configure.configuration.**set_replication_worker_host**(*host*, *port='5432'*)
Set replication worker host on Configure / Configuration pages.

Parameters **host** – Address of the hostname to replicate to.

cfme.configure.configuration.**set_server_roles**(***roles*)
Set server roles on Configure / Configuration pages.

Parameters ****roles** – Roles specified as in server_roles Form in this module. Set to True or False

Submodules

cfme.configure.about module

cfme.configure.access_control module

class cfme.configure.access_control.**Group**(*description=None*, *role=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

create()

delete()

edit_tags(*tag*, *value*)

group_form = <cfme.web_ui.Form fields=[('description_txt', <cfme.web_ui.Input _names=('description',), _use_id=Fals

pretty_attrs = ['description', 'role']

remove_tag(*tag*, *value*)

update(*updates*)

class cfme.configure.access_control.**Role**(*name=None*, *vm_restriction=None*, *product_features=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

copy(*name=None*)

create()

delete()

form = <cfme.web_ui.Form fields=[('name_txt', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('vm_restrictio

pretty_attrs = ['name', 'product_features']

update(*updates*)

class cfme.configure.access_control.**User**(*name=None*, *credential=None*, *email=None*, *group=None*, *cost_center=None*, *value_assign=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

copy()

create()

delete()

description

edit_tags(*tag*, *value*)

pretty_attrs = ['name', 'group']

remove_tag(*tag*, *value*)

update(*updates*)

user_form = <cfme.web_ui.Form fields=[('name_txt', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('userid

cfme.configure.access_control.**ac_tree**(**path*)
DRY function to access the shared level of the accordion tree.

Parameters ***path** – Path to click in the tree that follows the '[cfme] region xyz' node

cfme.configure.access_control.**get_group_order**()

cfme.configure.access_control.**server_region**()

cfme.configure.access_control.**server_region_pair**()

cfme.configure.access_control.**set_group_order**(*items*)

cfme.configure.access_control.**simple_user**(*userid*, *password*)

cfme.configure.red_hat_updates module

cfme.configure.settings module

Module dealing with Configure/My Setting section.

class cfme.configure.settings.**DefaultFilter**(*name=None*, *filters=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

filter_form = <cfme.web_ui.Form fields=[('filter_tree', {Version ('lowest'): <cfme.web_ui.CheckboxTree locator="//di

pretty_attrs = ['name', 'filters']

update(*updates*, *expect_success=True*)

class cfme.configure.settings.**Timeprofile**(*description=None*, *scope=None*, *days=None*, *hours=None*, *timezone=None*)
Bases: utils.update.Updateable

copy()

create()

delete()

save_button = FormButton('Add this Time Profile')

timeprofile_form = <cfme.web_ui.Form fields=[('description', <cfme.web_ui.Input _names=('description',), _use_id=

update(*updates*)

class cfme.configure.settings.**Visual**
Bases: utils.update.Updateable

check_image_exists()

cloud_provider_quad

datastore_quad

display_form = <cfme.web_ui.Form fields=[('chart_theme', {Version ('lowest'): Select('//select[@id="display_reportth

grid_view_limit

host_quad

infra_provider_quad

item_form = <cfme.web_ui.Form fields=[('grid_view', {Version ('lowest'): Select('//select[@id="perpage_grid"]', multi=

list_view_limit

login_page

pretty_attrs = ['name']

quadicons_form = <cfme.web_ui.Form fields=[('infra_provider_quad', <cfme.web_ui.Input _names=('quadicons_ems

report_view_limit

save_button = FormButton('Add this Time Profile')

startpage_form = <cfme.web_ui.Form fields=[('login_page', {Version ('lowest'): Select('//select[@id="start_page"]', r

template_quad

tile_view_limit

timezone

vm_quad

cfme.configure.tasks module

Module contents

cfme.control package

Submodules

cfme.control.explorer module

cfme.control.import_export module

cfme.control.import_export.**import_file**(*filename*, *cancel=False*)

Go to Control / Import Export and import given file.

Parameters

- **filename** – Full path to file to import.
- **cancel** – Whether to click Cancel instead of commit.

cfme.control.import_export.**is_imported**(*policy_profile*)

cfme.control.snmp_form module This file contains useful classes for working with SNMP filling.

class cfme.control.snmp_form.**SNMPForm**

Bases: object

Class encapsulating the most common (and hopefully single) configuration of SNMP form

Usage:

```
form = SNMPForm()
fill(form, dict(
    hosts=["host1", "host2"],
    traps=[
        ("aaa", "Counter32", 125),                      # Takes 3-tuples
        ("bbb", "Null"),                                # 2-tuples with no value specified
        SNMPTrap("ccc", "Gauge32", 256),                # objects dtto
        SNMPTrap("ddd", "Null"),                        # value can be unspecified too
        {"oid": "eee", "type": "Integer", "value": 42}  # omg dict too! Yay.
    ],
    version="v2",
    id="aabcd",
))
```

fields = <cfme.web_ui.Form fields=[('hosts', <cfme.control.snmp_form.SNMPHostsField object at 0x7f52eed25a10>), (

class cfme.control.snmp_form.**SNMPHostsField**
Bases: object

Class designed for handling the two-type snmp hosts field.

They can be 3 or just single.

host_fields
Returns list of locators to all host fields

class cfme.control.snmp_form.**SNMPTrap**(*oid*, *type*, *value=None*)
Bases: utils.pretty.Pretty

Nicer representation of the single SNMP trap

Parameters

- **oid** – SNMP OID
- **type** – SNMP type
- **value** – Value (default: *None*)

as_tuple
Return the contents as a tuple used for filling

pretty_attrs = ['oid', 'type', 'value']

class cfme.control.snmp_form.**SNMPTrapField**(*seq_id*)
Bases: utils.pretty.Pretty

Class representing SNMP trap field consisting of 3 elements - oid, type and value

Parameters **seq_id** – Sequential id of the field. Usually in range 1-10

oid

oid_loc

pretty_attrs = ['seq_id']

type

type_loc

value

value_loc

class cfme.control.snmp_form.**SNMPTrapsField**(*num_fields*)
Bases: utils.pretty.Pretty

Encapsulates all trap fields to simplify form filling

Parameters **num_fields** – How many SNMPTrapField to generate

pretty_attrs = ['num_fields']

cfme.control.snmp_form.**fill_snmp_form**(*form*, *values*, *action*)
I wanted to use dict but that is overrided in web_ui that it disassembles dict to list of tuples :(

cfme.control.snmp_form.**fill_snmp_hosts_field_basestr**(*field*, *value*)

cfme.control.snmp_form.**fill_snmp_hosts_field_list**(*field*, *values*)

cfme.control.snmp_form.**fill_snmp_trap_field_dict**(*field*, *val*)

cfme.control.snmp_form.**fill_snmp_trap_field_trap**(*field*, *val*)

cfme.control.snmp_form.**fill_snmp_trap_field_tuple**(*field*, *val*)

cfme.control.snmp_form.**fill_snmp_traps_field_list**(*field*, *values*)

cfme.control.snmp_form.**fill_snmp_traps_field_single_trap**(*field*, *value*)

Module contents

cfme.fixtures package

Submodules

cfme.fixtures.configure_auth_mode module
cfme.fixtures.configure_auth_mode.**available_auth_modes**()
cfme.fixtures.configure_auth_mode.**configure_aws_iam_auth_mode**(*browser*, *available_auth_modes*)
Configure AWS IAM authentication mode

cfme.fixtures.configure_auth_mode.**configure_external_auth_ipa**(*request*)

cfme.fixtures.configure_auth_mode.**configure_external_auth_ipa_class**(*request*)

cfme.fixtures.configure_auth_mode.**configure_external_auth_ipa_module**(*request*)

cfme.fixtures.configure_auth_mode.**configure_ldap_auth_mode**(*browser*, *available_auth_modes*)
Configure LDAP authentication mode

cfme.fixtures.configure_auth_mode.**configure_openldap_auth_mode**(*browser*, *available_auth_modes*)
Configure LDAP authentication mode

cfme.fixtures.login module
cfme.fixtures.login.**logged_in**(*browser*)
Logs into the system as admin and then returns the browser object.

Parameters **browser** – Current browser object.

Yields: Browser object
cfme.fixtures.login.**recycle**()

cfme.fixtures.pytest_selenium module Provides a number of useful functions for integrating with selenium.

The aim is that no direct calls to selenium be made at all. One reason for this it to ensure that all function calls to selenium wait for the ajax response which is needed in CFME.

Members of this module are available in the the pytest.sel namespace, e.g.:

```
pytest.sel.click(locator)
```

var ajax_wait_js A Javascript function for ajax wait checking

var class_selector Regular expression to detect simple CSS locators

class cfme.fixtures.pytest_selenium.**ByText**(*text*)
Bases: utils.pretty.Pretty

pretty_attrs = ['text']

class cfme.fixtures.pytest_selenium.**ByValue**(*value*)
Bases: utils.pretty.Pretty

pretty_attrs = ['value']

class cfme.fixtures.pytest_selenium.**ContextWrapper**
Bases: dict

Dict that provides .attribute access + dumps all keys when not found.

cfme.fixtures.pytest_selenium.**ScreenShot**
alias of screenshot

class cfme.fixtures.pytest_selenium.**Select**(*loc*, *multi=False*, *none=None*)
Bases: selenium.webdriver.support.select.Select, utils.pretty.Pretty

A proxy class for the real selenium Select() object.

We differ in one important point, that we can instantiate the object without it being present on the page. The object is located at the beginning of each function call.

Can hadle patternfly `selectpicker` kind of select. It alters the behaviour slightly, it does not use `move_to_element()` and uses JavaScript more extensively.

Parameters **loc** – A locator.

Returns: A `cfme.web_ui.Select` object.

class Option
Bases: tuple

Option(text, value)

text
Alias for field number 0

value
Alias for field number 1

Select.**all_options**
Returns a list of tuples of all the options in the Select

Select.**all_selected_options**
Fast variant of the original all_selected_options.

Selenium's all_selected_options iterates over ALL of the options, this directly returns only those that are selected.

Select.**classes**

Select.**deselect_all**()
Fast variant of the original deselect_all.

Uses all_selected_options, mimics selenium's exception behaviour.

Select.**first_selected_option**
Fast variant of the original first_selected_option.

Uses all_selected_options, mimics selenium's exception behaviour.

Select.**first_selected_option_text**

Select.**get_value_by_text**(*text*)

Select.**is_patternfly**

Select.**locate**()
Guards against passing wrong locator (not resolving to a select).

Select.**none**

Select.**observer_wait**()

Select.**pretty_attrs = ['_loc', 'is_multiple']**

Select.**select_by_value**(*value*)

Select.**select_by_visible_text**(*text*)
Dump all of the options if the required option is not present.

cfme.fixtures.pytest_selenium.**ajax_timeout**(**args*, ***kwds*)
Change the AJAX timeout in this context. Useful when something takes a long time.

Parameters **seconds** – Numebr of seconnds to wait.

cfme.fixtures.pytest_selenium.**base_url**()
Returns the base url.

Returns: *base_url* from env config yaml

cfme.fixtures.pytest_selenium.**check**(*loc*)
Convenience function to check a checkbox

Parameters **loc** – The locator of the element

cfme.fixtures.pytest_selenium.**checkbox**(*loc*, *set_to=False*)
Checks or unchecks a given checkbox

Finds an element given by loc and checks it

Parameters

- **loc** – The locator of the element
- **value** – The value the checkbox should represent as a bool (or None to do nothing)

Returns: Previous state of the checkbox

cfme.fixtures.pytest_selenium.**classes**(*loc*)
Return a list of classes attached to the element.

cfme.fixtures.pytest_selenium.**click**(*loc*, *wait_ajax=True*, *no_custom_handler=False*)
Clicks on an element.

If the element implements *_custom_click_handler* the control will be given to it. Then the handler decides what to do (eg. do not click under some circumstances).

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple or an object implementing *_custom_click_handler* method.
- **wait_ajax** – Whether to wait for ajax call to finish. Default True but sometimes it's handy to not do that. (some toolbar clicks)
- **no_custom_handler** – To prevent recursion, the custom handler sets this to True.

cfme.fixtures.pytest_selenium.**click_fn**(**els*)
Returns a function which successively clicks on a series of elements.

Parameters **els** – An iterable of elements:

Returns: The click function

cfme.fixtures.pytest_selenium.**current_url**()
Returns the current_url of the page

Returns: A url.

cfme.fixtures.pytest_selenium.**deselect_by_text**(*select_element*, *txt*)
Works on a select element and deselects an option by the visible text.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **text** – The select element option's visible text.

cfme.fixtures.pytest_selenium.**deselect_by_value**(*select_element*, *val*)
Works on a select element and deselects an option by the value attribute.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **value** – The select element's option value.

cfme.fixtures.pytest_selenium.**detect_observed_field**(*loc*)
Detect observed fields; sleep if needed

Used after filling most form fields, this function will inspect the filled field for one of the known CFME observed field attribues, and if found, sleep long enough for the observed field's AJAX request to go out, and then block until no AJAX requests are in flight.

Observed fields occasionally declare their own wait interval before firing their AJAX request. If found, that interval will be used instead of the default.

cfme.fixtures.pytest_selenium.**double_click**(*loc*, *wait_ajax=True*)
Double-clicks on an element.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **wait_ajax** – Whether to wait for ajax call to finish. Default True but sometimes it's handy to not do that. (some toolbar clicks)

cfme.fixtures.pytest_selenium.**drag_and_drop**(*source_element*, *dest_element*)
Drag and Drop element.

Parameters

- **source_element** – A locator, expects either a string, WebElement, tuple.

- **dest_element** – A locator, expects either a string, WebElement, tuple.
- **wait_ajax** – Whether to wait for ajax call to finish. Default True but sometimes it's handy to not do that. (some toolbar clicks)

cfme.fixtures.pytest_selenium.**drag_and_drop_by_offset**(*source_element*, *x=0*, *y=0*)
Drag and Drop element by offset

Parameters

- **source_element** – A locator, expects either a string, WebElement, tuple.
- **x** – Distance in pixels on X axis to move it.
- **y** – Distance in pixels on Y axis to move it.

cfme.fixtures.pytest_selenium.**element**(*o*, ***kwargs*)
Convert o to a single matching WebElement.

Parameters **o** – An object to be converted to a matching web element, expected string, WebElement, tuple.

Keywords:

_no_deeper: Whether this call of the function can call for something that can retrieve elements too. Recursion protection.

Returns: A WebElement object

Raises `NoSuchElementException` – When element is not found on page

cfme.fixtures.pytest_selenium.**execute_script**(*script*, **args*, ***kwargs*)
Wrapper for execute_script() to not have to pull browser() from somewhere.

It also provides our library which is stored in data/lib.js file.

cfme.fixtures.pytest_selenium.**first_from**(**locs*, ***kwargs*)
Goes through locators and first valid element received is returned.

Useful for things that could be located different way

Parameters

- ***locs** – Locators to pass through
- ****kwargs** – Keyword arguments to pass to element()

Raises `NoSuchElementException` – When none of the locator could find the element.

Returns: `WebElement`

cfme.fixtures.pytest_selenium.**force_navigate**(*page_name*)
Given a page name, attempt to navigate to that page no matter what breaks.

Parameters **page_name** – Name a page from the current `ui_navigate.nav_tree` tree to navigate to.

cfme.fixtures.pytest_selenium.**get**(*url*)
Changes page to the specified URL

Parameters **url** – URL to navigate to.

cfme.fixtures.pytest_selenium.**get_attribute**(*loc*, *attr*)
Returns the value of the HTML attribute of the given locator.

Parameters

- **loc** – A locator, expects eithera string, WebElement, tuple.
- **attr** – An attribute name.

Returns: Text describing the attribute of the element.

cfme.fixtures.pytest_selenium.**get_rails_error**()
Get displayed rails error. If not present, return None

cfme.fixtures.pytest_selenium.**go_to**(*page_name*)
go_to task mark, used to ensure tests start on the named page, logged in as Administrator.

Parameters **page_name** – Name a page from the current `ui_navigate.nav_tree` tree to navigate to.

Usage:

```
@pytest.sel.go_to('page_name')
def test_something_on_page_name():
    # ...
```

cfme.fixtures.pytest_selenium.**go_to_fixture**(*fixtureconf*, *browser*)
"Private" implementation of go_to in fixture form.

Used by the `go_to()` decorator, this is the actual fixture that does the work set up by the go_to decorator. py.test fixtures themselves can't have underscores in their name, so we can't imply privacy with that convention.

Don't use this fixture directly, use the go_to decorator instead.

cfme.fixtures.pytest_selenium.**handle_alert**(*cancel=False*, *wait=30.0*, *squash=False*)
Handles an alert popup.

Parameters

- **cancel** – Whether or not to cancel the alert. Accepts the Alert (False) by default.
- **wait** – Time to wait for an alert to appear. Default 30 seconds, can be set to 0 to disable waiting.
- **squash** – Whether or not to squash errors during alert handling. Default False

Returns True if the alert was handled, False if exceptions were squashed, None if there was no alert.

No exceptions will be raised if `squash` is True.

Raises

- `utils.wait.TimedOutError` – If the alert popup does not appear
- `selenium.common.exceptions.NoAlertPresentException` – If no alert is present when accepting or dismissing the alert.

cfme.fixtures.pytest_selenium.**in_flight**()
Check remaining (running) ajax requests

The element visibility check is complex because lightbox_div invokes visibility of spinner_div although it is not visible.

Returns Dictionary of js-related keys and booleans as its values, depending on status. The keys are: `jquery, prototype, miq, spinner and document`. The values are: `True` if running, `False` otherwise.

cfme.fixtures.pytest_selenium.**is_displayed**(*loc*, *_deep=0*, ***kwargs*)
Checks if a particular locator is displayed

Parameters **loc** – A locator, expects either a string, WebElement, tuple.

Keywords: move_to: Uses `move_to_element()` instead of `element()`

Returns: `True` if element is displayed, `False` if not

Raises

- `NoSuchElementException` – If element is not found on page
- `CFMEExceptionOccured` – When there is a CFME rails exception on the page.

`cfme.fixtures.pytest_selenium.`**`is_displayed_text`**(*text*)
Checks if a particular text is displayed

Parameters **text** – A string.

Returns: A string containing the text

`cfme.fixtures.pytest_selenium.`**`move_to_element`**(*loc*, ***kwargs*)
Moves to an element.

Parameters **loc** – A locator, expects either a string, WebElement, tuple.

Returns: Returns the element it was moved to to enable chaining.

`cfme.fixtures.pytest_selenium.`**`move_to_fn`**(**els*)
Returns a function which successively moves through a series of elements.

Parameters **els** – An iterable of elements:

Returns: The move function

`cfme.fixtures.pytest_selenium.`**`multi_check`**(*locators*)
Mass-check and uncheck for checkboxes.

Parameters **locators** – `dict` or `list` or whatever iterable of tuples. Key is the locator, value bool with check status.

Returns: list of booleans indicating for each locator, whether any action was taken.

`cfme.fixtures.pytest_selenium.`**`on_cfme_page`**()
Check whether we are on a CFME page and not another or blank page

`cfme.fixtures.pytest_selenium.`**`raw_click`**(*loc*, *wait_ajax=True*)
Does raw selenium's .click() call on element. Circumvents mouse move.

Parameters

- **loc** – Locator to click on.
- **wait_ajax** – Whether to wait for ajax.

`cfme.fixtures.pytest_selenium.`**`refresh`**()
Refreshes the current browser window.

`cfme.fixtures.pytest_selenium.`**`select_by_text`**(*select_element*, *txt*)
Works on a select element and selects an option by the visible text.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **text** – The select element option's visible text.

Returns: previously selected text

cfme.fixtures.pytest_selenium.**select_by_value**(*select_element*, *val*)
Works on a select element and selects an option by the value attribute.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **value** – The select element's option value.

cfme.fixtures.pytest_selenium.**send_keys**(*loc*, *text*)
Sends the supplied keys to an element.

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **text** – The text to inject into the element.

cfme.fixtures.pytest_selenium.**set_angularjs_value**(*loc*, *value*)
Sets value of an element managed by angularjs

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **value** – Value to set.

cfme.fixtures.pytest_selenium.**set_attribute**(*loc*, *attr*, *value*)
Sets the attribute of an element.

This is usually not done, that's why it is not implemented in selenium. But sometimes ...

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **attr** – Attribute name.
- **value** – Value to set.

cfme.fixtures.pytest_selenium.**tag**(*loc*)
Returns the tag name of an element

Parameters **loc** – A locator, expects either a string, WebElement, tuple.

Returns: A string containing the tag element's name.

cfme.fixtures.pytest_selenium.**take_screenshot**()

cfme.fixtures.pytest_selenium.**text**(*loc*, ***kwargs*)
Returns the text of an element.

Parameters **loc** – A locator, expects eithera string, WebElement, tuple.

Returns: A string containing the text of the element.

cfme.fixtures.pytest_selenium.**text_sane**(*loc*, ***kwargs*)
Returns text decoded from UTF-8 and stripped

Parameters **loc** – A locator, expects eithera string, WebElement, tuple.

Returns: A string containing the text of the element, decoded and stripped.

cfme.fixtures.pytest_selenium.**title**()

cfme.fixtures.pytest_selenium.**uncheck**(*loc*)
Convenience function to uncheck a checkbox

Parameters **loc** – The locator of the element

cfme.fixtures.pytest_selenium.**unset_attribute**(*loc*, *attr*)
Removes an attribute of an element.

This is usually not done, that's why it is not implemented in selenium. But sometimes ...

Parameters

- **loc** – A locator, expects either a string, WebElement, tuple.
- **attr** – Attribute name.

cfme.fixtures.pytest_selenium.**value**(*loc*)
Returns the value of an input element.

Parameters **loc** – A locator, expects eithera string, WebElement, tuple.

Returns: A string containing the value of the input element.

cfme.fixtures.pytest_selenium.**wait_for_ajax**()
Waits until all ajax timers are complete, in other words, waits until there are no more pending ajax requests, page load should be finished completely.

Raises TimedOutError – when ajax did not load in time

cfme.fixtures.pytest_selenium.**wait_for_element**(**locs*, ***kwargs*)
Wrapper around wait_until, specific to an element.

Parameters **loc** – A locator, expects either a string, WebElement, tuple.

Keywords: all_elements: Whether to wait not for one, but all elements (Default False) timeout: How much time to wait

cfme.fixtures.pytest_selenium.**wait_until**(*f*, *msg='Webdriver wait timed out'*, *timeout=120.0*)
This used to be a wrapper around WebDriverWait from selenium.

Now it is just compatibility layer using utils.wait.wait_for()

cfme.fixtures.rdb module Rdb: Remote debugger

Given the following configuration in conf/rdb.yaml:

```
breakpoints:
  - subject: Brief explanation of a problem
    exceptions:
      - cfme.exceptions.ImportableExampleException
      - BuiltinException (e.g. ValueError)
    recipients:
      - user@example.com
```

Any time an exception listed in a breakpoint's "exceptions" list is raised in rdb_catch() context in the course of a test run, a remote debugger will be started on a random port, and the users listed in "recipients" will be emailed instructions to access the remote debugger via telnet.

The exceptions will be imported, so their fully-qualified importable path is required. Exceptions without a module path are assumed to be builtins.

An Rdb instance can be used just like a Pdb instance.

Additionally, a signal handler has been set up to allow for triggering Rdb during a test run. To invoke it, kill -USR1 a test-running process and Rdb will start up. No emails are sent when operating in this mode, so check the py.test console for the endpoint address.

By default, Rdb assumes that there is a working MTA available on localhost, but this can be configured in `conf['env']['smtp']['server']`.

Note: This is very insecure, and should be used as a last resort for debugging elusive failures.

class `cfme.fixtures.rdb.`**Rdb**(*prompt_msg=''*)
Bases: `pdb.Pdb`

Remote Debugger

When set_trace is called, it will open a socket on a random unprivileged port connected to a Pdb debugging session. This session can be accessed via telnet, and will end when "continue" is called in the Pdb session.

do_c(*arg*)

do_cont(*arg*)

do_continue(*arg*)

interaction(**args*, ***kwargs*)

set_trace(**args*, ***kwargs*)
Start a pdb debugger available via telnet, and optionally email people the endpoint

The endpoint will always be seen in the py.test runner output.

Keyword Arguments

- **recipients** – A list where, if set, an email will be sent to email addresses in this list.
- **subject** – If set, an optional custom email subject

`cfme.fixtures.rdb.`**pytest_internalerror**(*excrepr*, *excinfo*)

`cfme.fixtures.rdb.`**rdb_catch**(**args*, ***kwds*)
Context Manager used to wrap mysterious failures for remote debugging.

`cfme.fixtures.rdb.`**rdb_handle_signal**(*signal*, *frame*)

`cfme.fixtures.rdb.`**send_breakpoint_email**(*exctype*, *msg=''*)

cfme.fixtures.rest_api module Fixtures, providing an access to the CFME REST API.

See `rest_api()` and py:func:*rest_api_modscope*

`cfme.fixtures.rest_api.`**rest_api**()

`cfme.fixtures.rest_api.`**rest_api_modscope**()

cfme.fixtures.smtp module This module provides a fixture useful for checking the e-mails arrived.

Main use is of fixture `smtp_test()`, which is function scoped. There is also a `smtp_test_module()` fixture for which the smtp_test is just a function-scoped wrapper to speed things up. The base of all this is the session-scoped _smtp_test_session that keeps care about the collector.

`cfme.fixtures.smtp.`**pytest_runtest_call**(*item*)

`cfme.fixtures.smtp.`**smtp_test**(*request*)
Fixture, which prepares the appliance for e-mail capturing tests

Returns: `util.smtp_collector_client.SMTPCollectorClient` instance.

cfme.fixtures.storage module

cfme.fixtures.storage.**use_storage**(*uses_ssh*)

cfme.fixtures.tracer module

cfme.fixtures.tracer.**import_module**(*module_str*)

Use __import__ to import a module and then retrieve the imported submodule

cfme.fixtures.tracer.**load**()

cfme.fixtures.tracer.**pytest_addoption**(*parser*)

cfme.fixtures.tracer.**pytest_configure**(*config*)

cfme.fixtures.tracer.**pytest_runtest_call**(*item*)

hook to run each test with traced function calls

cfme.fixtures.tracer.**trace_on**(**args*, ***kwds*)

cfme.fixtures.vm_name module

cfme.fixtures.vm_name.**vm_name**(*provider*)

Module contents

A variety of modules intended to make life easier for QE developers.

- cfme.fixtures.login - A module providing a login generator
- cfme.fixtures.pytest_selenium - A module offering a large number of CFME optimized selenium wrappers and other auxilliary functions.

cfme.infrastructure package

Submodules

cfme.infrastructure.cluster module

cfme.infrastructure.config_management module

class cfme.infrastructure.config_management.**ConfigManager**(*name*, *url*, *ssl*, *credentials*, *key=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Configuration manager object (Foreman, RH Satellite)

Parameters

- **name** – Name of the config. manager
- **url** – URL, hostname or IP of the config. manager
- **ssl** – Boolean value; *True* if SSL certificate validity should be checked, *False* otherwise
- **credentials** – Credentials to access the config. manager
- **key** – Key to access the cfme_data yaml data (same as *name* if not specified)

Usage:

```
.. code-block:: python

    cfg_mgr = ConfigManager('my_foreman', 'my-foreman.example.com', False,
                        ConfigManager.Credential(principal='admin', secret='testing'))
    cfg_mgr.create()
```

class `Credential` (*principal=None*, *secret=None*, *verify_secret=None*, ***ignore*)
Bases: `cfme.Credential`, `utils.update.Updateable`

`ConfigManager.`**`config_profiles`**
Returns 'ConfigProfile' configuration profiles (hostgroups) available on this manager

`ConfigManager.`**`create`** (*cancel=False*, *validate_credentials=True*, *validate=True*, *force=False*)
Creates the manager through UI

Parameters

- **cancel** (*bool*) – Whether to cancel out of the creation. The cancel is done after all the information present in the manager has been filled in the UI.
- **validate_credentials** (*bool*) – Whether to validate credentials - if True and the credentials are invalid, an error will be raised.
- **validate** (*bool*) – Whether we want to wait for the manager's data to load and show up in it's detail page. True will also wait, False will only set it up.
- **force** (*bool*) – Whether to force the creation even if the manager already exists. True will try anyway; False will check for its existence and leave, if present.

`ConfigManager.`**`delete`** (*cancel=False*, *wait_deleted=True*, *force=False*)
Deletes the manager through UI

Parameters

- **cancel** (*bool*) – Whether to cancel out of the deletion, when the alert pops up.
- **wait_deleted** (*bool*) – Whether we want to wait for the manager to disappear from the UI. True will wait; False will only delete it and move on.
- **force** (*bool*) – Whether to try to delete the manager even though it doesn't exist. True will try to delete it anyway; False will check for its existence and leave, if not present.

`ConfigManager.`**`exists`**
Returns whether the manager exists in the UI or not

classmethod `ConfigManager.`**`load_from_yaml`** (*key*)
Returns 'ConfigManager' object loaded from yamls, based on its key

`ConfigManager.`**`navigate`**`()`
Navigates to the manager's detail page

`ConfigManager.`**`pretty_attr = ['name', 'url']`**

`ConfigManager.`**`refresh_relationships`** (*cancel=False*)
Refreshes relationships and power states of this manager

`ConfigManager.`**`systems`**
Returns 'ConfigSystem' configured systems (hosts) available on this manager

`ConfigManager.`**`type`**
Returns presumed type of the manager based on CFME version

Note: We cannot actually know the type of the provider from the UI. This represents the supported type by CFME version and is to be used in navigation.

This attribute is lazily evaluated and cached.

ConfigManager.**update**(*updates*, *cancel=False*, *validate_credentials=False*)
Updates the manager through UI

Parameters

- **updates** (*dict*) – Data to change.
- **cancel** (*bool*) – Whether to cancel out of the update. The cancel is done after all the new information has been filled in the UI.
- **validate_credentials** (*bool*) – Whether to validate credentials - if True and the credentials are invalid, an error will be raised.

Note: utils.update use is recommended over use of this method.

ConfigManager.**yaml_data**
Returns yaml data for this manager

class cfme.infrastructure.config_management.**ConfigProfile**(*name*, *manager*)
Bases: utils.pretty.Pretty

Configuration profile object (foreman-side hostgroup)

Parameters

- **name** – Name of the profile
- **manager** – ConfigManager object which this profile is bound to

navigate()
Navigates to the profile's detail page

pretty_attr = ['name', 'manager']

systems
Returns 'ConfigSystem' objects that are active under this profile

class cfme.infrastructure.config_management.**ConfigSystem**(*name*, *profile*)
Bases: utils.pretty.Pretty

navigate()
Navigates to the system's detail page

pretty_attr = ['name', 'manager_key']

tag(*tag*)
Tags the system by given tag

tags
Returns a list of this system's active tags

untag(*tag*)
Removes the selected tag off the system

cfme.infrastructure.datastore module

cfme.infrastructure.host module A model of an Infrastructure Host in CFME

var page A `cfme.web_ui.Region` object describing common elements on the Providers pages.

var properties_form A `cfme.web_ui.Form` object describing the main add form.

var credentials_form A `cfme.web_ui.Form` object describing the credentials form.

class `cfme.infrastructure.host.`**`Host`**(*name=None*, *hostname=None*, *ip_address=None*, *custom_ident=None*, *host_platform=None*, *ipmi_address=None*, *mac_address=None*, *credentials=None*, *ipmi_credentials=None*, *interface_type='lan'*)

Bases: `utils.update.Updateable`, `utils.pretty.Pretty`

Model of an infrastructure host in cfme.

Parameters

- **name** – Name of the host.
- **hostname** – Hostname of the host.
- **ip_address** – The IP address as a string.
- **custom_ident** – The custom identifiter.
- **host_platform** – Included but appears unused in CFME at the moment.
- **ipmi_address** – The IPMI address.
- **mac_address** – The mac address of the system.
- **credentials** (*Credential*) – see Credential inner class.
- **ipmi_credentials** (*Credential*) – see Credential inner class.

Usage:

```
myhost = Host(name='vmware',
              credentials=Provider.Credential(principal='admin', secret='foobar'))
myhost.create()
```

class **`Credential`**(***kwargs*)

Bases: `cfme.Credential`, `utils.update.Updateable`

Provider credentials

Parameters ****kwargs** – If using IPMI type credential, ipmi = True

`Host.`**`assign_policy_profiles`**(**policy_profile_names*)

Assign Policy Profiles to this Host.

Parameters **policy_profile_names** – `str` with Policy Profile names. After Control/Explorer coverage goes in, PolicyProfile objects will be also passable.

`Host.`**`create`**(*cancel=False*, *validate_credentials=False*)

Creates a host in the UI

Parameters

- **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the Host has been filled in the UI.
- **validate_credentials** (*boolean*) – Whether to validate credentials - if True and the credentials are invalid, an error will be raised.

Host.**delete**(*cancel=True*)
Deletes a host from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

Host.**equal_drift_results**(*row_text*, *section*, **indexes*)
Compares drift analysis results of a row specified by it's title text

Parameters

- **row_text** – Title text of the row to compare
- **section** – Accordion section where the change happened; this section must be activated
- **indexes** – Indexes of results to compare starting with 0 for first row (latest result). Compares all available drifts, if left empty (default).

Note: There have to be at least 2 drift results available for this to work.

Returns `True` if equal, `False` otherwise.

Host.**execute_button**(*button_group*, *button*, *cancel=True*)

Host.**exists**

Host.**get_datastores**()
Gets list of all datastores used by this host

Host.**get_db_id**

Host.**get_detail**(**ident*)
Gets details from the details infoblock

The function first ensures that we are on the detail page for the specific host.

Parameters ***ident** – An InfoBlock title, followed by the Key name, e.g. "Relationships", "Images"

Returns: A string representing the contents of the InfoBlock's value.

Host.**get_ipmi**()

Host.**has_valid_credentials**
Check if host has valid credentials saved

Returns: `True` if credentials are saved and valid; `False` otherwise

Host.**power_off**()

Host.**power_on**()

Host.**pretty_attrs = ['name', 'hostname', 'ip_address', 'custom_ident']**

Host.**run_smartstate_analysis**()
Runs smartstate analysis on this host

Note: The host must have valid credentials already set up for this to work.

Host.**tag**(*tag*, ***kwargs*)
Tags the system by given tag

Host.**unassign_policy_profiles**(**policy_profile_names*)
Unssign Policy Profiles to this Host.

Parameters **policy_profile_names** – `str` with Policy Profile names. After Control/Explorer coverage goes in, PolicyProfile objects will be also passable.

`Host.`**`untag`**(*tag*)
Removes the selected tag off the system

`Host.`**`update`**(*updates*, *cancel=False*, *validate_credentials=False*)
Updates a host in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

`cfme.infrastructure.host.`**`find_quadicon`**(*host*, *do_not_navigate=False*)
Find and return a quadicon belonging to a specific host

Parameters **host** – Host name as displayed at the quadicon

Returns: `cfme.web_ui.Quadicon` instance

`cfme.infrastructure.host.`**`get_all_hosts`**(*do_not_navigate=False*)
Returns list of all hosts

`cfme.infrastructure.host.`**`get_credentials_from_config`**(*credential_config_name*)

`cfme.infrastructure.host.`**`get_from_config`**(*provider_config_name*)
Creates a Host object given a yaml entry in cfme_data.

Usage:

```
get_from_config('esx')
```

Returns: A Host object that has methods that operate on CFME

`cfme.infrastructure.host.`**`wait_for_a_host`**()

`cfme.infrastructure.host.`**`wait_for_host_delete`**(*host*)

`cfme.infrastructure.host.`**`wait_for_host_to_appear`**(*host*)

cfme.infrastructure.provider module A model of an Infrastructure Provider in CFME

var page A `cfme.web_ui.Region` object describing common elements on the Providers pages.

var discover_form A `cfme.web_ui.Form` object describing the discover form.

var properties_form A `cfme.web_ui.Form` object describing the main add form.

var default_form A `cfme.web_ui.Form` object describing the default credentials form.

var candu_form A `cfme.web_ui.Form` object describing the C&U credentials form.

class `cfme.infrastructure.provider.`**`OpenstackInfraProvider`**(*name=None*, *credentials=None*, *key=None*, *hostname=None*, *ip_address=None*, *start_ip=None*, *end_ip=None*, *provider_data=None*)
Bases: `cfme.infrastructure.provider.Provider`

STATS_TO_MATCH = ['num_template', 'num_host']

class cfme.infrastructure.provider.**Provider**(*name=None*, *credentials=None*, *key=None*, *zone=None*, *provider_data=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty, cfme.common.provider.CloudInfraProvide

Abstract model of an infrastructure provider in cfme. See VMwareProvider or RHEVMProvider.

Parameters

- **name** – Name of the provider.
- **details** – a details record (see VMwareDetails, RHEVMDetails inner class).
- **credentials** (*Credential*) – see Credential inner class.
- **key** – The CFME key of the provider in the yaml.
- **candu** – C&U credentials if this is a RHEVMDetails class.

Usage:

```
myprov = VMwareProvider(name='foo',
                        region='us-west-1',
                        credentials=Provider.Credential(principal='admin', secret='foobar'))
myprov.create()
```

STATS_TO_MATCH = ['num_template', 'num_vm', 'num_datastore', 'num_host', 'num_cluster']

add_provider_button = FormButton('Add this Infrastructure Provider')

detail_page_suffix = 'provider'

discover()

Begins provider discovery from a provider instance

Usage:

```
discover_from_config(utils.providers.get_crud('rhevm'))
```

edit_page_suffix = 'provider_edit'

hosts

Returns list of cfme.infrastructure.host.Host that should belong to this provider according to the YAML

num_cluster(*db=True*)

Returns the providers number of templates, as shown on the Details page.

num_datastore(*db=True*)

Returns the providers number of templates, as shown on the Details page.

num_host(*db=True*)

Returns the providers number of instances, as shown on the Details page.

page_name = 'infrastructure'

pretty_attrs = ['name', 'key', 'zone']

properties_form = <cfme.web_ui.Form fields=[('type_select', {Version ('lowest'): Select('select#server_emstype', mul

quad_name = 'infra_prov'

save_button = FormButton('Save Changes')

string_name = 'Infrastructure'

class cfme.infrastructure.provider.**RHEVMProvider**(*name=None*, *credentials=None*, *zone=None*, *key=None*, *hostname=None*, *ip_address=None*, *api_port=None*, *start_ip=None*, *end_ip=None*, *provider_data=None*)

Bases: cfme.infrastructure.provider.Provider

class cfme.infrastructure.provider.**SCVMMProvider**(*name=None*, *credentials=None*, *key=None*, *zone=None*, *hostname=None*, *ip_address=None*, *start_ip=None*, *end_ip=None*, *sec_protocol=None*, *sec_realm=None*, *provider_data=None*)

Bases: cfme.infrastructure.provider.Provider

STATS_TO_MATCH = ['num_template', 'num_vm']

class cfme.infrastructure.provider.**VMwareProvider**(*name=None*, *credentials=None*, *key=None*, *zone=None*, *hostname=None*, *ip_address=None*, *start_ip=None*, *end_ip=None*, *provider_data=None*)

Bases: cfme.infrastructure.provider.Provider

cfme.infrastructure.provider.**discover**(*rhevm=False*, *vmware=False*, *scvmm=False*, *cancel=False*, *start_ip=None*, *end_ip=None*)

Discover infrastructure providers. Note: only starts discovery, doesn't wait for it to finish.

Parameters

- **rhvem** – Whether to scan for RHEVM providers
- **vmware** – Whether to scan for VMware providers
- **scvmm** – Whether to scan for SCVMM providers
- **cancel** – Whether to cancel out of the discover UI.

cfme.infrastructure.provider.**get_all_providers**(*do_not_navigate=False*)

Returns list of all providers

cfme.infrastructure.provider.**wait_for_a_provider**()

cfme.infrastructure.pxe module A model of a PXE Server in CFME

class cfme.infrastructure.pxe.**CustomizationTemplate**(*name=None*, *description=None*, *image_type=None*, *script_type=None*, *script_data=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Model of a Customization Template in CFME

Parameters

- **name** – The name of the template.
- **description** – Template description.
- **image_type** – Image type name, must be one of an existing System Image Type.

- **script_type** – Script type, either Kickstart, Cloudinit or Sysprep.
- **script_data** – The scripts data.

create(*cancel=False*)
Creates a Customization Template object

Parameters **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the CT has been filled in the UI.

delete(*cancel=True*)
Deletes a Customization Template server from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

exists(*db=True*)
Checks if the Customization template already exists

pretty_attrs = ['name', 'image_type']

update(*updates*, *cancel=False*)
Updates a Customization Template server in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

class cfme.infrastructure.pxe.**ISODatastore**(*provider=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

Model of a PXE Server object in CFME

Parameters **provider** – Provider name.

create(*cancel=False*, *refresh=True*, *refresh_timeout=120*)
Creates an ISO datastore object

Parameters

- **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the ISO datastore has been filled in the UI.
- **refresh** (*boolean*) – Whether to run the refresh operation on the ISO datastore after the add has been completed.

delete(*cancel=True*)
Deletes an ISO Datastore from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

exists(*db=True*)
Checks if the ISO Datastore already exists

pretty_attrs = ['provider']

refresh(*wait=True*, *timeout=120*)
Refreshes the PXE relationships and waits for it to be updated

set_iso_image_type(*image_name*, *image_type*)
Function to set the image type of a PXE image

class cfme.infrastructure.pxe.**PXEServer**(*name=None*, *depot_type=None*, *uri=None*, *userid=None*, *password=None*, *access_url=None*, *pxe_dir=None*, *windows_dir=None*, *customize_dir=None*, *menu_filename=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Model of a PXE Server object in CFME

Parameters

- **name** – Name of PXE server.
- **depot_type** – Depot type, either Samba or Network File System.
- **uri** – The Depot URI.
- **userid** – The Samba username.
- **password** – The Samba password.
- **access_url** – HTTP access path for PXE server.
- **pxe_dir** – The PXE dir for accessing configuration.
- **windows_dir** – Windows source directory.
- **customize_dir** – Customization directory for templates.
- **menu_filename** – Menu filename for iPXE/syslinux menu.

create(*cancel=False*, *refresh=True*, *refresh_timeout=120*)

Creates a PXE server object

Parameters

- **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the PXE Server has been filled in the UI.
- **refresh** (*boolean*) – Whether to run the refresh operation on the PXE server after the add has been completed.

delete(*cancel=True*)

Deletes a PXE server from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

exists(*db=True*)

Checks if the PXE server already exists

get_pxe_image_type(*image_name*, *db=True*)

pretty_attrs = ['name', 'uri', 'access_url']

refresh(*wait=True*, *timeout=120*)

Refreshes the PXE relationships and waits for it to be updated

set_pxe_image_type(*image_name*, *image_type*)

Function to set the image type of a PXE image

update(*updates*, *cancel=False*)

Updates a PXE server in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

class cfme.infrastructure.pxe.**SystemImageType**(*name=None*, *provision_type=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

Model of a System Image Type in CFME.

Parameters

- **name** – The name of the System Image Type.
- **provision_type** – The provision type, either Vm or Host.

create(*cancel=False*)
Creates a System Image Type object

Parameters **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the SIT has been filled in the UI.

delete(*cancel=True*)
Deletes a System Image Type from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to True

pretty_attrs = ['name', 'provision_type']

update(*updates*, *cancel=False*)
Updates a System Image Type in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

cfme.infrastructure.pxe.**get_pxe_server_from_config**(*pxe_config_name*)
Convenience function to grab the details for a pxe server fomr the yamls.

cfme.infrastructure.pxe.**get_template_from_config**(*template_config_name*)
Convenience function to grab the details for a template from the yamls.

cfme.infrastructure.pxe.**remove_all_pxe_servers**()
Convenience function to remove all PXE servers

cfme.infrastructure.repositories module Infrastructure / Repositories

class cfme.infrastructure.repositories.**Repository**(*name=None*, *path=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

Model of an infrastructure repository in cfme.

Parameters

- **name** – Name of the repository host
- **path** – UNC path to the repository share

Usage:

```
myrepo = Repository(name='vmware', path='//hostname/path/to/share')
myrepo.create()
```

create(*cancel=False*, *validate_credentials=False*)
Creates a repository in the UI

Parameters

- **cancel** (*boolean*) – Whether to cancel out of the creation. The cancel is done after all the information present in the Host has been filled in the UI.
- **validate_credentials** (*boolean*) – Whether to validate credentials - if True and the credentials are invalid, an error will be raised.

delete(*cancel=False*)
Deletes a repository from CFME

Parameters **cancel** – Whether to cancel the deletion, defaults to False

exists

get_detail(**ident*)
Gets details from the details infoblock

The function first ensures that we are on the detail page for the specific repository.

Parameters ***ident** – An InfoBlock title, followed by the Key name, e.g. "Relationships", "Images"

Returns: A string representing the contents of the InfoBlock's value.

pretty_attrs = ['name', 'path']

update(*updates*, *cancel=False*, *validate_credentials=False*)
Updates a repository in the UI. Better to use utils.update.update context manager than call this directly.

Parameters

- **updates** (*dict*) – fields that are changing.
- **cancel** (*boolean*) – whether to cancel out of the update.

cfme.infrastructure.resource_pool module

cfme.infrastructure.virtual_machines module

Module contents

cfme.intelligence package

Subpackages

cfme.intelligence.reports package

Submodules

cfme.intelligence.reports.dashboards module Module handling Dashboards accordion.

class cfme.intelligence.reports.dashboards.**Dashboard**(*name*, *group*, *title=None*, *locked=None*, *widgets=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('title', <cfme.web_

group

pretty_attrs = ['name', 'group', 'title', 'widgets']

update(*updates*)

class cfme.intelligence.reports.dashboards.**DefaultDashboard**(*title=None*, *locked=None*, *widgets=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('title', <cfme.web_ui.Input _names=('description',), _use_id=False>), ('locked', <cfm

pretty_attrs = ['title', 'widgets']

update(*updates*)

cfme.intelligence.reports.dashboards.**go_to_default_func**(_)

This can change, because the title of the default dashboard is mutable. However, we can xpath there quite reliable, so we use it that way we extract the name from the tree directly.

cfme.intelligence.reports.import_export module

cfme.intelligence.reports.import_export.**export_reports**(**custom_report_names*)

cfme.intelligence.reports.import_export.**import_reports**(*filename*, *overwrite=False*)

cfme.intelligence.reports.menus module

Module handling report menus contents

cfme.intelligence.reports.menus.**add_folder**(*group*, *folder*)

Adds a folder under top-level.

Parameters

- **group** – User group.
- **folder** – Name of the new folder.

cfme.intelligence.reports.menus.**add_subfolder**(*group*, *folder*, *subfolder*)

Adds a subfolder under specified folder.

Parameters

- **group** – User group.
- **folder** – Name of the folder.
- **subfolder** – Name of the new subdfolder.

cfme.intelligence.reports.menus.**get_folders**(*group*)

Returns list of folders for given user group.

Parameters **group** – User group to check.

cfme.intelligence.reports.menus.**get_subfolders**(*group*, *folder*)

Returns list of sub-folders for given user group and folder.

Parameters

- **group** – User group to check.
- **folder** – Folder to read.

cfme.intelligence.reports.menus.**manage_folder**(**args*, ***kwds*)
Context manager to use when modifying the folder contents.

You can use manager's FolderManager.bail_out() classmethod to end and discard the changes done inside the with block. This context manager does not give the manager as a value to the with block so you have to import and use the FolderManager class manually.

Parameters

- **group** – User group.
- **folder** – Which folder to manage. If None, top-level will be managed.

Returns: Context-managed cfme.intelligence.reports.ui_elements.FolderManager inst.

cfme.intelligence.reports.menus.**manage_subfolder**(**args*, ***kwds*)
Context manager to use when modifying the subfolder contents.

You can use manager's FolderManager.bail_out() classmethod to end and discard the changes done inside the with block.

Parameters

- **group** – User group.
- **folder** – Parent folder name.
- **subfolder** – Subfodler name to manage.

Returns: Context-managed cfme.intelligence.reports.ui_elements.FolderManager inst.

cfme.intelligence.reports.menus.**reset_to_default**(*group*)
Clicks the *Default* button.

Parameters **group** – Group to set to Default

cfme.intelligence.reports.reports module Module handling definition, CRUD, queuing Reports.

Extensively uses cfme.intelligence.reports.ui_elements

class cfme.intelligence.reports.reports.**CannedSavedReport**(*path_to_report*, *datetime*)
Bases: cfme.intelligence.reports.reports.CustomSavedReport

As we cannot create or edit canned reports, we don't know their titles and so, so we need to change the navigation a little bit for it to work correctly.

Parameters

- **path_to_report** – Iterable with path to report.
- **datetime** – Datetime of "Run At" of the report. That's what queue_canned_report() returns.

navigate()

classmethod new(*path*)

class cfme.intelligence.reports.reports.**CustomReport**(***values*)
Bases: utils.update.Updateable

create(*cancel=False*)

delete(*cancel=False*)

get_saved_reports()

queue(*wait_for_finish=False*)

update(*updates*)

class cfme.intelligence.reports.reports.**CustomSavedReport**(*report*, *datetime*)
Bases: utils.update.Updateable, utils.pretty.Pretty

Custom Saved Report. Enables us to retrieve data from the table.

Parameters

- **report** – Report that we have data from.
- **datetime** – Datetime of "Run At" of the report. That's what queue_canned_report() returns.

data
Retrieves data from the saved report.

Returns: SavedReportData

This attribute is lazily evaluated and cached.

download(*extension*)

navigate()

pretty_attrs = ['report', 'datetime']

class cfme.intelligence.reports.reports.**SavedReportData**(*headers*, *body*)
Bases: utils.pretty.Pretty

This class stores data retrieved from saved report.

Parameters

- **headers** – Tuple with header columns.
- **body** – List of tuples with body rows.

find_cell(*column*, *value*, *cell*)

find_row(*column*, *value*)

pretty_attrs = ['headers', 'body']

rows

cfme.intelligence.reports.reports.**get_report_name**(*o*)

cfme.intelligence.reports.reports.**get_saved_canned_reports**(**path*)

cfme.intelligence.reports.reports.**queue_canned_report**(**path*)
Queue report from selection of pre-prepared reports.

Parameters ***path** – Path in tree after All Reports

Returns: Value of Run At in the table so the run can be then checked.

cfme.intelligence.reports.reports.**reload_view**()
Reloads and keeps on the current tabstrip page

cfme.intelligence.reports.reports.**select**(***kwargs*)

cfme.intelligence.reports.reports.**tag**(*tag_name*, ***kwargs*)

cfme.intelligence.reports.saved module

cfme.intelligence.reports.saved.**delete_saved_report**(*cancel=False*)

cfme.intelligence.reports.saved.**get_saved_reports_for**(*name*)

cfme.intelligence.reports.saved.**go_to_latest_saved_report_for**(*name*)

cfme.intelligence.reports.saved.**show_full_screen**(*cancel=False*)

cfme.intelligence.reports.schedules module Module handling schedules

class cfme.intelligence.reports.schedules.**Schedule**(*name*, *description*, *filter*, *active=None*, *timer=None*, *send_email=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Represents a schedule in Intelligence/Reports/Schedules.

Parameters

- **name** – Schedule name.
- **description** – Schedule description.
- **filter** – 3-tuple with filter selection (see the UI).
- **active** – Whether is this schedule active.
- **run** – Specifies how often this schedule runs. It can be either string "Once", or a tuple, which maps to the two selects in UI ("Hourly", "Every hour")...
- **time_zone** – Specify time zone.
- **start_date** – Specify the start date.
- **start_time** – Specify the start time either as a string ("0:15") or tuple ("0", "15")
- **send_email** – If specifies, turns on e-mail sending. Can be string, or list or set.

create(*cancel=False*)

delete(*cancel=False*)

classmethod delete_schedules(**schedules*, ***kwargs*)

Select and delete specified schedules from VMDB.

Parameters

- ***schedules** – Schedules to delete. Can be objects or strings.
- **cancel** – (kwarg) Whether to cancel the deletion (Default: False)

Raises: NameError when some of the schedules were not found.

classmethod disable_schedules(**schedules*)

Select and disable specified schedules.

Parameters ***schedules** – Schedules to disable. Can be objects or strings.

Raises: NameError when some of the schedules were not found.

classmethod enable_schedules(**schedules*)

Select and enable specified schedules.

Parameters ***schedules** – Schedules to enable. Can be objects or strings.

Raises: NameError when some of the schedules were not found.

exists

form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>), ('description', <cfm

pretty_attrs = ['name', 'filter']

queue(*wait_for_finish=False*)
Queue this schedule.

Parameters **wait_for_finish** – If True, then this function blocks until the action is finished.

classmethod queue_schedules(**schedules*)
Select and queue specified schedules.

Parameters ***schedules** – Schedules to queue. Can be objects or strings.

Raises: `NameError` when some of the schedules were not found.

table_item(*item*)
Works both up- and downstream.

I think this should be incorporated into InfoBlock somehow. Currently there is the fieldset issue.

update(*updates*)

cfme.intelligence.reports.schedules.**get_sch_name**(*sch*)
Enables us using both string and schedule object

cfme.intelligence.reports.ui_elements module This file contains element definitions of elements that are common in reports.

class cfme.intelligence.reports.ui_elements.**ColumnHeaderFormatTable**(*table_locator*, *header_offset=0*, *body_offset=0*)
Bases: `cfme.web_ui.Table`

Used to fill the table with header names and value formatting.

The value expected for filling is a `dict` where keys are names of the columns (leftmost cells in the table) and values are `dict` , `str` or `list`. In case of dictionary, the `header` and `format` fields are required, they correspond to the table columns. If a string is specified, it is considered a header. If a list is specified, then first item in it is considered a `header`, second one a `format` (if present).

```
Archived:
    header: Test1
    format: Boolean (T/F)
Busy:
    header: Such busy
    format: Boolean (Yes/No)
asdf: fghj
qwer:
- thisisheader
- thisisformat
```

class cfme.intelligence.reports.ui_elements.**ColumnStyleTable**(*div_id*)
Bases: `utils.pretty.Pretty`

We cannot inherit Table because it does too much WebElement chaining. This avoids that with using xpath-only locating making it much more reliable.

This is the kind of table that is used in Styling tab. The fill value is expected to be a `dict`. Keys of the dictionary are names of the columns (leftmost table cell). The values of the dictionary are lists up to 3 fields long. First element of each of the lists is the `Style` to be selected in the same-named table column. Second one is the

operation (=, `IS NULL`, ...) to happen. If the operation has some operand, it is the third (and last) element of the list. If any of the lists has operation set as `Default`, no other lists cannot follow after them.

```
Name:
 -
     - Blue Text
     - "="
     - asdf
 -
     - Yellow Background
     - IS NULL
 -
     - Red Background
     - IS NOT NULL
```

Parameters **div_id** – *id* of *div* where the table is located in.

get_if_input(*name*, *id=0*)

Return the *input* element with value selection.

Parameters

- **name** – Text written in leftmost column of the wanted row.
- **id** – Sequential id in the sub-row, 0..2.

Returns: `str` with locator.

get_if_select(*name*, *id=0*)

Return Select element with operator selection.

Parameters

- **name** – Text written in leftmost column of the wanted row.
- **id** – Sequential id in the sub-row, 0..2.

Returns: `cfme.web_ui.Select`.

get_style_select(*name*, *id=0*)

Return Select element with selected style.

Parameters

- **name** – Text written in leftmost column of the wanted row.
- **id** – Sequential id in the sub-row, 0..2.

Returns: `cfme.web_ui.Select`.

pretty_attrs = ['_div_id']

class `cfme.intelligence.reports.ui_elements.`**DashboardWidgetSelector**(*root_loc="//div[@id='form_widgets*

Bases: `utils.pretty.Pretty`

This object encapsulates the selector of widgets that will appear on a dashboard.

It cannot move them around, just add and remove them.

The filling of this element expects a `list` of strings (or just `str` itself). The strings are names of the widgets.

clear()

combo

deselect(**items*)

pretty_attrs = ['_root_loc']

select(**items*)

selected_items

class cfme.intelligence.reports.ui_elements.**ExternalRSSFeed**
Bases: object

This element encapsulates selection of an external RSS source either from canned selection or custom one.

It expects a str filling object. If the string is not found in the dropdown, it is considered to be custom url, so it selects custom URL option in the dropdown and fills the text input with the URL. If the option is available in the dropdown, then it is selected.

form = <cfme.web_ui.Region title=None>

class cfme.intelligence.reports.ui_elements.**FolderManager**(*root*)
Bases: utils.pretty.Pretty

Class used in Reports/Edit Reports menus.

add(*subfolder*)

add_subfolder()

classmethod bail_out()
If something gets wrong, you can use this method to cancel editing of the items in the context manager.

Raises: FolderManager._BailOut exception

clear()

commit()

delete_field(*field*)

delete_folder()

discard()

fields
Returns all fields' text values

has_field(*field*)
Returns if the field is present.

Parameters **field** – Field to check.

move_bottom()

move_down()

move_first(*field*)

move_last(*field*)

move_top()

move_up()

pretty_attrs = ['root']

select_field(*field*)
Select field by text.

Parameters **field** – Field text.

selected_field
Return selected field's text.

Returns: `str` if field is selected, else *None*

selected_field_element
Return selected field's element.

Returns: `WebElement` if field is selected, else *None*

class `cfme.intelligence.reports.ui_elements.`**MenuShortcuts**(*select_name*)
Bases: `utils.pretty.Pretty`

This class operates the web ui object that handles adding new menus and shortcuts for widgets

The expected object for filling is one of `dict`, `list` or `str`. If `dict`, then the keys are menu item names and their values are their aliases. If you don't want to specify an alias for such particular menu item, use None. `str` behaves same as single element `list`. If `list`, then it is the same as it would be with `dict` but you cannot specify aliases, just menu names.

Parameters **select_name** – Name of the select

add(*menu*, *alias=None*)
Add a new shortcut.

Parameters

- **menu** – What menu item to select.
- **alias** – Optional alias for this menu item.

clear()
Clear the selection.

close_box(*id*)

get_text_of(*id*)

mapping
Determine mapping Menu item => menu item id.

Needed because the boxes with shortcuts are accessible only via ids. Need to close boxes because boxes displayed are not in the Select.

This attribute is lazily evaluated and cached.

opened_boxes_ids
Return ids of all opened boxes.

pretty_attrs = ['_select_name']

select

set_text_of(*id*, *text*)

class `cfme.intelligence.reports.ui_elements.`**NewerDashboardWidgetSelector**(*root_loc="//div[@id='form_*
Bases: `cfme.intelligence.reports.ui_elements.DashboardWidgetSelector`

Dashboard widget selector from 5.5 onwards.

combo

select(**items*)

exception `cfme.intelligence.reports.ui_elements.`**NotDisplayedException**
Bases: `exceptions.Exception`

class cfme.intelligence.reports.ui_elements.**PivotCalcSelect**(*root_el_id*)
Bases: utils.pretty.Pretty

This class encapsulates those JS pseudo-selects in Edit Report/Consolidation

classmethod all()
For debugging purposes

check(*item*)

clear_selection()

classmethod close_all_boxes()
No other solution as the boxes have no ID

id

items()

pretty_attrs = ['_id']

uncheck(*item*)

class cfme.intelligence.reports.ui_elements.**RecordGrouper**(*table_loc*)
Bases: utils.pretty.Pretty

This class encapsulates the grouping editing in Edit Report/Consolidation in the table at the bottom

Filling this element expects a dict. The key of the dictionary is the name of the column (leftmost table cell). The value of the dictionary is a list of values that will get selected in the dropdown (Minimum, Average, ...)

```
CPU - % Overallocated:
  - Maximum
  - Minimum
  - Average
```

pretty_attrs = ['_table_loc']

class cfme.intelligence.reports.ui_elements.**Timer**
Bases: object

form = <cfme.web_ui.Form fields=[('run', Select("//select[@id='timer_typ']", multi=False)), ('hours', Select("//select[@id

cfme.intelligence.reports.widgets module Module handling Dashboard Widgets accordion.

class cfme.intelligence.reports.widgets.**ChartWidget**(*title*, *description=None*, *active=None*, *filter=None*, *timer=None*, *visibility=None*)
Bases: cfme.intelligence.reports.widgets.Widget

DETAIL_PAGE = 'reports_widgets_chart'

TITLE = 'Chart'

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('title', <cfme.web_ui.Input _names=('title',), _use_id=False>), ('description', <cfme.w

pretty_attrs = ['title', 'description', 'filter', 'visibility']

update(*updates*)

class cfme.intelligence.reports.widgets.**MenuWidget**(*title*, *description=None*, *active=None*, *shortcuts=None*, *visibility=None*)

Bases: cfme.intelligence.reports.widgets.Widget

DETAIL_PAGE = 'reports_widgets_menu'

TITLE = 'Menu'

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('title', <cfme.web_ui.Input _names=('title',), _use_id=False>), ('description', <cfme.v

pretty_attrs = ['description', 'shortcuts', 'visibility']

update(*updates*)

class cfme.intelligence.reports.widgets.**RSSFeedWidget**(*title*, *description=None*, *active=None*, *type=None*, *feed=None*, *external=None*, *rows=None*, *timer=None*, *visibility=None*)

Bases: cfme.intelligence.reports.widgets.Widget

DETAIL_PAGE = 'reports_widgets_rss_feed'

TITLE = 'RSS Feed'

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('title', <cfme.web_ui.Input _names=('title',), _use_id=False>), ('description', <cfme.v

pretty_attrs = ['title', 'description', 'type', 'feed', 'visibility']

update(*updates*)

class cfme.intelligence.reports.widgets.**ReportWidget**(*title*, *description=None*, *active=None*, *filter=None*, *columns=None*, *rows=None*, *timer=None*, *visibility=None*)

Bases: cfme.intelligence.reports.widgets.Widget

DETAIL_PAGE = 'reports_widgets_report'

TITLE = 'Report'

create(*cancel=False*)

delete(*cancel=False*)

form = <cfme.web_ui.Form fields=[('title', <cfme.web_ui.Input _names=('title',), _use_id=False>), ('description', <cfme.v

pretty_attrs = ['description', 'filter', 'visibility']

update(*updates*)

class cfme.intelligence.reports.widgets.**Widget**

Bases: utils.update.Updateable, utils.pretty.Pretty

DETAIL_PAGE = None

TITLE = None

WAIT_STATES = set(['Running', 'Queued'])

check_status()

classmethod detect(*t*, **args*, ***kwargs*)

generate(*wait=True*, ***kwargs*)

go_to_detail()

on_widget_page

status_info = <cfme.web_ui.InfoBlock title='Status'>

wait_generated(*timeout=600*)

Module contents This is a directory of modules, each one represents one accordion item.

- cfme.intelligence.reports.reports
- cfme.intelligence.reports.schedules
- cfme.intelligence.reports.import_export
- cfme.intelligence.reports.saved
- cfme.intelligence.reports.widgets
- cfme.intelligence.reports.dashboards

Submodules

cfme.intelligence.chargeback module

class cfme.intelligence.chargeback.**Assign**(*assign_to=None*, *tag_category=None*, *selections=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Model of Chargeback Assignment page in cfme.

Parameters

- **assign_to** – Assign the chargeback rate to entities such as VM,Provider,datastore or the Enterprise itself.
- **tag_category** – Tag category of the entity
- **selections** – Selection of a particular entity to which the rate is to be assigned. Eg:If the chargeback rate is to be assigned to providers,select which of the managed providers the rate is to be assigned.

Usage:

```
tagged_datastore = Assign(
    assign_to="Tagged Datastores",
    tag_category="Location",
    selections={
        "Chicago": "Default"
})
```

tagged_datastore.storageassign()

computeassign()

storageassign()

class cfme.intelligence.chargeback.**AssignFormTable**(*entry_loc*)
Bases: utils.pretty.Pretty

locate()

pretty_attrs = ['entry_loc']

row_by_name(*name*)

rows

select_by_name(*name*)

select_from_row(*row*)

class cfme.intelligence.chargeback.**ComputeRate**(*description=None*, *cpu_alloc=None*, *cpu_used=None*, *disk_io=None*, *compute_fixed_1=None*, *compute_fixed_2=None*, *mem_alloc=None*, *mem_used=None*, *net_io=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

create()

delete()

pretty_attrs = ['description']

update(*updates*)

class cfme.intelligence.chargeback.**RateFormItem**(*rate_loc=None*, *unit_select_loc=None*)
Bases: utils.pretty.Pretty

pretty_attrs = ['rate_loc', 'unit_select_loc']

class cfme.intelligence.chargeback.**StorageRate**(*description=None*, *storage_fixed_1=None*, *storage_fixed_2=None*, *storage_alloc=None*, *storage_used=None*)
Bases: utils.update.Updateable, utils.pretty.Pretty

create()

delete()

pretty_attrs = ['description']

update(*updates*)

Module contents

This is a directory of modules, each one represents one menu sub-item.

- cfme.intelligence.reports
- cfme.intelligence.chargeback

cfme.services package

Subpackages

cfme.services.catalogs package

Submodules

cfme.services.catalogs.catalog module

class cfme.services.catalogs.catalog.**Catalog**(*name=None*, *description=None*, *items=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

Represents a Catalog

create()

delete()

pretty_attrs = ['name', 'items']

update(*updates*)

cfme.services.catalogs.catalog_item module

class cfme.services.catalogs.catalog_item.**CatalogBundle**(*name=None*, *description=None*, *display_in=False*, *catalog=None*, *dialog=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

create(*cat_items*)

pretty_attrs = ['name', 'catalog', 'dialog']

update(*updates*)

class cfme.services.catalogs.catalog_item.**CatalogItem**(*item_type=None*, *name=None*, *description=None*, *display_in=False*, *catalog=None*, *dialog=None*, *catalog_name=None*, *orch_template=None*, *provider_type=None*, *provider=None*, *prov_data=None*, *domain='ManageIQ (Locked)'*)

Bases: utils.update.Updateable, utils.pretty.Pretty

add_button()

add_button_group()

create()

delete()

edit_tags(*tag*, *value*)

pretty_attrs = ['name', 'item_type', 'catalog', 'catalog_name', 'provider', 'domain']

update(*updates*)

cfme.services.catalogs.cloud_catalog_item module

cfme.services.catalogs.myservice module

class cfme.services.catalogs.myservice.**MyService**(*service_name*, *vm_name=None*)

Bases: utils.update.Updateable

Create,Edit and Delete Button Groups

Parameters

- **service_name** – The name of service to retire.
- **vm_name** – Name of vm in the service.
- **retirement_date** – Date to retire service.

check_vm_add(*add_vm_name*)

delete(*name*)

edit_tags(*tag*, *value*)

get_detail(*properties=None*)

Gets details from the details infoblock

Parameters ***ident** – An InfoBlock title, followed by the Key name e.g. "Relationships", "Images"

Returns: A string representing the contents of the InfoBlock's value.

reconfigure_service()

retire()

retire_on_date(*retirement_date*)

set_ownership(*owner*, *group*)

update(*name*, *description*)

cfme.services.catalogs.orchestration_template module

class cfme.services.catalogs.orchestration_template.**OrchestrationTemplate**(*template_type=None*, *template_name=None*, *description=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

copy_template(*template_name*, *content*)

create(*content*)

create_service_dialog(*dialog_name*)

create_service_dialog_from_template(*dialog_name*, *template_name*)

delete()

pretty_attrs = ['template_type', 'template_name']

update(*updates*)

cfme.services.catalogs.service_catalogs module

class cfme.services.catalogs.service_catalogs.**ServiceCatalogs**(*service_name=None*, *stack_data=None*)

Bases: utils.update.Updateable, utils.pretty.Pretty

order(*catalog*, *catalog_item*)

order_stack_item(*catalog*, *catalog_item*)

pretty_attrs = ['service_name']

Module contents

Submodules

cfme.services.requests module

cfme.services.requests.**approve**(*reason*, *cancel=False*)
Approve currently opened request

Parameters

- **reason** – Reason for approving the request.
- **cancel** – Whether to cancel the approval.

cfme.services.requests.**approve_request**(*cells*, *reason*, *cancel=False*)
Open the specified request and approve it.

Parameters

- **cells** – Search data for the requests table.
- **reason** – Reason for approving the request.
- **cancel** – Whether to cancel the approval.

Raises RequestException – cfme.exceptions.RequestException if the request was not found

cfme.services.requests.**copy_request**(**args*, ***kwds*)
Context manager that opens the request for editing and saves or cancels depending on success.

Parameters **cells** – Search data for the requests table.

cfme.services.requests.**debug_requests**()

cfme.services.requests.**delete**(*cancel=False*)
Delete currently opened request

Parameters **cancel** – Whether to cancel the deletion.

cfme.services.requests.**delete_request**(*cells*, *cancel=False*)
Open the specified request and delete it.

Parameters

- **cells** – Search data for the requests table.
- **cancel** – Whether to cancel the deletion.

Raises RequestException – cfme.exceptions.RequestException if the request was not found

cfme.services.requests.**deny**(*reason*, *cancel=False*)
Deny currently opened request

Parameters

- **reason** – Reason for denying the request.
- **cancel** – Whether to cancel the denial.

cfme.services.requests.**deny_request**(*cells*, *reason*, *cancel=False*)
Open the specified request and deny it.

Parameters

- **cells** – Search data for the requests table.
- **reason** – Reason for denying the request.
- **cancel** – Whether to cancel the denial.

Raises `RequestException` – `cfme.exceptions.RequestException` if the request was not found

`cfme.services.requests.`**`edit_request`**(**args*, ***kwds*)
Context manager that opens the request for editing and saves or cancels depending on success.

Parameters **cells** – Search data for the requests table.

`cfme.services.requests.`**`go_to_request`**(*cells*)
Finds the request and opens the page

See `wait_for_request()` for futher details.

Parameters **cells** – Search data for the requests table.

Returns: Success of the action.

`cfme.services.requests.`**`reload`**()

`cfme.services.requests.`**`wait_for_request`**(*cells*, *partial_check=False*)
helper function checks if a request is complete

After finding the request's row using the `cells` argument, this will wait for a request to reach the 'Finished' state and return it. In the event of an 'Error' state, it will raise an AssertionError, for use with `pytest.raises`, if desired.

Parameters **cells** – A dict of cells use to identify the request row to inspect in the `request_list` Table. See `cfme.web_ui.Table.find_rows_by_cells()` for more.

Usage:

```
# Filter on the "Description" column
description = 'Provision from [%s] to [%s]' % (template_name, vm_name)
cells = {'Description': description}

# Filter on the "Request ID" column
# Text must match exactly, you can use "{:,}".format(request_id) to add commas if needed.
request_id = '{:,}'.format(1000000000001)  # Becomes '1,000,000,000,001', as in the table
cells = {'Request ID': request_id}

# However you construct the cells dict, pass it to wait_for_request
# Provisioning requests often take more than 5 minutes but less than 10.
wait_for(wait_for_request, [cells], num_sec=600)
```

Raises

- `AssertionError` – if the matched request has status 'Error'
- `RequestException` – if multiple matching requests were found

Returns The matching `cfme.web_ui.Table.Row` if found, `False` otherwise.

cfme.services.workloads module A model of Workloads page in CFME

Module contents

cfme.storage package

Submodules

cfme.storage.file_shares module

class cfme.storage.file_shares.**FileShare**

Bases: tuple

FileShare(name, element_name, vms, hosts, datastores, op_status, region, last_upd_status)

datastores
Alias for field number 4

element_name
Alias for field number 1

hosts
Alias for field number 3

last_upd_status
Alias for field number 7

name
Alias for field number 0

op_status
Alias for field number 5

region
Alias for field number 6

vms
Alias for field number 2

cfme.storage.file_shares.**all**()

Returns all of the file shares available

cfme.storage.filers module

class cfme.storage.filers.**Filer**

Bases: tuple

Filer(name, element_name, vms, hosts, datastores, health_status, op_status, description, region, last_upd_status)

datastores
Alias for field number 4

description
Alias for field number 7

element_name
Alias for field number 1

health_status
Alias for field number 5

hosts
Alias for field number 3

last_upd_status
Alias for field number 9

name
Alias for field number 0

op_status
Alias for field number 6

region
Alias for field number 8

vms
Alias for field number 2

cfme.storage.filers.**all**()
Returns all of the file shares available

cfme.storage.luns module

class cfme.storage.luns.**LUN**
Bases: tuple

LUN(name, element_name, vms, hosts, datastores, health_status, op_status, description, region, last_upd_status)

datastores
Alias for field number 4

description
Alias for field number 7

element_name
Alias for field number 1

health_status
Alias for field number 5

hosts
Alias for field number 3

last_upd_status
Alias for field number 9

name
Alias for field number 0

op_status
Alias for field number 6

region
Alias for field number 8

vms
Alias for field number 2

cfme.storage.luns.**all**()
Returns all of the file shares available

cfme.storage.managers module

class cfme.storage.managers.**StorageManager**(*name=None*, *type=None*, *hostname=None*, *ip=None*, *port=None*, *credentials=None*)
Bases: utils.update.Updateable

Represents the Storage / Storage Managers object. Allows interaction

Parameters

- **name** – Name of the Storage Namager as it appears in the UI.
- **type** – Type of the Storage Manager (eg. StorageManager.NETAPP_RS, ...)
- **hostname** – Host name of the machine.
- **ip** – IP Address of the machine.
- **port** – Port of the machine.
- **credentials** – `dict` or `StorageManager.Credential`

class `Credential`(*username=None*, *password=None*)
Bases: `utils.update.Updateable`

`StorageManager.`**`NETAPP_RS = 'NetApp Remote Service'`**

`StorageManager.`**`add = FormButton('Add this Storage Manager')`**

`StorageManager.`**`create`**(*validate=True*, *cancel=False*)

`StorageManager.`**`delete`**(*cancel=False*)

`StorageManager.`**`exists`**

`StorageManager.`**`form = <cfme.web_ui.Form fields=[('name', <cfme.web_ui.Input _names=('name',), _use_id=False>`**

`StorageManager.`**`navigate`**()

`StorageManager.`**`refresh_inventory`**()

`StorageManager.`**`refresh_status`**()

`StorageManager.`**`update`**(*updates*, *validate=True*, *cancel=False*)

`StorageManager.`**`validate = FormButton('Validate the credentials by logging into the Server')`**

`StorageManager.`**`wait_until_updated`**(*num_sec=300*)

cfme.storage.volumes module

class `cfme.storage.volumes.`**`Volume`**
Bases: `tuple`

Volume(name, element_name, vms, hosts, datastores, health_status, op_status, description, region, last_upd_status)

`datastores`
Alias for field number 4

`description`
Alias for field number 7

`element_name`
Alias for field number 1

`health_status`
Alias for field number 5

`hosts`
Alias for field number 3

`last_upd_status`
Alias for field number 9

name
Alias for field number 0

op_status
Alias for field number 6

region
Alias for field number 8

vms
Alias for field number 2

cfme.storage.volumes.**all**()
Returns all of the file shares available

Module contents

cfme.web_ui package

Submodules

cfme.web_ui.accordion module A set of functions for dealing with accordions in the UI.

Usage:

```
Using Accordions is simply a case of either selecting it to return the element,
or using the built in click method. As shown below::

  acc = web_ui.accordion

  acc.click('Diagnostics')
  acc.is_active('Diagnostics')
```

cfme.web_ui.accordion.**click**(*name*)
Clicks an accordion and returns it

Parameters **name** – The name of the accordion.

Returns: A web element of the clicked accordion.

cfme.web_ui.accordion.**is_active**(*name*)
Checks if an accordion is currently open

Note: Only works on traditional accordions.

Parameters **name** – The name of the accordion.

Returns: `True` if the button is depressed, `False` if not.

cfme.web_ui.accordion.**locate**(*name*)
Returns an accordion by name

Parameters **name** – The name of the accordion.

Returns: A web element of the selected accordion.

cfme.web_ui.accordion.**tree**(*name*, **path*)
Get underlying Tree() object. And eventually click path.

If the accordion is not active, will be clicked. Attention! The object is 'live' so when it's obscured, it won't work!

Usage:

```
accordion.tree("Something").click_path("level 1", "level 2")
accordion.tree("Something", "level 1", "level 2")  # is the same
```

Parameters ***path** – If specified, it will directly pass these parameters into click_path of Tree. Otherwise it returns the Tree object.

cfme.web_ui.cfme_exception module Module handling the Rails exceptions from CFME

cfme.web_ui.cfme_exception.**assert_no_cfme_exception**()
Raise an exception if CFME exception occured

Raises: cfme.exceptions.CFMEExceptionOccured

cfme.web_ui.cfme_exception.**cfme_exception_text**()
Get the error message from the exception

cfme.web_ui.cfme_exception.**is_cfme_exception**()
Check whether an exception is displayed on the page

cfme.web_ui.expression_editor module The expression editor present in some locations of CFME.

class cfme.web_ui.expression_editor.**Expression**(*show_func=<function <lambda> at 0x7f52eaa7a6e0>*)
Bases: utils.pretty.Pretty

This class enables to embed the expression in a Form.

Parameters **show_func** – Function to call to show the expression if there are more of them.

pretty_attrs = ['show_func']

cfme.web_ui.expression_editor.**any_expression_present**()

cfme.web_ui.expression_editor.**click_and**()

cfme.web_ui.expression_editor.**click_commit**()

cfme.web_ui.expression_editor.**click_discard**()

cfme.web_ui.expression_editor.**click_not**()

cfme.web_ui.expression_editor.**click_or**()

cfme.web_ui.expression_editor.**click_redo**()

cfme.web_ui.expression_editor.**click_remove**()

cfme.web_ui.expression_editor.**click_undo**()

cfme.web_ui.expression_editor.**delete_whole_expression**()

cfme.web_ui.expression_editor.**fill_count**(*count=None*, *key=None*, *value=None*)
Fills the 'Count of' type of form.

If the value is unspecified and we are in the advanced search form (user input), the user_input checkbox will be checked if the value is None.

Parameters

- **count** – Name of the field to compare (Host.VMs, ...).
- **key** – Operation to do (=, <, >=, ...).
- **value** – Value to check against.

Returns: See `cfme.web_ui.fill()`.

`cfme.web_ui.expression_editor.`**`fill_field`**(*field=None*, *key=None*, *value=None*)
Fills the 'Field' type of form.

Parameters

- **tag** – Name of the field to compare (Host.VMs, ...).
- **key** – Operation to do (=, <, >=, IS NULL, ...).
- **value** – Value to check against.

Returns: See `cfme.web_ui.fill()`.

`cfme.web_ui.expression_editor.`**`fill_find`**(*field=None*, *skey=None*, *value=None*, *check=None*, *cfield=None*, *ckey=None*, *cvalue=None*)

`cfme.web_ui.expression_editor.`**`fill_registry`**(*key=None*, *value=None*, *operation=None*, *contents=None*)
Fills the 'Registry' type of form.

`cfme.web_ui.expression_editor.`**`fill_tag`**(*tag=None*, *value=None*)
Fills the 'Tag' type of form.

Parameters

- **tag** – Name of the field to compare.
- **value** – Value to check against.

Returns: See `cfme.web_ui.fill()`.

`cfme.web_ui.expression_editor.`**`get_expression_as_text`**()
Returns whole expression as represented visually.

`cfme.web_ui.expression_editor.`**`get_func`**(*name*)
Return callable from this module by its name.

Parameters **name** – Name of the variable containing the callable.

Returns: Callable from this module

`cfme.web_ui.expression_editor.`**`is_editing`**()

`cfme.web_ui.expression_editor.`**`no_expression_present`**()

`cfme.web_ui.expression_editor.`**`run_commands`**(*command_list*, *clear_expression=True*)
Run commands from the command list.

Command list syntax:

```
[
    "function1",                                                    # no args
    "function2",                                                    # dtto
    {"fill_fields": {"field1": "value", "field2": "value"}},        # Passes kwargs
    {"do_other_things": [1,2,3]}                                    # Passes args
]
```

In YAML:

```
- function1
- function2
-
    fill_fields:
        field1: value
        field2: value
-
    do_other_things:
        - 1
        - 2
        - 3
```

Parameters

- **command_list** – `list` object of the commands
- **clear_expression** – Whether to clear the expression before entering new one (default *True*)

`cfme.web_ui.expression_editor.`**`select_expression_by_text`**`(`*text*`)`

`cfme.web_ui.expression_editor.`**`select_first_expression`**`()`
There is always at least one (???), so no checking of bounds.

cfme.web_ui.flash module Provides functions for the flash area.

var area A `cfme.web_ui.Region` object representing the flash region.

class `cfme.web_ui.flash.`**`Message`**`(`*message=None*, *level=None*`)`
Bases: `utils.pretty.Pretty`

A simple class to represent a flash error in CFME.

Parameters

- **message** – The message string.
- **level** – The level of the message.

`pretty_attrs = ['message', 'level']`

`cfme.web_ui.flash.`**`assert_message_contain`**`(`**args*, ***kwargs*`)`
Asserts that a message contains a specific string

`cfme.web_ui.flash.`**`assert_message_match`**`(`**args*, ***kwargs*`)`
Asserts that a message matches a specific string.

`cfme.web_ui.flash.`**`assert_no_errors`**`(`**args*, ***kwargs*`)`
Asserts that there are no current Error messages. If no messages are passed in, they will be retrieved from the UI.

`cfme.web_ui.flash.`**`assert_success_message`**`(`**args*, ***kwargs*`)`
Asserts that there are no errors and a (green) info message matches the given string.

`cfme.web_ui.flash.`**`dismiss`**`()`
Dismiss the current flash message

`cfme.web_ui.flash.`**`get_all_messages`**`()`
Returns a list of all flash messages, (including ones hidden behind the currently showing one, if any). All flash messages will be dismissed.

`cfme.web_ui.flash.`**`get_message_level_up`**`(`*el*`)`

cfme.web_ui.flash.**get_message_text_up**(*el*)

cfme.web_ui.flash.**get_messages**()
Return a list of visible flash messages

cfme.web_ui.flash.**is_error**(*message*)
Checks a given message to see if is an Error.'

Parameters **message** – The message object.

cfme.web_ui.flash.**message**(*el*)
Turns an element into a Message object.

Parameters **el** – The element containing the flass message.

Returns: A Message object.

cfme.web_ui.flash.**verify_rails_error**(*f*)

cfme.web_ui.flash.**verpick_message**(*f*)
Wrapper that resolves eventual verpick dictionary passed to the function.

cfme.web_ui.form_buttons module This module unifies working with CRUD form buttons.

Whenever you use Add, Save, Cancel, Reset button, use this module. You can use it also for the other buttons with same shape like those CRUD ones.

class cfme.web_ui.form_buttons.**FormButton**(*alt*, *dimmed_alt=None*, *force_click=False*, *partial_alt=False*, *ng_click=None*)
Bases: utils.pretty.Pretty

This class represents the buttons usually located in forms or CRUD.

Parameters

- **alt** – The text from alt field of the image.
- **dimmed_alt** – In case the alt param is different in the dimmed variant of the button.
- **force_click** – Click always, even if it is dimmed. (Causes an error if not visible)
- **partial_alt** – Whether the alt matching should be only partial (in).
- **ng_click** – To match the angular buttons, you can use this to specify the contents of ng-click attributeh.

class Button
Holds pieces of the XPath to be assembled.

DIMMED = "(contains(@class, 'dimmed') or contains(@class, 'disabled'))"

IS_DISPLAYED = "not(ancestor::*[contains(@style, 'display:none') or contains(@style, 'display: none')])"

NOT_DIMMED = "not(contains(@class, 'dimmed') or contains(@class, 'disabled'))"

ON_CURRENT_TAB = "not(ancestor::div[contains(@class, 'tab-pane') and not(contains(@class, 'active'))])"

TAG_TYPES = '//a | //button | //img | //input'

TYPE_CONDITION = "(contains(@class, 'button') or contains(@class, 'btn') or contains(@src, 'button'))"

FormButton.**alt_expr**(*dimmed=False*)

FormButton.**can_be_clicked**
Whether the button is displayed, therefore clickable.

FormButton.**is_dimmed**

FormButton.**locate**()

FormButton.**pretty_attrs = ['_alt', '_dimmed_alt', '_force', '_partial', '_ng_click']**

cfme.web_ui.jstimelines module A Timelines object represents the Timelines widget in CFME using JS integration instead of relying on WebElements

param loc A locator for the Timelines element, usually the div with id miq_timeline.

class cfme.web_ui.jstimelines.**Event**(*element*)
Bases: cfme.web_ui.jstimelines.Object

An event object.

block_info()
Attempts to return a dict with the information from the popup.

close_button = '//div[@class="timeline-event-bubble-title"]/../../div[contains(@style, \'close-button\')]'

data_block = '//div[@class="timeline-event-bubble-title"]/../..//div[@class="timeline-event-bubble-body"]'

image
Returns the image name of an event.

window_loc = '//div[@class="timeline-event-bubble-title"]/../..'

class cfme.web_ui.jstimelines.**Object**(*element*)
Bases: utils.pretty.Pretty

A generic timelines object.

Parameters element – A WebElement for the event.

locate()

pretty_attrs = ['element']

cfme.web_ui.jstimelines.**events**()
A generator yielding all events.

cfme.web_ui.jstimelines.**find_visible_events_for_vm**(*vm_name*)
Finds all events for a given vm.

Parameters vm_name – The vm name.

cfme.web_ui.listaccordion module A set of functions for dealing with accordions in the UI.

Usage:

```
Using Accordions is simply a case of either selecting it to return the element,
or using the built in click method. As shown below::

  acc = web_ui.accordion

  acc.click('Diagnostics')
  acc.is_active('Diagnostics')
```

Note: Inactive links are not available in any way.

class cfme.web_ui.listaccordion.**ListAccordionLink**(*title*, *root=None*)
Bases: utils.pretty.Pretty

Active link in an accordion section

Parameters **title** – The title of the link.

click()
Clicks a link by title.

Parameters **title** – The title of the button to check.

Raises ListAccordionLinkNotFound – when active link is not found.

locate()
Locates an active link.

Returns: An XPATH locator for the element.

pretty_attrs = ['title', 'root']

cfme.web_ui.listaccordion.**click**(*name*)
Clicks an accordion and returns it

Parameters **name** – The name of the accordion.

cfme.web_ui.listaccordion.**get_active_links**(*name*)
Returns all active links in a section specified by name

Parameters **name** – Name of the section

cfme.web_ui.listaccordion.**is_active**(*name*)
Checks if an accordion is currently open

Parameters **name** – The name of the accordion.

Returns: True if the button is depressed, False if not.

cfme.web_ui.listaccordion.**locate**(*name*)
Returns a list-accordion by name

Parameters **name** – The name of the accordion.

Returns: An xpath locator of the selected accordion.

cfme.web_ui.listaccordion.**select**(*name*, *link_title*)
Clicks an active link in accordion section

Parameters

- **name** – Name of the accordion.
- **link_title** – Title of link in expanded accordion section.

cfme.web_ui.menu module

cfme.web_ui.menu.**extend_nav**(*cls*)
A decorator, that when placed on a class will turn it to a nav tree extension.

Takes the original class and "compiles" it into the nav tree in form of lists/dicts that ui_navigate takes.

The classes in the structure are not instantiated during the scavenge process, they serve as a sort of static container of namespaced functions.

Example:

```
@extend_nav
class infra_vms(object):    # This will extend the node infra_vms
    class node_a(object):   # with node a
        def navigate(_):    # that can be reached this way from preceeding one
            pass

        def leaf_location(ctx):  # Leaf location, no other child locations
            pass

        class node_x(object):  # Or an another location that can contain locations
            def navigate(_):
                pass
```

Parameters **cls** – Class to be decorated.

cfme.web_ui.menu.**get_current_toplevel_name**()
Returns text of the currently selected top level menu item.

cfme.web_ui.menu.**is_page_active**(*toplevel*, *secondlevel=None*)

cfme.web_ui.menu.**nav_to_fn**(*toplevel*, *secondlevel=None*, *reset_action=None*)

cfme.web_ui.menu.**reverse_lookup**(*toplevel_path*, *secondlevel_path=None*)
Reverse lookup for navigation destinations defined in this module, based on menu text

Usage:

```
# Returns 'clouds'
reverse_lookup('Clouds')

# Returns 'clouds_providers'
reverse_lookup('Clouds', 'Providers')

# Returns 'automate_import_export'
reverse_lookup('Automate', 'Import / Export')
```

Note: It may be tempting to use this when you don't know the name of a page, e.g.:

go_to(reverse_lookup('Infrastructure', 'Providers'))

Don't do that; use the nav tree name.

---

cfme.web_ui.menu.**visible_pages**()
Return a list of all the menu pages currently visible top- and second-level pages

Mainly useful for RBAC testing

cfme.web_ui.menu.**visible_toplevel_tabs**()

cfme.web_ui.mixins module

cfme.web_ui.mixins.**add_tag**(*tag*, *single_value=False*, *navigate=True*)
cfme.web_ui.mixins.**get_tags**(*tag='My Company Tags'*)

cfme.web_ui.mixins.**left_half_size**()

cfme.web_ui.mixins.**pull_splitter**(*x*)
Pulls the vertical separator between accordion and detail view.

Parameters **x** – Negative values move left, positive right.

cfme.web_ui.mixins.**remove_tag**(*tag*)

cfme.web_ui.multibox module

class cfme.web_ui.multibox.**Async**(*value*)

Bases: utils.category.CategoryBase

class cfme.web_ui.multibox.**MultiBoxSelect**(*unselected*, *selected*, *to_unselected*, *to_selected*, *remove_all=None*, *sync=None*, *async=None*)

Bases: utils.pretty.Pretty

Common UI element for selecting multiple items.

Presence in eg. Control/Explorer/New Policy Profile (for selecting policies)

Parameters

- **unselected** – Locator for the left (unselected) list of items.
- **selected** – Locator for the right (selected) list of items.
- **to_unselected** – Locator for a button which moves items from right to left (unselecting)
- **to_selected** – Locator for a button which moves items from left to right (selecting)
- **remove_all** – If present, locator for a button which unselects all items (Default None)

add(**values*, ***kwargs*)

Mark items for selection and then clicks the button to select them.

Parameters ***values** – Values to select

Keywords:

flush: By using *flush* keyword, the selected items list is flushed prior to selecting new ones

Returns: bool with success.

all_selected

classmethod categorize(*values*, *sync_l*, *async_l*, *dont_care_l*)

Does categorization of values based on their Sync/Async status.

Parameters

- **values** – Values to be categorized.
- **sync_l** – List that will be used for appending the Sync values.
- **async_l** – List that will be used for appending the Async values.
- **dont_care_l** – List that will be used for appending all the other values.

classmethod default()

The most common type of the MultiBoxSelect

Returns: MultiBoxSelect instance

pretty_attrs = ['unselected', 'selected']

remove(**values*)

Mark items for deselection and then clicks the button to deselect them.

Parameters ***values** – Values to deselect

Returns: bool with success.

remove_all()
Flush the list of selected items.

Returns: bool with success.

set_async(**values*)

set_sync(**values*)

class cfme.web_ui.multibox.**SelectItem**
Bases: tuple

SelectItem(sync, value, text)

sync
Alias for field number 0

text
Alias for field number 2

value
Alias for field number 1

class cfme.web_ui.multibox.**Sync**(*value*)
Bases: utils.category.CategoryBase

cfme.web_ui.paginator module A set of functions for dealing with the paginator controls.

cfme.web_ui.paginator.**check_all**()
Returns the Check All locator.

cfme.web_ui.paginator.**click_element**(*el*)
Advance the page until the given element is displayed, and click it

cfme.web_ui.paginator.**find**(*pred*)
Advance the pages until pred (a no-arg function) is true.

cfme.web_ui.paginator.**find_element**(*el*)
Advance the pages until the given element is displayed

cfme.web_ui.paginator.**first**()
Returns the First button locator.

cfme.web_ui.paginator.**last**()
Returns the Last button locator.

cfme.web_ui.paginator.**next**()
Returns the Next button locator.

cfme.web_ui.paginator.**page_controls_exist**()
Simple check to see if page controls exist.

cfme.web_ui.paginator.**pages**()
A generator to facilitate looping over pages

Usage:

```
for page in pages():
    # Do seleniumy things here, like finding and clicking elements
```

cfme.web_ui.paginator.**previous**()
Returns the Previous button locator.

cfme.web_ui.paginator.**rec_end**()
Returns the record set index.

cfme.web_ui.paginator.**rec_offset**()
Returns the first record offset.

cfme.web_ui.paginator.**rec_total**()
Returns the total number of records.

cfme.web_ui.paginator.**reset**()
Reset the paginator to the first page or do nothing if no pages

cfme.web_ui.paginator.**results_per_page**(*num*)
Changes the number of results on a page.

Parameters **num** – Number of results per page

cfme.web_ui.paginator.**sort_by**(*sort*)
Changes the sort by field.

Parameters **sort** – Value to sort by (visible text in select box)

cfme.web_ui.search module This module operates the *Advanced search* box located on multiple pages.

cfme.web_ui.search.**delete_filter**(*cancel=False*)
If possible, deletes the currently loaded filter.

cfme.web_ui.search.**ensure_advanced_search_closed**()
Checks if the advanced search box is open and if it does, closes it.

cfme.web_ui.search.**ensure_advanced_search_open**()
Make sure the advanced search box is opened.

If the advanced search box is closed, open it if it exists (otherwise exception raised). If it is opened but not in the default view (expression editor), close it and open again to ensure the default view is present.

cfme.web_ui.search.**ensure_no_filter_applied**()
If any filter is applied in the quadicon view, it will be disabled.

cfme.web_ui.search.**ensure_normal_search_empty**()
Makes sure that the normal search field is empty.

cfme.web_ui.search.**fill_and_apply_filter**(*expression_program*, *fill_callback=None*, *cancel_on_user_filling=False*)
Fill the filtering expression and apply it

Parameters

- **expression_program** – Expression to fill to the filter.
- **fill_callback** – Function to be called for each asked user input (`_process_user_filling()`).

cfme.web_ui.search.**fill_expression**(*expression_program*)
Wrapper to open the box and fill the expression

Parameters **expression_program** – the expression to be filled.

cfme.web_ui.search.**is_advanced_filter_applied**()
Checks whether any filter is in effect on quadicon view

cfme.web_ui.search.**is_advanced_search_opened**()
Checks whether the advanced search box is currently opened

cfme.web_ui.search.**is_advanced_search_possible**()
Checks for advanced search possibility in the quadicon view

cfme.web_ui.search.**load_and_apply_filter**(*saved_filter=None*, *report_filter=None*, *fill_callback=None*, *cancel_on_user_filling=False*)
Load the filtering expression and apply it

Parameters

- **saved_filter** – *Choose a saved XYZ filter.*
- **report_filter** – *Choose a XYZ report filter.*
- **fill_callback** – Function to be called for each asked user input.

cfme.web_ui.search.**load_filter**(*saved_filter=None*, *report_filter=None*, *cancel=False*)
Load saved filter

Parameters

- **saved_filter** – *Choose a saved XYZ filter*
- **report_filter** – *Choose a XYZ report filter*

cfme.web_ui.search.**normal_search**(*search_term*)
Do normal search via the search bar.

Parameters **search_term** – What to search.

cfme.web_ui.search.**reset_filter**()
Clears the filter expression

cfme.web_ui.search.**save_filter**(*expression_program*, *save_name*, *global_search=False*, *cancel=False*)
Fill the filtering expression and save it

Parameters

- **expression_program** – the expression to be filled.
- **save_name** – Name of the filter to be saved with.
- **global_search** – Whether to check the Global search checkbox.

cfme.web_ui.tabstrip module The tab strip manipulation which appears in Configure / Configuration and possibly other pages.

Usage:

```
import cfme.web_ui.tabstrip as tabs
tabs.select_tab("Authentication")
print is_tab_selected("Authentication")
print get_selected_tab()
```

class cfme.web_ui.tabstrip.**TabStripForm**(*fields=None*, *tab_fields=None*, *identifying_loc=None*, *order=None*, *fields_end=None*)
Bases: cfme.web_ui.Form

A class for interacting with tabstrip-contained Form elements on pages.

This behaves exactly like a Form, but is able to deal with form elements being broken up into tabs, accessible via a tab strip.

Parameters

- **fields** – A list of field name/locator tuples (same as Form implementation)
- **tab_fields** – A dict with tab names as keys, and each key's value being a list of field name/locator tuples. The ordering of fields within a tab is guaranteed (as it is with the normal Form) but the ordering of tabs is not guaranteed by default. If such ordering is needed, tab_fields can be a `collections.OrderedDict`.
- **identifying_loc** – A locator which should be present if the form is visible.
- **order** – If specified, specifies order of the tabs. Can be lower number than number of tabs, remaining values will be complemented.
- **fields_end** – Same as fields, but these are appended at the end of generated fields instead.

Usage:

```
provisioning_form = web_ui.TabStripForm(
    tab_fields={
        'Request': [
            ('email', Input("requester__owner_email")),
            ('first_name', Input("requester__owner_first_name")),
            ('last_name', Input("requester__owner_last_name")),
            ('notes', '//textarea[@id="requester__request_notes"]'),
        ],
        'Catalog': [
            ('instance_name', Input("service__vm_name")),
            ('instance_description', '//textarea[@id="service__vm_description"]'),
        ]
    }
)
```

Each tab's fields will be exposed by their name on the resulting instance just like fields on a Form. Don't use duplicate field names in the `tab_fields` dict.

Forms can then be filled in like so:

```
request_info = {
    'email': 'your@email.com',
    'first_name': 'First',
    'last_name': 'Last',
    'notes': 'Notes about this request',
    'instance_name': 'An instance name',
    'instance_description': 'This is my instance!',
}
web_ui.fill(provisioning_form, request_info)
```

`cfme.web_ui.tabstrip.`**`get_all_tabs`**`()`
Return list of all tabs present.

Returns: `list` of `str` Displayed names.

`cfme.web_ui.tabstrip.`**`get_clickable_tab`**`(`*ident_string*`)`
Returns the relevant tab element that can be clicked on.

Parameters **ident_string** – The text diplayed on the tab.

`cfme.web_ui.tabstrip.`**`get_selected_tab`**`()`
Return currently selected tab.

Returns: str Displayed name

cfme.web_ui.tabstrip.**is_tab_element_selected**(*element*)
Determine whether the passed element is selected.

This function takes the element, climbs to its parent and looks whether the aria-selected attribute contains true. If yes, element is selected.

Parameters **element** – WebElement with the link (a)

Returns: bool

cfme.web_ui.tabstrip.**is_tab_selected**(*ident_string*)
Determine whether the element identified by passed name is selected.

Parameters **ident_string** – Identification string (displayed name) of the tab button.

Returns: bool

cfme.web_ui.tabstrip.**select_tab**(*ident_string*)
Clicks on the tab with text from ident_string.

Clicks only if it's not actually selected.

Parameters **ident_string** – The text displayed on the tab.

cfme.web_ui.toolbar module A set of functions for dealing with the toolbar buttons

The main CFME toolbar is accessed by using the Root and Sub titles of the buttons.

Usage:

```
tb = web_ui.toolbar
tb.select('Configuration', 'Add a New Host')
```

cfme.web_ui.toolbar.**is_active**(*root*)
Checks if a button is currently depressed

Parameters **root** – The root button's name as a string.

Returns: True if the button is depressed, False if not.

cfme.web_ui.toolbar.**is_greyed**(*root*, *sub=None*)
Checks if a button is greyed out.

Parameters **root** – The root button's name as a string.

Returns: True if the button is greyed, False if not.

cfme.web_ui.toolbar.**is_vms_compressed_view**()
Returns whether compressed view is selected or not.

cfme.web_ui.toolbar.**is_vms_details_view**()
Returns whether details view is selected or not.

cfme.web_ui.toolbar.**is_vms_exists_view**()
Returns whether exists mode is selected or not.

cfme.web_ui.toolbar.**is_vms_expanded_view**()
Returns whether expanded view is selected or not.

cfme.web_ui.toolbar.**is_vms_graph_view**()
Returns whether graph view is selected or not.

cfme.web_ui.toolbar.**is_vms_grid_view**()
Returns whether grid view is selected or not.

cfme.web_ui.toolbar.**is_vms_hybrid_view**()
Returns whether hybrid view is selected or not.

cfme.web_ui.toolbar.**is_vms_list_view**()
Returns whether list view is selected or not.

cfme.web_ui.toolbar.**is_vms_tabular_view**()
Returns whether tabular view is selected or not.

cfme.web_ui.toolbar.**is_vms_tile_view**()
Returns whether tile view is selected or not.

cfme.web_ui.toolbar.**refresh**()
Refreshes page, attempts to use cfme refresh button otherwise falls back to browser refresh.

cfme.web_ui.toolbar.**root_loc**(*root*)
Returns the locator of the root button

Parameters **root** – The string name of the button.

Returns: A locator for the root button.

cfme.web_ui.toolbar.**select**(*root*, *sub=None*, *invokes_alert=False*)
Clicks on a button by calling the `click_n_move()` method.

Parameters

- **root** – The root button's name as a string.
- **sub** – The sub button's name as a string. (optional)
- **invokes_alert** – If `True`, then the behaviour is little bit different. After the last click, no ajax wait and no move away is done to be able to operate the alert that appears after click afterwards. Defaults to `False`.

Returns: `True` if everything went smoothly Raises: `cfme.exceptions.ToolbarOptionGreyed`

cfme.web_ui.toolbar.**select_n_move**(*el*)
Clicks an element and then moves the mouse away

This is required because if the button is active and we clicked it, the CSS class doesn't change until the mouse is moved away.

Parameters **el** – The element to click on.

Returns: None

cfme.web_ui.toolbar.**set_vms_compressed_view**()
Set the view to compressed.

cfme.web_ui.toolbar.**set_vms_details_view**()
Set the view to details.

cfme.web_ui.toolbar.**set_vms_exists_view**()
Set the view to exists.

cfme.web_ui.toolbar.**set_vms_expanded_view**()
Set the view to expanded.

cfme.web_ui.toolbar.**set_vms_graph_view**()
Set the view to graph.

cfme.web_ui.toolbar.**set_vms_grid_view**()
Set the view to grid.

cfme.web_ui.toolbar.**set_vms_hybrid_view**()
Set the view to hybrid.

cfme.web_ui.toolbar.**set_vms_list_view**()
Set the view to list.

cfme.web_ui.toolbar.**set_vms_tabular_view**()
Set the view to tabular.

cfme.web_ui.toolbar.**set_vms_tile_view**()
Set the view to tile.

cfme.web_ui.toolbar.**sub_loc**(*sub*)
Returns the locator of the sub button

Parameters **sub** – The string name of the button.

Returns: A locator for the sub button.

Module contents

Provides a number of objects to help with managing certain elements in the CFME UI.

Specifically there are two categories of objects, organizational and elemental.

- **Organizational**
  - Region
  - cfme.web_ui.menu
- **Elemental**
  - AngularCalendarInput
  - ButtonGroup
  - Calendar
  - ColorGroup
  - CheckboxTable
  - CheckboxSelect
  - DHTMLSelect
  - DriftGrid
  - DynamicTable
  - EmailSelectForm
  - Filter
  - Form
  - InfoBlock
  - Input
  - MultiFill
  - Quadicon

- Radio
- ScriptBox
- Select
- ShowingInputs
- SplitCheckboxTable
- SplitTable
- Timelines
- Table
- Tree
- cfme.web_ui.accordion
- cfme.web_ui.cfme_exception
- cfme.web_ui.flash
- cfme.web_ui.form_buttons
- cfme.web_ui.listaccordion
- cfme.web_ui.menu
- cfme.web_ui.paginator
- cfme.web_ui.snmp_form
- cfme.web_ui.tabstrip
- cfme.web_ui.toolbar

class cfme.web_ui.**AngularCalendarInput**(*input_name*, *click_away_element*)
Bases: utils.pretty.Pretty

clear()

clear_button

fill(*value*)

input

locate()

pretty_attrs = ('input_name', 'click_away_element')

class cfme.web_ui.**AngularSelect**(*loc*, *none=None*, *multi=False*)
Bases: object

BUTTON = "//button[@data-id='{}']"

all_options

classes
Combines class from the button and from select.

did

first_selected_option

first_selected_option_text

is_open

locate()

open()

options

select

select_by_value(*value*)

select_by_visible_text(*text*)

class cfme.web_ui.**ButtonGroup**(*key*)
Bases: object

active
Returns the alt tag text of the active button in thr group.

choose(*alt*)
Sets the ButtonGroup to select the button identified by the alt text.

locate()
Moves to the element

locator

locator_base

status(*alt*)
Returns the status of the button identified by the Alt Text of the image.

class cfme.web_ui.**Calendar**(*name*)
Bases: utils.pretty.Pretty

A CFME calendar form field

Calendar fields are readonly, and managed by the dxhtmlCalendar widget. A Calendar field will accept any object that can be coerced into a string, but the value may not match the format expected by dhtmlxCalendar or CFME. For best results, either a datetime.date or datetime.datetime object should be used to create a valid date field.

Parameters **name** – "name" property of the readonly calendar field.

Usage:

```
calendar = web_ui.Calendar("miq_date_1")
web_ui.fill(calendar, date(2000, 1, 1))
web_ui.fill(calendar, '1/1/2001')
```

locate()

class cfme.web_ui.**CheckboxSelect**(*search_root*, *text_access_func=None*)
Bases: utils.pretty.Pretty

Class used for filling those bunches of checkboxes I (@mfalesni) always hated to search for.

Can fill by values, text or both. To search the text for the checkbox, you have 2 choices:

- **If the text can be got from parent's tag (like * <input type="checkbox">blablabla *** where blablabla is the checkbox's description looked up), you can leave the *text_access_func* unfilled.
- **If there is more complicated layout and you don't mind a bit slower operation, you can pass** the text_access_func, which should be like *lambda checkbox_el: get_text_of(checkbox_el)*. The checkbox *WebElement* is passed to it and the description text is the expected output of the function.

Parameters

- **search_root** – Root element for checkbox search
- **text_access_func** – Function returning descriptive text about passed CB element.

check(*values*)
Checking function.

Parameters **values** – Dictionary with key=CB name, value=bool with status.

Look in the function to see.

checkbox_by_id(*id*)
Find checkbox's WebElement by id.

checkbox_by_text(*text*)
Returns checkbox's WebElement by searched by its text.

checkboxes
All checkboxes.

pretty_attrs = ['_root']

select_all()
Selects all checkboxes.

selected_checkboxes
Only selected checkboxes.

selected_values
Only selected checkboxes' values.

unselect_all()
Unselects all checkboxes.

unselected_checkboxes
Only unselected checkboxes.

unselected_values
Only unselected checkboxes' values.

class cfme.web_ui.**CheckboxTable**(*table_locator*, *header_offset=0*, *body_offset=0*, *header_checkbox_locator=None*, *body_checkbox_locator=None*)
Bases: cfme.web_ui.Table

Table with support for checkboxes

Parameters

- **table_locator** – See cfme.web_ui.Table
- **header_checkbox_locator** – Locator of header checkbox (default *None*) Specify in case the header checkbox is not part of the header row
- **body_checkbox_locator** – Locator for checkboxes in body rows
- **header_offset** – See cfme.web_ui.Table
- **body_offset** – See cfme.web_ui.Table

deselect_all()
Deselect all rows using the header checkbox or one by one if not present

deselect_row(*header*, *value*)
Deselect a single row specified by column header and cell value

Parameters

- **header** – See `Table.find_row()`
- **value** – See `Table.find_row()`

Returns: *True* if successful, *False* otherwise

deselect_row_by_cells(*cells*)
Deselect the first row matched by `cells`

Parameters **cells** – See `Table.find_rows_by_cells()`

deselect_rows(*cell_map*)
Deselect multiple rows

Parameters **cell_map** – See `Table.click_cells()`

Raises `NotAllCheckboxesFound` – If some cells were unable to be found

deselect_rows_by_cells(*cells*)
Deselect the rows matched by `cells`

Parameters **cells** – See `Table.find_rows_by_cells()`

deselect_rows_by_indexes(**indexes*)
Deselect rows specified by row indexes (starting with 0)

header_checkbox
Checkbox used to select/deselect all rows

select_all()
Select all rows using the header checkbox or one by one if not present

select_row(*header*, *value*)
Select a single row specified by column header and cell value

Parameters

- **header** – See `Table.find_row()`
- **value** – See `Table.find_row()`

Returns: *True* if successful, *False* otherwise

select_row_by_cells(*cells*)
Select the first row matched by `cells`

Parameters **cells** – See `Table.find_rows_by_cells()`

select_rows(*cell_map*)
Select multiple rows

Parameters **cell_map** – See `Table.click_cells()`

Raises `NotAllCheckboxesFound` – If some cells were unable to be found

select_rows_by_cells(*cells*)
Select the rows matched by `cells`

Parameters **cells** – See `Table.find_rows_by_cells()`

select_rows_by_indexes(**indexes*)
Select rows specified by row indexes (starting with 0)

class cfme.web_ui.**CheckboxTree**(*locator*)
Bases: cfme.web_ui.Tree

Tree that has a checkbox on each node, adds methods to check/uncheck them

check_node(**path*)
Convenience function to check a node

Parameters ***path** – The path as multiple positional string arguments denoting the course to take.

uncheck_node(**path*)
Convenience function to uncheck a node

Parameters ***path** – The path as multiple positional string arguments denoting the course to take.

class cfme.web_ui.**ColorGroup**(*key*)
Bases: object

active
Returns the alt tag text of the active button in thr group.

choose(*color*)
Sets the ColorGroup to select the button identified by the title text.

locate()
Moves to the element

status(*color*)
Returns the status of the color button identified by the Title Text of the image.

class cfme.web_ui.**DHTMLSelect**(*loc*, *multi=False*, *none=None*)
Bases: cfme.fixtures.pytest_selenium.Select

A special Select object for CFME's icon enhanced DHTMLx Select elements.

Parameters **loc** – A locator.

Returns a cfme.web_ui.DHTMLSelect object.

all_selected_options
Returns all selected options.

Note: Since the DHTML select can only have one option selected at a time, we simple return the first element (the only element).

Returns: A Web element.

first_selected_option
Returns the first selected option in the DHTML select

Note: In a DHTML select, there is only one option selectable at a time.

Returns: A webelement.

locate()

options
Returns a list of options of the select as webelements.

Returns: A list of Webelements.

select_by_index(*index*, *_cascade=None*)
Selects an option by index.

Parameters **index** – The select element's option by index.

select_by_value(*value*, *_cascade=None*)
Selects an option by value.

Parameters **value** – The select element's option value.

select_by_visible_text(*text*)
Selects an option by visible text.

Parameters **text** – The select element option's visible text.

class cfme.web_ui.**DriftGrid**(*loc="//div[@id='drift_grid_div']"*)
Bases: utils.pretty.Pretty

Class representing the table (grid) specific to host drift analysis comparison page

cell_indicates_change(*row_text*, *col_index*)
Finds out if a cell, specified by column index and row text, indicates change

Parameters

- **row_text** – Title text of the cell's row
- **col_index** – Column index of the cell

Note: *col_index* of 0 is used for the 2nd actual column in the drift grid, because the 1st column does not contain headers, only row descriptions.

Returns `True` if there is a change present, `False` otherwise

expand_all_sections()
Expands all sections to make the row elements found therein available

get_cell(*row_text*, *col_index*)
Finds cell element of the grid specified by column index and row text

Parameters

- **row_text** – Title text of the cell's row
- **col_index** – Column index of the cell, starting with 0 for 1st data-containing column

---

Note: *col_index* of 0 is used for the 2nd actual column in the drift grid, because the 1st column does not contain headers, only row descriptions.

Returns Selenium element of the cell.

class cfme.web_ui.**DynamicTable**(*root_loc*, *default_row_item=None*)
Bases: utils.pretty.Pretty

A table that can add or remove the rows.

ROWS = ".//tbody/tr[not(contains(@id, 'new_tr'))]"

class Row(*table*, *root*)
Bases: object

inputs

inputs_for_filling

values

DynamicTable.**add_row**(*data*)

DynamicTable.**clear**()

DynamicTable.**click_add**()

DynamicTable.**click_save**()

DynamicTable.**delete_row**(*by*)

DynamicTable.**header_names**

DynamicTable.**pretty_attrs = ('root_loc', 'default_row_item')**

DynamicTable.**rows**

class cfme.web_ui.**EmailSelectForm**
Bases: utils.pretty.Pretty

Class encapsulating the e-mail selector, eg. in Control/Alarms editing.

fields = <cfme.web_ui.Region title=None>

remove_email(*email*)
Remove specified e-mail

Parameters **email** – E-mail to remove

to_emails
Returns list of e-mails that are selected

user_emails
Returns list of e-mail that users inside CFME have so that they can be selected

class cfme.web_ui.**Filter**(*fields=None*, *identifying_loc=None*)
Bases: cfme.web_ui.Form

Filters requests pages

This class inherits Form as its base and adds a few methods to assist in filtering request pages.

Usage:

```
f = Filter(fields=[
    ('type', Select('//select[@id="type_choice"]')),
    ('approved', Input("state_choice__approved")),
    ('denied', Input"state_choice__denied")),
    ('pending_approval', Input("state_choice__pending_approval")),
    ('date', Select('//select[@id="time_period"]')),
    ('reason', Input("reason_text")),
])

f.apply_filter(type="VM Clone", approved=False,
    pending_approval=False, date="Last 24 Hours", reason="Just Because")
```

apply_filter(***kwargs*)
Method to apply a filter.

First resets the filter to default and then applies the filter.

Parameters ****kwargs** – A dictionary of form elements to fill and their values.

buttons = {'reset': '//div[@id="buttons_on"]//a[@title="Reset filter changes"]', 'apply': '//div[@id="buttons_on"]//a[@

default_filter()
Method to reset the filter back to defaults.

reset_filter()
Method to reset the changes to the filter since last applying.

class cfme.web_ui.**Form**(*fields=None*, *identifying_loc=None*)
Bases: cfme.web_ui.Region

A class for interacting with Form elements on pages.

The Form class takes a set of locators and binds them together to create a unified Form object. This Form object has a defined field order so that the user does not have to worry about which order the information is provided. This enables the data to be provided as a dict meaning it can be passed directly from yamls. It inherits the base Region class, meaning that locators can still be referenced in the same way a Region's locators can. You can also add one more field which will be a dict of metadata, determining mostly field validity. See field_valid()

Parameters

- **fields** – A list of field name/locator tuples. The argument not only defines the order of the elements but also which elements comprise part of the form.
- **identifying_loc** – A locator which should be present if the form is visible.

Usage:

```
provider_form = web_ui.Form(
    fields=[
        ('type_select', "//*[@id='server_emstype']"),
        ('name_text', "//*[@id='name']"),
        ('hostname_text', "//*[@id='hostname']"),
        ('ipaddress_text', "//*[@id='ipaddress']"),
        ('amazon_region_select', "//*[@id='hostname']"),
        ('api_port', "//*[@id='port']"),
    ])
```

Forms can then be filled in like so.:

```
provider_info = {
   'type_select': "OpenStack",
   'name_text': "RHOS-01",
   'hostname_text': "RHOS-01",
   'ipaddress_text': "10.0.0.0",
   'api_port': "5000",
}
web_ui.fill(provider_form, provider_info)
```

Note: Using supertuples in a list, although ordered due to the properties of a List, will not overide the field order defined in the Form.

---

field_valid(*field_name*)
Add the validity constraints here.

fill(*fill_data*)

pretty_attrs = ['fields']

class cfme.web_ui.**InfoBlock**(*title*)
Bases: utils.pretty.Pretty

DETAIL = 'detail'

FORM = 'form'

class Member(*ib*, *name*)

Bases: utils.pretty.Pretty

container

element

elements

icon_href

locate()

pair

pair_locator

pretty_attrs = ('name', 'ib')

text

title

InfoBlock.**by_member_icon**(*icon*)

In case you want to find the item by icon in the value field (like OS infra diff.)

classmethod InfoBlock.**container**(*args*, ***kwargs*)

classmethod InfoBlock.**element**(**args*, ***kwargs*)

classmethod InfoBlock.**elements**(**args*, ***kwargs*)

classmethod InfoBlock.**icon_href**(**args*, ***kwargs*)

InfoBlock.**member**(*name*)

InfoBlock.**pretty_attrs = ['title']**

InfoBlock.**root**

classmethod InfoBlock.**text**(**args*, ***kwargs*)

InfoBlock.**type**

class cfme.web_ui.**Input**(**names*, ***kwargs*)

Bases: utils.pretty.Pretty

Class designed to handle things about <input> tags that have name attr in one place.

Also applies on textarea, which is basically input with multiple lines (if it has name).

Parameters ***names** – Possible values (or) of the name attribute.

Keywords:

use_id: Whether to use id instead of name. Useful if there is some input that does not have name attribute present.

angular_help_block

Returns the angular helper text (like 'Required').

locate()

names

pretty_attrs = ['_names', '_use_id']

class cfme.web_ui.**MultiFill**(**fields*)
Bases: object

Class designed to fill the same value to multiple fields

Parameters ***fields** – The fields where the value will be mirrored

class cfme.web_ui.**MultiSelect**(*available_select=None*, *selected_select=None*, *select_arrow=None*, *deselect_arrow=None*)
Bases: cfme.web_ui.Region

Represents a UI widget where there are two select boxes, one with possible selections, and another with selected items. Has two arrow buttons to move items between the two

class cfme.web_ui.**Quadicon**(*name*, *qtype=None*)
Bases: utils.pretty.Pretty

Represents a single quadruple icon in the CFME UI.

A Quadicon contains multiple quadrants. These are accessed via attributes. The qtype is currently one of the following and determines which attribute names are present. They are mapped internally and can be reassigned easily if the UI changes.

A Quadicon is used by defining the name of the icon and the type. After that, it can be used to obtain the locator of the Quadicon, or query its quadrants, via attributes.

Parameters

- **name** – The label of the icon.
- **qtype** – The type of the quad icon. By default it is None, therefore plain quad without any retrievable data usable for selecting/clicking.

Usage:

```
qi = web_ui.Quadicon('hostname.local', 'host')
qi.creds
click(qi)
```

Known Quadicon Types and Attributes

•**host** - *from the infra/host page* - has quads:

- 1.**no_vm** - Number of VMs
- 2.**state** - The current state of the host
- 3.**vendor** - The vendor of the host
- 4.**creds** - If the creds are valid

•**infra_prov** - *from the infra/providers page* - has quads:

- 1.**no_host** - Number of hosts
- 2.*Blank*
- 3.**vendor** - The vendor of the provider
- 4.**creds** - If the creds are valid

- **vm** - *from the infra/virtual_machines page* - has quads:
  - 1.**os** - The OS of the vm
  - 2.**state** - The current state of the vm
  - 3.**vendor** - The vendor of the vm's host
  - 4.**no_snapshot** - The number of snapshots
  - 7.**policy** - The state of the policy
- **cloud_prov** - *from the cloud/providers page* - has quads:
  - 1.**no_instance** - Number of instances
  - 2.**no_image** - Number of machine images
  - 3.**vendor** - The vendor of the provider
  - 4.**creds** - If the creds are valid
- **instance** - *from the cloud/instances page* - has quads:
  - 1.**os** - The OS of the instance
  - 2.**state** - The current state of the instance
  - 3.**vendor** - The vendor of the instance's host
  - 4.**no_snapshot** - The number of snapshots
  - 7.**policy** - The state of the policy
- **datastore** - *from the infra/datastores page* - has quads:
  - 1.**type** - File system type
  - 2.**no_vm** - Number of VMs
  - 3.**no_host** - Number of hosts
  - 4.**avail_space** - Available space
- **repository** - *from the infra/repositories page* - has no quads
- **cluster** - *from the infra/cluster page* - has no quads
- **resource_pool** - *from the infra/resource_pool page* - has no quads
- **stack** - *from the clouds/stacks page* - has no quads

Returns: A `Quadicon` object.

QUADS = {'resource_pool': {}, 'template': {'vendor': ('c', 'img'), 'state': ('b', 'img'), 'os': ('a', 'img'), 'no_snapshot': ('d'

classmethod all(*qtype=None*, *this_page=False*)

Allows iteration over Quadicons.

Parameters

- **qtype** – Quadicon type. Refer to the constructor for reference.
- **this_page** – Whether to look for Quadicons only on current page (do not list pages).

Returns: `list` of `Quadicon`

any_present = False

check_for_single_quadrant_icon
Checks if the quad icon is a single quadrant icon.

checkbox()
Returns: a locator for the internal checkbox for the quadicon

exists

classmethod first(*qtype=None*)

static get_first_quad_title()

locate()
Returns: a locator for the quadicon anchor

name
Returns name of the quadicon.

pretty_attrs = ['_name', '_qtype']

qtype

static select_first_quad()

class cfme.web_ui.**Radio**(**names*, ***kwargs*)
Bases: cfme.web_ui.Input

A class for Radio button groups

Radio allows the usage of HTML radio elements without resorting to previous practice of iterating over elements to find the value. The name of the radio group is passed and then when choices are required, the locator is built.

Parameters **name** – The HTML elements `name` attribute that identifies a group of radio buttons.

Usage:

```
radio = Radio("schedule__schedule_type")
```

A specific radio element can then be returned by running the following:

```
el = radio.choice('immediately')
click(el)
```

The `Radio` object can be reused over and over with repeated calls to the `Radio.choice()` method.

choice(*val*)
Returns the locator for a choice

Parameters **val** – A string representing the `value` attribute of the specific radio element.

Returns: A string containing the XPATH of the specific radio element.

observer_wait(*val*)

class cfme.web_ui.**Region**(*locators=None*, *title=None*, *identifying_loc=None*, ***kwargs*)
Bases: utils.pretty.Pretty

Base class for all UI regions/pages

Parameters

- **locators** – A dict of locator objects for the given region
- **title** – A string containing the title of the page, or a versioned dict of page title strings

- **identifying_loc** – Single locator key from locators used by `Region.is_displayed()` to check if the region is currently visible

Usage:

```
page = Region(locators={
    'configuration_button': (By.CSS_SELECTOR, "div.dhx_toolbar_btn[title='Configuration']"),
    'discover_button': (By.CSS_SELECTOR,
        "tr[title='Discover Cloud Providers']>td.td_btn_txt>" "div.btn_sel_text")
    },
    title='Cloud Providers',
    identifying_loc='discover_button'
)
```

The elements can then accessed like so:

```
page.configuration_button
```

Locator attributes will return the locator tuple for that particular element, and can be passed on to other functions, such as `element()` and `click()`.

Note: When specifying a region title, omit the "Cloudforms Management Engine: " or "ManageIQ: " prefix. They're included on every page, and different for the two versions of the appliance, and `is_displayed()` strips them off before checking for equality.

---

is_displayed()
Checks to see if the region is currently displayed.

Returns: A boolean describing if the region is currently displayed

pretty_attrs = ['title']

title

class cfme.web_ui.**ScriptBox**(*name=None*, *ta_locator="//textarea[contains(@id, 'method_data')]"*)
Bases: utils.pretty.Pretty

Represents a script box as is present on the customization templates pages. This box has to be activated before keys can be sent. Since this can't be done until the box element is visible, and some dropdowns change the element, it must be activated "inline".

Args:

get_value()

name

pretty_attrs = ['locator']

class cfme.web_ui.**ShowingInputs**(**locators*, ***kwargs*)
Bases: utils.pretty.Pretty

This class abstracts out as a container of inputs, that appear after preceeding was filled.

Parameters ***locators** – In-order-of-display specification of locators.

Keywords: min_values: How many values are required (Default: 0)

pretty_attrs = ['locators', 'min_values']

zip(*with_values*)

class cfme.web_ui.**SortTable**(*table_locator*, *header_offset=0*, *body_offset=0*)
Bases: cfme.web_ui.Table

This table is the same as Table, but with added sorting functionality.

click_header_cell(*text*)
Clicks on the header to change sorting conditions.

Parameters **text** – Header cell text.

sort_by(*header*, *order*)
Sorts the table by given conditions

Parameters

- **header** – Text of the header cell to use for sorting.
- **order** – ascending or descending

sort_order
Return order.

Returns: 'ascending' or 'descending'

sorted_by
Return column name what is used for sorting now.

class cfme.web_ui.**SplitCheckboxTable**(*header_data*, *body_data*, *header_checkbox_locator=None*, *body_checkbox_locator=None*)
Bases: cfme.web_ui.SplitTable, cfme.web_ui.CheckboxTable

SplitTable with support for checkboxes

Parameters

- **header_data** – See cfme.web_ui.SplitTable
- **body_data** – See cfme.web_ui.SplitTable
- **header_checkbox_locator** – See cfme.web_ui.CheckboxTable
- **body_checkbox_locator** – See cfme.web_ui.CheckboxTable
- **header_offset** – See cfme.web_ui.Table
- **body_offset** – See cfme.web_ui.Table

class cfme.web_ui.**SplitTable**(*header_data*, *body_data*)
Bases: cfme.web_ui.Table

Table that supports the header and body rows being in separate tables

Parameters

- **header_data** – A tuple, containing an element locator and an offset value. These point to the container of the header row. The offset is used in case there is a padding row above the header, or in the case that the header and the body are contained inside the same table element.
- **body_data** – A tuple, containing an element locator and an offset value. These point to the container of the body rows. The offset is used in case there is a padding row above the body rows, or in the case that the header and the body are contained inside the same table element.

Usage:

```
table = SplitTable(header_data=('//div[@id="header_table"]//table/tbody', 0),
    body_data=('//div[@id="body_table"]//table/tbody', 1))
```

The HTML code for a split table looks something like this:

```
<div id="prov_pxe_img_div">
  <table id="header_table">
      <tbody>
          <tr>
              <td>Name</td>
              <td>Animal</td>
              <td>Size</td>
          </tr>
      </tbody>
  </table>
  <table id="body_table">
      <tbody>
          <tr>
              <td>Useless</td>
              <td>Padding</td>
              <td>Row</td>
          </tr>
          <tr>
              <td>John</td>
              <td>Monkey</td>
              <td>Small</td>
          </tr>
          <tr>
              <td>Mike</td>
              <td>Tiger</td>
              <td>Large</td>
          </tr>
      </tbody>
  </table>
</div>
```

Note the use of the offset to skip the "Useless Padding Row" in `body_data`. Most split tables require an offset for both the heading and body rows.

body
Property representing the element that contains body rows

header_row
Property representing the `<tr>` element that contains header cells

locate()

class cfme.web_ui.**Table**(*table_locator*, *header_offset=0*, *body_offset=0*)
Bases: utils.pretty.Pretty

Helper class for Table/List objects

Turns CFME custom Table/Lists into iterable objects using a generator.

Parameters

- **table_locator** – locator pointing to a table element with child thead and tbody elements representing that table's header and body row containers

- **header_offset** – In the case of a padding table row above the header, the row offset can be used to skip rows in `<thead>` to locate the correct header row. This offset is 1-indexed, not 0-indexed, so an offset of 1 is the first child row element
- **body_offset** – In the case of a padding table row above the body rows, the row offset can be used to skip rows in `<ttbody>` to locate the correct header row. This offset is 1-indexed, not 0-indexed, so an offset of 1 is the first child row element

header_indexes
A dict of header names related to their int index as a column.

Usage:

```
table = Table('//div[@id="prov_pxe_img_div"]//table')
```

The HTML code for the table looks something like this:

```
<div id="prov_pxe_img_div">
  <table>
      <thead>
          <tr>
              <th>Name</th>
              <th>Animal</th>
              <th>Size</th>
          </tr>
      </thead>
      <tbody>
          <tr>
              <td>John</td>
              <td>Monkey</td>
              <td>Small</td>
          </tr>
          <tr>
              <td>Mike</td>
              <td>Tiger</td>
              <td>Large</td>
          </tr>
      </tbody>
  </table>
</div>
```

We can now click on an element in the list like so, by providing the column name and the value that we are searching for:

```
table.click_cell('name', 'Mike')
```

We can also perform the same, by using the index of the column, like so:

```
table.click_cell(1, 'Tiger')
```

Additionally, the rows of a table can be iterated over, and that row's columns can be accessed by name or index (left to right, 0-index):

```
for row in table.rows()
    # Get the first cell in the row
    row[0]
    # Get the row's contents for the column with header 'Row Name'
    # All of these will work, though the first is preferred
    row.row_name, row['row_name'], row['Row Name']
```

When doing bulk opererations, such as selecting rows in a table based on their content, the `*_by_cells` methods are able to find matching row much more quickly than iterating, as the work can be done with fewer selenium calls.

- `find_rows_by_cells()`
- `find_row_by_cells()`
- `click_rows_by_cells()`
- `click_row_by_cells()`

Note: A table is defined by the containers of the header and data areas, and offsets to them. This allows a table to include one or more padding rows above the header row. In the example above, there is no padding row, as our offset values are set to 0.

---

class Row (*row_element*, *parent_table*)

Bases: `utils.pretty.Pretty`

An object representing a row in a Table.

The Row object returns a dymanically addressable attribute space so that the tables headers are automatically generated.

Parameters

- **row_element** – A table row `WebElement`
- **parent_table** – `Table` containing `row_element`

Notes

Attributes are dynamically generated. The index/key accessor is more flexible than the attr accessor, as it can operate on int indices and header names.

columns

A list of WebElements corresponding to the `<td>` elements in this row

locate()

pretty_attrs = ['row_element', 'table']

`Table`.**body**

Property representing the `<tbody>` element that contains body rows

`Table`.**click_cell** (*header*, *value*)

Clicks on a cell defined in the row.

Uses the header identifier and a value to determine which cell to click on.

Parameters

- **header** – A string or int, describing which column to inspect.
- **value** – The value to be compared when trying to identify the correct cell to click the cell in.

Returns: `True` if item was found and clicked, else `False`.

`Table`.**click_cells** (*cell_map*)

Submits multiple cells to be clicked on

Parameters **cell_map** – A mapping of header names and values, representing cells to click. As an example, `{'name': ['wing', 'nut']}, {'age': ['12']}` would click on the cells which had `wing` and `nut` in the name column and `12` in the age column. The yaml example for this would be as follows:

```
list_items:
    name:
        - wing
        - nut
    age:
        - 12
```

Raises `NotAllItemsClicked` – If some cells were unable to be found.

`Table.`**`click_row_by_cells`**(*cells*, *click_column=None*, *partial_check=False*)
Click the cell at `click_column` in the first row matched by `cells`

Parameters

- **cells** – See `Table.find_rows_by_cells()`
- **click_column** – See `Table.click_rows_by_cells()`

`Table.`**`click_rows_by_cells`**(*cells*, *click_column=None*, *partial_check=False*)
Click the cell at `click_column` in the rows matched by `cells`

Parameters

- **cells** – See `Table.find_rows_by_cells()`
- **click_column** – Which column in the row to click, defaults to None, which will attempt to click the row element

---

Note: The value of click_column can be a string or an int, and will be passed directly to the item accessor (`__getitem__`) for `Table.Row`

`Table.`**`create_row_from_element`**(*row_element*)
Given a row element in this table, create a `Table.Row`

Parameters **row_element** – A table row (`<tr>`) WebElement representing a row in this table.

Returns: A `Table.Row` for `row_element`

`Table.`**`find_cell`**(*header*, *value*)
Finds an item in the Table by iterating through each visible item, this work used to be done by the :py:meth::*click_cell* method but has not been abstracted out to be called separately.

Parameters

- **header** – A string or int, describing which column to inspect.
- **value** – The value to be compared when trying to identify the correct cell to click.

Returns: WebElement of the element if item was found, else `None`.

`Table.`**`find_row`**(*header*, *value*)
Finds a row in the Table by iterating through each visible item.

Parameters

- **header** – A string or int, describing which column to inspect.
- **value** – The value to be compared when trying to identify the correct row to return.

Returns Table.Row containing the requested cell, else None.

Table.**find_row_by_cells**(*cells*, *partial_check=False*)
Find the first row containing cells

Parameters **cells** – See Table.find_rows_by_cells()

Returns: The first matching row found, or None if no matching row was found

Table.**find_rows_by_cells**(*cells*, *partial_check=False*)
A fast row finder, based on cell content.

Parameters **cells** – A dict of header: value pairs or a sequence of nested (header, value) pairs.

Returns: A list of containing Table.Row objects whose contents match all of the header: value pairs in cells

Table.**header_indexes**
Dictionary of header name – column index for this table's rows

Derived from headers

Table.**header_row**
Property representing the <tr> element that contains header cells

Table.**headers**
List of <td> or <th> elements in header_row

Table.**locate**()

Table.**pretty_attrs = ['_loc']**

Table.**rows**()
A generator method holding the Row objects

This generator yields Row objects starting at the first data row.

Yields Table.Row object corresponding to the next row in the table.

class cfme.web_ui.**Timelines**(*loc*)
Bases: utils.pretty.Pretty

A Timelines object represents the Timelines widget in CFME

Parameters **loc** – A locator for the Timelines element, usually the div with id miq_timeline.

class Event(*element*)
Bases: cfme.web_ui.Object

An event object.

block_info()
Attempts to return a dict with the information from the popup.

close_block()
Closes the events info block.

close_button = '//div[@class="timeline-event-bubble-title"]/../../div[contains(@style, \'close-button\')]'

data_block = '//div[@class="timeline-event-bubble-title"]/../..//div[@class="timeline-event-bubble-body"]'

image
Returns the image name of an event.

open_block()
Opens the events info block.

window_loc = '//div[@class="timeline-event-bubble-title"]/../..'

class Timelines.**Marker**(*element*)
Bases: cfme.web_ui.Object

A proxied object in case it needs more methods further down the line.

class Timelines.**Object**(*element*)
Bases: utils.pretty.Pretty

A generic timelines object.

Parameters **element** – A WebElement for the event.

locate()

pretty_attrs = ['element']

Timelines.**events**()
A generator yielding all events.

Timelines.**find_first_event_in_range**()
Finds the first event on screen.

Timelines.**find_first_marker_in_range**()
Finds the first marker on screen.

Timelines.**find_visible_events_for_vm**(*vm_name*)
Finds all events for a given vm.

Parameters **vm_name** – The vm name.

Timelines.**markers**()
A generator yielding all markers.

Timelines.**pretty_attrs = ['element']**

Timelines.**visible_events**()
A generator giving all visible events.

class cfme.web_ui.**Tree**(*locator*)
Bases: utils.pretty.Pretty

A class directed at CFME Tree elements

The Tree class aims to deal with all kinds of CFME trees, at time of writing there are two distinct types. One which uses <table> elements and another which uses <ul> elements.

Parameters **locator** – This is a locator object pointing to either the outer <table> or <ul> element which contains the rest of the table.

Returns: A Tree object.

A Tree object is set up by using a locator which contains the node elements. This element will usually be a <ul> in the case of a Dynatree, or a <table> in the case of a Legacy tree.

Usage:

```
tree = web_ui.Tree((By.XPATH, '//table//tr[@title="Datastore"]/../..'))
```

The path can then be navigated to return the last object in the path list, like so:

```
tree.click_path('Automation', 'VM Lifecycle Management (VMLifecycle)',
    'VM Migrate (Migrate)')
```

Each path element will be expanded along the way, but will not be clicked.

When used in a `Form`, a list of path tuples is expected in the form fill data. The paths will be passed individually to `Tree.check_node()`:

```
form = Form(fields=[
    ('tree_field', List(locator)),
])

form_fill_data = {
    'tree_field': [
        ('Tree Node', 'Value'),
        ('Tree Node', 'Branch Node', 'Value'),
    ]
]
```

Note: For legacy trees, the first element is often ignored as it is not a proper tree element ie. in Automate->Explorer the Datastore element doesn't really exist, so we omit it from the click map.

Legacy trees rely on a complex `<table><tbody><tr><td>` setup. We class a `<tbody>` as a node.

Note: Dynatrees, rely on a `<ul><li>` setup. We class a `<li>` as a node.

classmethod browse(*tree*, **path*)

Browse through tree via path.

If node not found, raises exception. If the browsing reached leaf(str), returns True if also the step was last, otherwise False. If the result of the path is a subtree, it is returned.

Parameters

- **tree** – List with tree.
- ***path** – Path to browse.

click_path(**path*)

Exposes a path and then clicks it.

Parameters ***path** – The path as multiple positional string arguments denoting the course to take.

Returns: The leaf web element.

expand_path(**path*)

Clicks through a series of elements in a path.

Clicks through a tree, by expanding the levels in a single straight path and returns the final element without clicking it.

Parameters ***path** – The path as multiple positional string arguments denoting the course to take.

Returns: The element at the leaf of the tree.

Raises

- `cfme.exceptions.CandidateNotFound` – A candidate in the tree could not be found to continue down the path.

- cfme.exceptions.TreeTypeUnknown – A locator was passed to the constructor which does not correspond to a known tree type.

find_path_to(*target*)
Method used to look up the exact path to an item we know only by its regexp or partial description.

Expands whole tree during the execution.

Parameters **target** – Item searched for. Can be regexp made by re.compile, otherwise it is taken as a string for *in* matching.

Returns: list with path to that item.

classmethod flatten_level(*tree*)
Extracts just node names from current tree (top).

It makes:

```
["asd", "fgh", ("ijk", [...]), ("lmn", [...])]
```

to

```
["asd", "fgh", "ijk", "lmn"]
```

Useful for checking of contents of current tree level

node_element(*node_name*, *parent*)

node_root_element(*node_name*, *parent*)

nodes_root_elements(*parent*)

pretty_attrs = ['locator']

read_contents(*parent=None*, *unexpand=False*)
Reads complete contents of the tree recursively.

Tree is represented as a list. If the item in the list is string, it is leaf element and it is its name. If the item is a tuple, first element of the tuple is the name and second element is the subtree (list).

Parameters

- **parent** – Starting element, used during recursion
- **unexpand** – Whether it should unexpand the expanded levels to original state.

Returns: Tree in format mentioned in description

class cfme.web_ui.**UpDownSelect**(*select_loc*, *up_loc*, *down_loc*)
Bases: cfme.web_ui.Region

Multiselect with two arrows (up/down) next to it. Eg. in AE/Domain priority selection.

Parameters

- **select_loc** – Locator for the select box (without Select element wrapping)
- **up_loc** – Locator of the Move Up arrow.
- **down_loc** – Locator with Move Down arrow.

get_items()

move_bottom(*item*)

move_down(*item*)

move_top(*item*)

move_up(*item*)

cfme.web_ui.**fill_callable**(*f*, *val*)
Fill in a Callable by just calling it with the value, allow for arbitrary actions

cfme.web_ui.**fill_cb_select_bool**(*select*, *all_state*)

cfme.web_ui.**fill_cb_select_dictlist**(*select*, *dictlist*)

cfme.web_ui.**fill_cb_select_set**(*select*, *names*)

cfme.web_ui.**fill_cb_select_string**(*select*, *cb*)

cfme.web_ui.**fill_checkbox**(*cb*, *val*)

cfme.web_ui.**fill_click**(*el*, *val*)
Click only when given a truthy value

cfme.web_ui.**fill_email_select_form**(*form*, *emails*)

cfme.web_ui.**fill_file**(*fd*, *val*)

cfme.web_ui.**fill_multiselect**(*ms*, *items*)

cfme.web_ui.**fill_password**(*pwbox*, *password*)

cfme.web_ui.**fill_scriptbox**(*sb*, *script*)
This function now clears and sets the ScriptBox.

cfme.web_ui.**fill_select**(*slist*, *val*)

cfme.web_ui.**fill_select_tag**(*select*, *value*)

cfme.web_ui.**fill_text**(*textbox*, *val*)

cfme.web_ui.**get_context_current_page**()
Returns the current page name

Returns: A string containing the current page name

cfme.web_ui.**select_dhtml**(*dhtml*, *s*)

cfme.web_ui.**select_multiselect**(*ms*, *values*)

cfme.web_ui.**table_in_object**(*table_title*)
If you want to point to tables inside object view, this is what you want to use.

Works both on down- and upstream.

Parameters **table_title** – Text in *p* element preceeding the table

Returns: XPath locator for the desired table.

3.1.2 Submodules

cfme.dashboard module

Provides functions to manipulate the dashboard landing page.

var page A cfme.web_ui.Region holding locators on the dashboard page

class cfme.dashboard.**BaseWidgetContent**(*widget_box_id*)
Bases: utils.pretty.Pretty

data

pretty_attrs = ['widget_box_id']

class cfme.dashboard.**RSSWidgetContent**(*widget_box_id*)
Bases: cfme.dashboard.BaseWidgetContent

data

class cfme.dashboard.**ReportWidgetContent**(*widget_box_id*)
Bases: cfme.dashboard.BaseWidgetContent

data

class cfme.dashboard.**Widget**(*div_id*)
Bases: utils.pretty.Pretty

classmethod all()
Returns objects with all Widgets currently present.

classmethod by_name(*name*)
Returns Widget with specified name.

classmethod by_type(*content_type*)
Returns Widget with specified content_type.

can_zoom
Can this Widget be zoomed?

close_dropdown_menu()

classmethod close_zoom()

content

content_type

footer

classmethod get_zoomed_name()

is_dropdown_menu_opened

is_minimized

classmethod is_zoomed()

minimize()
Minimize this Widget.

name

newer_version

open_dropdown_menu()

pretty_attrs = ['_div_id']

remove(*cancel=False*)
Remove this Widget.

restore()
Return the Widget back from minimalization.

time_next

time_updated

zoom()
Zoom this Widget.

cfme.dashboard.**dashboards**()
Returns a generator that iterates through the available dashboards

cfme.dashboard.**get_csrf_token**()
Retuns current CSRF token.

Returns: Current CSRF token.

cfme.dashboard.**reset_widgets**(*cancel=False*)
Resets the widgets on the dashboard page.

Parameters **cancel** – Set whether to accept the popup confirmation box. Defaults to `False`.

cfme.dashboard.**set_csrf_token**(*csrf_token*)
Changing the CSRF Token on the fly via the DOM by iterating over the meta tags

Parameters **csrf_token** – Token to set as the CSRF token.

cfme.exceptions module

Provides custom exceptions for the `cfme` module.

exception cfme.exceptions.**AccordionItemNotFound**
Bases: cfme.exceptions.CFMEException

Raised when it's not possible to locate and accordion item.

exception cfme.exceptions.**AddProviderError**
Bases: cfme.exceptions.CFMEException

exception cfme.exceptions.**AuthModeUnknown**
Bases: cfme.exceptions.CFMEException

Raised if an invalid authenctication mode is passed to cfme.configure.configuration.set_auth_mode()

exception cfme.exceptions.**AutomateImportError**
Bases: cfme.exceptions.CFMEException

Raised by scripts dealing with Automate when importing automate XML fails

exception cfme.exceptions.**BlockTypeUnknown**
Bases: cfme.exceptions.CFMEException

Raised if the block type requested to cfme.web_ui.InfoBlock.

exception cfme.exceptions.**CFMEException**
Bases: exceptions.Exception

Base class for exceptions in the CFME tree

Used to easily catch errors of our own making, versus errors from external libraries.

exception cfme.exceptions.**CFMEExceptionOccured**
Bases: cfme.exceptions.CFMEException

Raised by cfme.web_ui.cfme_exception.assert_no_cfme_exception() when there is a Rails exception currently on page.

exception cfme.exceptions.**CandidateNotFound**
Bases: cfme.exceptions.CFMEException

Raised if there is no candidate found whilst trying to traverse a tree in cfme.web_ui.Tree.click_path().

exception cfme.exceptions.**CannotContinueWithNavigation**
Bases: cfme.exceptions.CFMEException

Used for telling force_navigate that is not possible to continue with navigation.

Raising it will recycle the browser, therefore refresh the session. If you pass a string to the constructor, it will be written to the log.

exception cfme.exceptions.**CannotScrollException**
Bases: cfme.exceptions.CFMEException

Raised when even during the heaviest workarounds for scrolling failure comes.

exception cfme.exceptions.**ElementOrBlockNotFound**
Bases: cfme.exceptions.CFMEException

Raised if an Element or a Block is not found whilst locating in cfme.web_ui.InfoBlock().

exception cfme.exceptions.**FlashMessageException**
Bases: cfme.exceptions.CFMEException

Raised by functions in cfme.web_ui.flash

skip_and_log(*message='Skipping due to flash message'*)

exception cfme.exceptions.**HostNotFound**
Bases: cfme.exceptions.CFMEException

Raised if a specific host cannot be found in UI.

exception cfme.exceptions.**HostNotRemoved**
Bases: cfme.exceptions.CFMEException

Raised when utils.mgmt_system fails to remove host from cluster

exception cfme.exceptions.**HostStatsNotContains**
Bases: cfme.exceptions.CFMEException

Raised if the hosts information does not contain the specified key whilst running cfme.cloud.provider.Provider.do_stats_match().

exception cfme.exceptions.**InstanceNotFound**
Bases: cfme.exceptions.VmOrInstanceNotFound

Raised if a specific instance cannot be found.

exception cfme.exceptions.**ListAccordionLinkNotFound**
Bases: cfme.exceptions.CFMEException

Raised when active link containing specific text could not be found in expended cfme.web_ui.listaccordion content section.

exception cfme.exceptions.**NavigationError**(*page_name*)
Bases: cfme.exceptions.CFMEException

Raised when the pytest.sel.go_to function is unable to navigate to the requested page.

exception cfme.exceptions.**NoElementsInsideValue**
Bases: cfme.exceptions.CFMEException

Raised if the value part of key/value contains no elements during cfme.web_ui.InfoBlock.get_el_or_els().

exception cfme.exceptions.**NotAllCheckboxesFound**(*failed_selects*)
Bases: cfme.exceptions.CFMEException

Raised if not all the checkboxes could be found during e.g. cfme.web_ui.CheckboxTable.select_rows() and other methods of this class.

exception cfme.exceptions.**NotAllItemsClicked**(*failed_clicks*)
Bases: cfme.exceptions.CFMEException

Raised if not all the items could be clicked during cfme.web_ui.Table.click_cell().

exception cfme.exceptions.**OptionNotAvailable**
Bases: cfme.exceptions.CFMEException

Raised if a specified option is not available.

exception cfme.exceptions.**ProviderHasNoKey**
Bases: cfme.exceptions.CFMEException

Raised if the cfme.cloud.provider.Provider.get_mgmt_system() method is called but the Provider instance has no key.

exception cfme.exceptions.**ProviderHasNoProperty**
Bases: cfme.exceptions.CFMEException

Raised if the provider does not have the property requested whilst running cfme.cloud.provider.Provider.do_stats_match().

exception cfme.exceptions.**RequestException**
Bases: cfme.exceptions.CFMEException

Raised if a request was not found or multiple rows matched during _request functions in cfme.services.requests

exception cfme.exceptions.**ScheduleNotFound**
Bases: cfme.exceptions.CFMEException

Raised if a schedule was not found in cfme.configure.configuration.Schedule.delete_by_name()

exception cfme.exceptions.**StorageManagerNotFound**
Bases: cfme.exceptions.CFMEException

Raised when a Storage Manager is not found

exception cfme.exceptions.**TemplateNotFound**
Bases: cfme.exceptions.CFMEException

Raised if a specific Template cannot be found.

exception cfme.exceptions.**ToolbarOptionGreyedOrUnavailable**
Bases: cfme.exceptions.CFMEException

Raised when toolbar wants to click item that is greyed or unavailable

exception cfme.exceptions.**TreeTypeUnknown**
Bases: cfme.exceptions.CFMEException

Raised if the tree type is known whilst detection in cfme.web_ui.Tree

exception cfme.exceptions.**UnidentifiableTagType**
Bases: cfme.exceptions.CFMEException

Raised if a tag type is not identifiable when processing a form in cfme.web_ui.Form.fill_fields().

exception cfme.exceptions.**UnknownProviderType**
Bases: cfme.exceptions.CFMEException

Raised when the passed provider or provider type is not known or usable in given context e.g. when getting a provider from yaml and the provider type doesn't match any of known types or when an infra provider is passed to the cloud's instance_factory method

exception cfme.exceptions.**VmNotFound**
Bases: cfme.exceptions.VmOrInstanceNotFound

Raised if a specific VM cannot be found.

exception cfme.exceptions.**VmNotFoundViaIP**
Bases: cfme.exceptions.CFMEException

Raised if a specific VM cannot be found.

exception cfme.exceptions.**VmOrInstanceNotFound**
Bases: cfme.exceptions.CFMEException

exception cfme.exceptions.**ZoneNotFound**
Bases: cfme.exceptions.CFMEException

Raised when a specific Zone cannot be found in the method cfme.configure.configuration.

cfme.js module

cfme.login module

Provides functions to login as any user

Also provides a convenience function for logging in as admin using the credentials in the cfme yamls.

var page A cfme.web_ui.Region holding locators on the login page

cfme.login.**clear_fields**()
clears all form fields

cfme.login.**click_on_login**()
Convenience internal function to click the login locator submit button.

cfme.login.**close_password_update_form**()
Goes back to main login form on login page

cfme.login.**current_full_name**()
Returns the current username.

Returns: the current username.

cfme.login.**current_user**()

cfme.login.**current_username**()

cfme.login.**fill_login_fields**(*username*, *password*)
Fills in login information without submitting the form

cfme.login.**logged_in**()

cfme.login.**login**(*user*, *submit_method=<function _js_auth_fn at 0x7f52eb1cade8>*)
Login to CFME with the given username and password. Optionally, submit_method can be press_enter_after_password to use the enter key to login, rather than clicking the button.

Parameters

- **user** – The username to fill in the username field.
- **password** – The password to fill in the password field.

- **submit_method** – A function to call after the username and password have been input.

Raises `RuntimeError` – If the login fails, ie. if a flash message appears

`cfme.login.`**`login_admin`**(***kwargs*)
Convenience function to log into CFME using the admin credentials from the yamls.

Parameters **kwargs** – A dict of keyword arguments to supply to the `login()` method.

`cfme.login.`**`logout`**()
Logs out of CFME.

`cfme.login.`**`press_enter_after_password`**()
Convenience function to send a carriange return at the end of the password field.

`cfme.login.`**`show_password_update_form`**()
Shows the password update form

`cfme.login.`**`update_password`**(*username*, *password*, *new_password*, *verify_password=None*, *submit_method=<function click_on_login at 0x7f52eb1ca938>*)
Changes user password

cfme.provisioning module

cfme.roles module

`cfme.roles.`**`group_data`**()

3.1.3 Module contents

class `cfme.`**`Credential`**(*principal=None*, *secret=None*, *verify_secret=None*, ***ignore*)
Bases: `utils.pretty.Pretty`

A class to fill in credentials

Parameters

- **principal** – Something
- **secret** – Something
- **verify_secret** – Something

`pretty_attrs` **= ['principal', 'secret']**

3.2 fixtures package

3.2.1 Subpackages

fixtures.parallelizer package

Submodules

fixtures.parallelizer.hooks module

fixtures.parallelizer.parallelizer_tester module

fixtures.parallelizer.remote module

Module contents

3.2.2 Submodules

fixtures.artifactor_plugin module

An example config:

```
artifactor:
    log_dir: /home/test/workspace/cfme_tests/artiout
    per_run: test #test, run, None
    reuse_dir: True
    squash_exceptions: False
    threaded: False
    server_address: 127.0.0.1
    server_port: 21212
    server_enabled: True
    plugins:
```

`log_dir` is the destination for all artifacts

`per_run` denotes if the test artifacts should be group by run, test, or None

`reuse_dir` if this is False and Artifactor comes across a dir that has already been used, it will die

class `fixtures.artifactor_plugin.`**DummyClient**

Bases: `object`

fire_hook(**args*, ***kwargs*)

task_status()

terminate()

`fixtures.artifactor_plugin.`**get_test_idents**(*item*)

`fixtures.artifactor_plugin.`**pytest_addoption**(*parser*)

`fixtures.artifactor_plugin.`**pytest_configure**(*config*)

`fixtures.artifactor_plugin.`**pytest_runtest_logreport**(*report*)

`fixtures.artifactor_plugin.`**pytest_runtest_protocol**(*item*)

`fixtures.artifactor_plugin.`**pytest_runtest_teardown**(*item*, *nextitem*)

`fixtures.artifactor_plugin.`**pytest_unconfigure**()

fixtures.blockers module

Collection of fixtures for simplified work with blockers.

You can use the `blocker()` fixture to retrieve any blocker using blocker syntax (as described in `metaplugins.blockers`). The `bug()` fixture is specific for bugzilla, it accepts number argument and spits out the BUGZILLA BUG! (`utils.bz.BugWrapper`, not the `utils.blockers.BZ` instance!). The `blockers()` retrieves list of all blockers as specified in the meta marker. All of them are converted to the `utils.blockers.Blocker` instances

fixtures.blockers.**blocker**(*uses_blockers*)
Return any blocker that matches the expression.

Returns Instance of utils.blockers.Blocker

fixtures.blockers.**blockers**(*uses_blockers*, *meta*)
Returns list of all assigned blockers.

Returns List of utils.blockers.Blocker instances.

fixtures.blockers.**bug**(*blocker*)
Return bugzilla bug by its id.

Returns Instance of utils.bz.BugWrapper or NoneType if the bug is closed.

fixtures.blockers.**pytest_addoption**(*parser*)

fixtures.blockers.**pytest_collection_modifyitems**(*session*, *config*, *items*)

fixtures.browser module

fixtures.browser.**browser**()

fixtures.browser.**pytest_exception_interact**(*node*, *call*, *report*)

fixtures.browser.**pytest_namespace**()

fixtures.browser.**pytest_runtest_setup**(*item*)

fixtures.browser.**pytest_sessionfinish**(*session*, *exitstatus*)

fixtures.cfme_data module

fixtures.cfme_data.**cfme_data**(*request*)

fixtures.datafile module

fixtures.datafile.**datafile**(*filename*, *replacements*)
datafile fixture, with templating support

Parameters

- **filename** – filename to load from the data dir
- **replacements** – template replacements

Returns: Path to the loaded datafile

Usage:

```
Given a filename, it will attempt to open the given file from the
test's corresponding data dir. For example, this:

    datafile('testfile') # in tests/subdir/test_module_name.py

Would return a file object representing this file:

    /path/to/cfme_tests/data/subdir/test_module_name/testfile
```

```
Given a filename with a leading slash, it will attempt to load the file
relative to the root of the data dir. For example, this:

    datafile('/common/testfile') # in tests/subdir/test_module_name.py

Would return a file object representing this file:

    /path/to/cfme_tests/data/common/testfile

Note that the test module name is not used with the leading slash.
```

Templates:

This fixture can also handle template replacements. If the datafile being loaded is a python template, the dictionary of replacements can be passed as the 'replacements' keyword argument. In this case, the returned data file will be a NamedTemporaryFile prepopulated with the interpolated result from combining the template with the replacements mapping.

- http://docs.python.org/2/library/string.html#template-strings
- http://docs.python.org/2/library/tempfile.html#tempfile.NamedTemporaryFile

fixtures.datafile.**pytest_addoption**(*parser*)

fixtures.datafile.**pytest_sessionfinish**(*session*, *exitstatus*)

fixtures.db module

fixtures.db.**db**(*uses_db*)

Fixture providing utils.db.cfmedb

This is an SQLalchemy-based helper class which provides access to common database functions.

See also:

http://www.sqlalchemy.org/ session.

Usage:

```
# This example gets vm names and hostnames from the ext_management_systems table.
def test_that_tries_for_db(db):
    ems_table = db['ext_management_systems']
    for instance in db.session.query(ems_table.order_by(ems_table.id):)
        assert instance.name, instance.hostname
```

This fixture is module scoped to ensure predictable database access at the module level within tests.

fixtures.db.**db_yamls**(*db*)

Returns a mapping of database yaml names to the yaml contents serialized in python

fixtures.events module

fixtures.fixtureconf module

fixtures.fixtureconf.**fixtureconf**(*request*)
Provides easy access to the fixtureconf dict in fixtures

fixtures.log module

fixtures.log.**logger**()

fixtures.log.**pytest_collection_modifyitems**(*session*, *config*, *items*)

fixtures.log.**pytest_exception_interact**(*node*, *call*, *report*)

fixtures.log.**pytest_runtest_logreport**(*report*)

fixtures.log.**pytest_runtest_setup**(*item*)

fixtures.log.**pytest_sessionfinish**(*session*, *exitstatus*)

fixtures.log.**test_tracking = defaultdict(<type 'dict'>, {})**
A dict of tests, and their state at various test phases

fixtures.maximized module

Created on Mar 4, 2013

@author: bcrochet

fixtures.maximized.**maximized**()

fixtures.merkyl module

class fixtures.merkyl.**MerkylInspector**(*request*)
Bases: object

add_log(*log_name*)
Adds a log file to the merkyl process.

This function adds a log file path to the merkyl process on the appliance. This is relevant only for the duration of the test. At the end of the test, the file is removed from the merkyl tracker.

Note that this is a blocking call, ie, we ensure that the file is being logged by merkyl, before we continue. This is important and prevents the file_add operation being queued and processes which generate log information activating before the log is being monitored. This is achieved using the grab_result switch, but in fact, nothing will be received.

It is worth noting that the file path must be "discoverable" by merkyl. This may mean editing the allowed_files prior to deploying merkyl.

Parameters **log_name** – Full path to the log file wishing to be monitored.

get_log(*log_name*)
A simple getter for log files.

Returns the cached content of a particular log

Parameters **log_name** – Full path to the log file wishing to be received.

search_log(*needle*, *log_name*)
A simple search, test if needle is in cached log_contents.

Does a simple search of needle in contents. Note that this does not trawl the previous contents of the file, but only looks at the log information which has been gathered since merkyl was tracking the file.

fixtures.merkyl.**merkyl_inspector**(*request*)
Provides a MerkylInspector instance.

This fixture is used to gain access to a relevant MerkylInspector instance.

Example usage is below:

```
def test_test(merkyl_inspector):
    merkyl_inspector.add_log('/path/to/log/file')
    # Do something
    if merkyl_inspector.search_log('needle', '/path/to/log/file'):
        print merkyl_inspector.get_log('/path/to/log/file')
```

fixtures.mgmt_system module

fixtures.nelson module

class fixtures.nelson.**GoogleDocstring**(**args*, ***kwargs*)
Bases: sphinxcontrib.napoleon.docstring.GoogleDocstring

Custom version of napoleon's GoogleDocstring that adds some special cases

fixtures.nelson.**get_meta**(*obj*)

fixtures.nelson.**pytest_collection_modifyitems**(*items*)

fixtures.nelson.**pytest_pycollect_makeitem**(*collector*, *name*, *obj*)
pytest hook that adds docstring metadata (if found) to a test's meta mark

fixtures.nelson.**setup**(*app*)
Sphinx extension setup function.

See also:

http://sphinx-doc.org/extensions.html

fixtures.nelson.**stripper**(*docstring*)
Slightly smarter dedent

It strips a docstring's first line indentation and dedents the rest

fixtures.page_screenshots module

fixtures.page_screenshots.**pytest_addoption**(*parser*)

fixtures.perf module

Fixtures specifically for performance tests.

fixtures.perf.**cfme_log_level_rails_debug**()

fixtures.perf.**ui_worker_pid**()

fixtures.portset module

fixtures.portset.**pytest_addoption**(*parser*)

fixtures.portset.**pytest_configure**(*config*)

fixtures.prov_filter module

class fixtures.prov_filter.**ProviderFilter**(*defaults*)
Bases: object

providers

fixtures.prov_filter.**parse_filter**(*cmd_filter*)
Parse a list of command line filters and return a filtered set of providers.

Parameters **cmd_filter** – A list of `--use-provider` options.

fixtures.prov_filter.**provider_keys**()

fixtures.prov_filter.**pytest_addoption**(*parser*)

fixtures.prov_filter.**pytest_configure**(*config*)
Filters the list of providers as part of pytest configuration.

fixtures.provider module

fixtures.pxe_provision module

fixtures.pxe_provision.**host_provisioning_setup_data**(*cfme_data*)

fixtures.pxe_provision.**provisioning_setup_data**(*request*, *cfme_data*)

fixtures.pxe_provision.**setup_customization_template**(*db*, *provisioning_setup_data*, *row_val*, *ks_file_handle=None*)

fixtures.pxe_provision.**setup_host_provisioning_pxe**(*uses_pxe*, *db*, *host_provisioning_setup_data*, *datafile*)

fixtures.pxe_provision.**setup_pxe_image**(*db*, *provisioning_setup_data*, *server_last_id*, *menu_last_id*, *row_val*)
Add PXE Image

fixtures.pxe_provision.**setup_pxe_menu**(*db*, *provisioning_setup_data*, *server_last_id*)
Add PXE Menu

fixtures.pxe_provision.**setup_pxe_provision**(*uses_pxe*, *db*, *provisioning_setup_data*)
Sets up Infrastructure PXE for provisioning

fixtures.pxe_provision.**setup_pxe_server**(*db*, *provisioning_setup_data*)

fixtures.pxe_provision.**setup_vm_provisioning_pxe**(*uses_pxe*, *db*, *vm_provisioning_setup_data*, *datafile*)

fixtures.pxe_provision.**vm_provisioning_setup_data**(*cfme_data*)

fixtures.pytest_store module

Storage for pytest objects during test runs

The objects in the module will change during the course of a test run, so they have been stashed into the 'store' namespace

Usage:

```
# as pytest.store
import pytest
pytest.store.config, pytest.store.pluginmanager, pytest.store.session

# imported directly (store is pytest.store)
from fixtures.pytest_store import store
store.config, store.pluginmanager, store.session
```

The availability of these objects varies during a test run, but all should be available in the collection and testing phases of a test run.

class fixtures.pytest_store.**FlexibleTerminalReporter**(*config=None*, *file=None*)
Bases: _pytest.terminal.TerminalReporter

A TerminalReporter stand-in that pretends to work even without a py.test config.

class fixtures.pytest_store.**Store**
Bases: object

pytest object store

If a property isn't available for any reason (including being accessed outside of a pytest run), it will be None.

base_url
If there is a current appliance the base url of that appliance is returned else, the base_url from the config is returned.

capturemanager

config = None
The py.test config instance, None if not in py.test

current_appliance

fixturemanager

has_config

in_pytest_session

my_ip_address

parallel_session

parallelizer_role = None
Parallelizer role, None if not running a parallelized session

pluginmanager

session = None
The current py.test session, None if not in a py.test session

slave_manager

terminaldistreporter

terminalreporter

user

write_line(*line*, ***kwargs*)

fixtures.pytest_store.**pytest_configure**(*config*)

fixtures.pytest_store.**pytest_namespace**()

fixtures.pytest_store.**pytest_sessionstart**(*session*)

fixtures.pytest_store.**write_line**(*line*, ***kwargs*)
A write-line helper that should *always* write a line to the terminal

It knows all of py.tests dirty tricks, including ones that we made, and works around them.

Parameters ****kwargs** – Normal kwargs for pytest line formatting, stripped from slave messages

fixtures.qa_contact module

fixtures.qa_contact.**dig_code**(*node*)

fixtures.qa_contact.**pytest_runtest_teardown**(*item*, *nextitem*)

fixtures.randomness module

fixtures.randomness.**random_string**()
Generate a random string for use in tests

fixtures.randomness.**random_uuid_as_string**()
Creates a random uuid and returns is as a string

fixtures.rbac module

RBAC Role based parametrization and checking

The purpose of this fixture is to allow tests to be run within the context of multiple different users, without the hastle or modifying the test. To this end, the RBAC module and fixture do not require any modifications to the test body.

The RBAC fixture starts by receiving a list of roles and associated errors from the test metadata. This data is in YAML format and an example can be seen below.

```
Metadata:
    test_flag: provision
    suite: infra_provisioning
    rbac:
        roles:
            default:
            evmgroup-super_administrator:
            evmgroup-administrator:
            evmgroup-operator: NoSuchElementException
            evmgroup-auditor: NoSuchElementException
```

Let's assume also we have a test that looks like the following:

```
def test_rbac(rbac_role):
    if rbac_role != 'evmgroup-superadministrator' or rbac_role != 'evmgroup-operator':
        1 / 0
```

This metadata defines the roles to be tested, and associates with them the exceptions that are expected for that particular test, or blank if no Exception is expected. In this way we can have 5 states of test result.

- **Test Passed** - This was expected - We do nothing to this and exit early. In the example above evmgroup-super_administrator fulfills this, as it expects no Exception.
- **Test Failed** - This was expected - We consume the Exception and change the result of the test to be a pass. In the example, this is fulfilled by evmgroup-auditor as it was expected to fail with the ZeroDivisionError.
- **Test Failed** - This was unexpected - We consume the Exception and raise another informing that the test should have passed. In the example above, evmgroup-administrator satisfies this condition as it didn't expect a failure, but got one.
- **Test Failed** - This was expected, but the wrong Exception appeared - We consume the Exception throw another stating that the Exception wasn't of the expected type. In the example above, the default user satifies this as it receives the ZeroDivisionError, but expects MonkeyError.
- **Test Passed** - This was unexpected - We have Exception to consume, but we raise an Exception of our own as the test should have failed. In the example above, evmgroup-operator satisfies this as it should have received the ZeroDivisionError, but actually passes with no error.

When a test is configured to run against the RBAC suite, it will first parametrize the test with the associated roles from the metadata. The test will then be wrapped and before it begins we login as the *new* user. This process is also two fold. The `pytest_store` holds the current user, and logging in is performed with whatever this user value is set to. So we first replace this value with our new user. This ensures that if the browser fails during a force_naviagate, we get the opportunity to log in again with the *right* user. Once the user is set, we attempt to login.

When the test finishes, we set the user back to `default` before moving on to handling the outcome of the test with the wrapped hook handler. This ensures that the next test will have the correct user at login, even if the test fails horribly, and even if the inspection of the outcome should fail.

To configure a test to use RBAC is simple. We simply need to add `rbac_role` to the list of fixtures and the addition and the ldap configuration fixture also. Below is a complete example of adding RBAC to a test.

```
import pytest

def test_rbac(rbac_role):
""" Tests provisioning from a template

Metadata:
    rbac:
        roles:
            default:
            evmgroup-super_administrator:
            evmgroup-administrator:
            evmgroup-operator: NoSuchElementException
            evmgroup-auditor: NoSuchElementException
"""
    if rbac_role != 'evmgroup-superadministrator' or rbac_role != 'evmgroup-operator':
        1 / 0
```

Exception matching is done with a simple string startswith match.

Currently there is no provision for skipping a role for a certain test, though this is easy to implement. There is also no provision, for tests that have multiple parameters, to change the expectation of the test, with relation to a parameter. For example, if there was a parameter called *rhos* and one called *ec2* we could not change the expected exception to be different depending on if the test was run against *rhos* or *ec2*.

`fixtures.rbac.`**`pytest_addoption`**`(`*parser*`)`

fixtures.rbac.**pytest_configure**(*config*)
Filters the list of providers as part of pytest configuration.

fixtures.rbac.**pytest_generate_tests**(*metafunc*)

fixtures.rbac.**pytest_pyfunc_call**(*pyfuncitem*)
Inspects and consumes certain exceptions

The guts of this function are explained above in the module documentation.

Parameters **pyfuncitem** – A pytest test item.

fixtures.rbac.**really_logout**()
A convenience function logging out

This function simply ensures that we are logged out and that a new browser is loaded ready for use.

fixtures.rbac.**save_screenshot**(*node*, *ss*, *sse*)

fixtures.rbac.**save_traceback_file**(*node*, *contents*)
A convenience function for artifactor file sending

This function simply takes the nodes id and the contents of the file and processes them and sends them to artifactor

Parameters

- **node** – A pytest node
- **contents** – The contents of the traceback file

fixtures.single_appliance_sprout module

fixtures.snmp module

Fixture providing SNMP client for tests that want it.

fixtures.snmp.**snmp_client**(*ssh_client*)

fixtures.soap_client module

fixtures.soap_client.**soap_client**(*uses_soap*)

fixtures.soft_assert module

Soft assert context manager and assert function

A "soft assert" is an assertion that, if it fails, does not fail the entire test. Soft assertions can be mixed with normal assertions as needed, and will be automatically collected/reported after a test runs.

Functionality Overview

1. If `soft_assert()` is used by a test, that test's call phase is wrapped in a context manager. Entering that context sets up a thread-local store for failed assertions.
2. Inside the test, `soft_assert()` is a function with access to the thread-local store of failed assertions, allowing it to store failed assertions during a test run.

3. After a test runs, the context manager wrapping the test's call phase exits, which inspects the thread-local store of failed assertions, raising a custom AssertionError if any are found.

No effort is made to clear the thread-local store; rather it's explicitly overwritten with an empty list by the context manager. Because the store is a list, failed assertions will be reported in the order that they failed.

exception fixtures.soft_assert.**SoftAssertionError**(*failed_assertions*)
Bases: exceptions.AssertionError

exception class containing failed assertions

Functions like AssertionError, but also stores the failed soft exceptions that it represents in order to properly display them when cast as str

Parameters

- **failed_assertions** – List of collected assertion failure messages
- **where** – Where the SoftAssert context was entered, can be omitted

failed_assertions
failed_assertions handed to the initializer, useful in cases where inspecting the failed soft assertions is desired.

fixtures.soft_assert.**handle_assert_artifacts**(*request*, *fail_message=None*)

fixtures.soft_assert.**pytest_runtest_call**(*item*)
pytest hook to handle soft_assert() fixture usage

fixtures.soft_assert.**soft_assert**(*request*)
soft assert fixture, used to defer AssertionError to the end of a test run

Usage:

```
# contents of test_soft_assert.py, for example
def test_uses_soft_assert(soft_assert):
    soft_assert(True)
    soft_assert(False, 'failure message')

    # soft_assert.catch_assert will intercept AssertionError
    # and turn it into a soft assert
    with soft_assert.catch_assert():
        assert None

    # Soft asserts can be cleared at any point within a test:
    soft_assert.clear_asserts()

    # If more in-depth interaction is desired with the caught_asserts, the list of failure
    # messages can be retrieved. This will return the directly mutable caught_asserts list:
    caught_asserts = soft_assert.caught_asserts()
```

The test above will report two soft assertion failures, with the following message:

```
SoftAssertionError:
failure message (test_soft_assert.py:3)
soft_assert(None) (test_soft_assert.py:8)
```

fixtures.ssh_client module

fixtures.ssh_client.**pytest_sessionfinish**(*session*, *exitstatus*)
Loop through the appliance stack and close ssh connections

fixtures.ssh_client.**ssh_client**(*uses_ssh*)
SSH Client Fixture

Usage:

```
def test_ssh(ssh_client):
    # Run a basic command
    result = ssh_client.run_command('ls -al')
    # rc is the numeric return code from the called command,
    # so 0 means everything is OK.
    assert result.rc == 0
    # and the output is available, too
    print result.output

    # Run a task using the CFME rails runner CLI
    ssh_client.run_rails_command('do stuff')

    # More useful: Run a rake task using the correct invokation
    ssh_client.run_rake_command('evm:stop')
```

Additionally, the ssh_client fixture can be used to create other ssh clients, if you need to connect to multiple hosts in a test run:

```
def test_multiple_ssh(ssh_client):
    # Normal behavior still works
    ssh_client.run_command('some_command')

    # Instantiate a client aimed at a different hostname
    ssh_client_2 = ssh_client(hostname='different.host')
    ssh_client_2.run_command('some_other_command')

    # Username and password can be changed, too
    ssh_client_3 = ssh_client(username='foo', password='bar')

    # Hint: **credentials['credentials_key'], e.g.
    ssh_client_4 = ssh_client(hostname='different.host', **credentials['ssh'])
```

fixtures.ssh_client.**ssh_client_modscope**(*uses_ssh*)
See `ssh_client()`.

fixtures.templateloader module

fixtures.terminalreporter module

fixtures.terminalreporter.**disable**()

fixtures.terminalreporter.**enable**()

fixtures.terminalreporter.**reporter**(*config=None*)
Return a py.test terminal reporter that will write to the console no matter what

Only useful when trying to write to the console before or during a `pytest_configure` hook.

fixtures.ui_coverage module

UI Coverage for a CFME/MIQ Appliance

Usage

```
py.test --ui-coverage
```

General Notes

simplecov can merge test results, but doesn't appear to like working in a multi-process environment. Specifically, it clobbers its own results when running simultaneously in multiple processes. To solve this, each process records its output to its own directory (configured in coverage_hook). All of the individual process' results are then manually merged (coverage_merger) into one big json result, and handed back to simplecov which generates the compiled html (for humans) and rcov (for jenkins) reports.

thing_toucher makes a best-effort pass at requiring all of the ruby files in the rails root, as well as any external MIQ libs/utils outside of the rails root (../lib and ../lib/util). This makes sure files that are never required still show up in the coverage report.

Workflow Overview

Pre-testing (`pytest_configure` hook):

1. Add `Gemfile.dev.rb` to the rails root, then run bundler to install simplecov and its dependencies.
2. Install and require the coverage hook (copy `coverage_hook` to config/, add require line to the end of `config/boot.rb`)
3. Restart EVM (Rudely) to start running coverage on the appliance processes: `killall -9 ruby; service evmserverd start`
4. TOUCH ALL THE THINGS (run `thing_toucher.rb` with the rails runner). Fork this process off and come back to it later

Post-testing (`pytest_unconfigure` hook):

1. Poll `thing_toucher` to make sure it completed; block if needed.
2. Stop EVM, but nicely this time so the coverage atexit hooks run: `service evmserverd stop`
3. Run `coverage_merger.rb` with the rails runner, which compiles all the individual process reports and runs coverage again, additionally creating an rcov report
4. Pull the coverage dir back for parsing and archiving
5. For fun: Read the results from `coverage/.last_run.json` and print it to the test terminal/log

Post-testing (e.g. ci environment): 1. Use the generated rcov report with the ruby stats plugin to get a coverage graph 2. Zip up and archive the entire coverage dir for review

class `fixtures.ui_coverage.`**CoverageManager**(*ipappliance*)
Bases: `object`

collect()

collection_appliance

install()

merge()

print_message(*message*)

class fixtures.ui_coverage.**UiCoveragePlugin**
Bases: object

pytest_collection_finish()

pytest_configure(*config*)

pytest_sessionfinish(*exitstatus*)

pytest_sessionstart(*session*)

fixtures.ui_coverage.**appliance_coverage_root = local('/var/www/miq/vmdb/coverage')**
coverage root, should match what's in the coverage hook and merger scripts

fixtures.ui_coverage.**clean_coverage_dir**()

fixtures.ui_coverage.**manager**()

fixtures.ui_coverage.**pytest_addoption**(*parser*)

fixtures.ui_coverage.**pytest_cmdline_main**(*config*)

fixtures.ui_coverage.**rails_root = local('/var/www/miq/vmdb')**
Corresponds to Rails.root in the rails env

fixtures.update_appliance module

Appliance update plugin

If update_urls is set in the env, re-trigger the update_rhel configuration step to update the appliance with the new URLs

fixtures.update_appliance.**pytest_parallel_configured**()

fixtures.version_file module

fixtures.version_file.**pytest_sessionstart**()

fixtures.video module

Provides video options

Yaml example:

```
logging:
   video:
       enabled: True
       dir: video
       display: ":99"
       quality: 10
```

fixtures.video.**get_path_and_file_name**(*node*)
Extract filename and location from the node.

Parameters **node** – py.test collection node to examine.

Returns: 2-tuple *(path, filename)*

fixtures.video.**pytest_runtest_setup**(*item*)

fixtures.video.**pytest_runtest_teardown**(*item*, *nextitem*)

fixtures.video.**pytest_unconfigure**(*config*)

fixtures.video.**stop_recording**()

fixtures.virtual_machine module

Fixtures ensuring that a VM/instance is in the specified state for the test

fixtures.virtual_machine.**verify_vm_paused**(*provider*, *vm_name*)
Ensures that the VM/instance is paused for the test

Uses calls to the actual provider api; it will pause the vm if necessary.

Parameters

- **provider.mgmt** – Provider class object
- **vm_name** – Name of the VM/instance

fixtures.virtual_machine.**verify_vm_running**(*provider*, *vm_name*)
Ensures that the VM/instance is in running state for the test

Uses calls to the actual provider api; it will start the vm if necessary.

Parameters

- **provider** – Provider class object
- **vm_name** – Name of the VM/instance

fixtures.virtual_machine.**verify_vm_stopped**(*provider*, *vm_name*)
Ensures that the VM/instance is stopped for the test

Uses calls to the actual provider api; it will stop the vm if necessary.

Parameters

- **provider** – Provider class object
- **vm_name** – Name of the VM/instance

fixtures.virtual_machine.**verify_vm_suspended**(*provider*, *vm_name*)
Ensures that the VM/instance is suspended for the test

Uses calls to the actual provider api; it will suspend the vm if necessary.

Parameters

- **provider.mgmt** – Provider class object
- **vm_name** – Name of the VM/instance

fixtures.widgets module

fixtures.widgets.**widgets_generated**(*any_provider_session*)

3.2.3 Module contents

3.3 markers package

3.3.1 Submodules

markers.crud module

crud: Marker for marking the test as a CRUD test (crud)

Useful for eg. running only crud tests. Tests will be marked automatically if:

- their name starts with crud_
- their name ends with _crud
- their name contains _crud_

markers.crud.**pytest_configure**(*config*)

markers.crud.**pytest_itemcollected**(*item*)

markers.fixtureconf module

fixtureconf: Marker for passing args and kwargs to test fixtures

Positional and keyword arguments to this marker will be stored on test items in the _fixtureconf attribute (dict). kwargs will be stored as-is, the args tuple will be packed into the dict under the 'args' key.

Use the "fixtureconf" fixture in tests to easily access the fixtureconf dict

markers.fixtureconf.**pytest_configure**(*config*)

markers.fixtureconf.**pytest_runtest_setup**(*item*)

markers.meta module

meta(**metadata): Marker for metadata addition.

To add metadata to a test simply pass the kwargs as plugins wish.

You can write your own plugins. They generally live in `metaplugins/` directory but you can define them pretty much everywhere py.test loads modules. Plugin has a name and a set of callbacks that are called when certain combination of keys is present in the metadata.

To define plugin, do like this:

```
@plugin("plugin_name")
def someaction(plugin_name):
    print plugin_name  # Will contain value of `plugin_name` key of metadict
```

This is the simplest usage, where it is supposed that the plugin checks only one key with the same name s the plugin's name. I won't use this one in the latter examples, I will use the more verbose one.

```
@plugin("plugin_name", keys=["plugin_name", "another_key"])
def someaction(plugin_name, another_key):
    print plugin_name  # Will contain value of `plugin_name` key of metadict
    print another_key  # Similarly this one
```

This one reacts when the two keys are present. You can make even more complex setups:

```
@plugin("plugin_name", keys=["plugin_name"])
@plugin("plugin_name", ["plugin_name", "another_key"])  # You don't have to write keys=
def someaction(plugin_name, another_key=None):
    print plugin_name  # Will contain value of `plugin_name` key of metadict
    print another_key  # Similarly this one if specified, otherwise None
```

This created a nonrequired parameter for the action.

You can specify as many actions as you wish per plugin. The only thing that limits you is the correct action choice. First, all the actions are filtered by present keys in metadata. Then after this selection, only the action with the most matched keywords is called. Bear this in your mind. If this is not enough in the future, it can be extended if you wish.

It has a command-line option that allows you to disable certain plugins. Just specify `--disablemetaplugins a,b,c` where a, b and c are the plugins that should be disabled

class markers.meta.**Plugin**

Bases: tuple

Plugin(name, metas, function, kwargs)

function

Alias for field number 2

kwargs

Alias for field number 3

metas

Alias for field number 1

name

Alias for field number 0

class markers.meta.**PluginContainer**

Bases: object

AFTER_RUN = 'after_run'

BEFORE_RUN = 'before_run'

DEFAULT = 'setup'

SETUP = 'setup'

TEARDOWN = 'teardown'

markers.meta.**meta**(*request*)

markers.meta.**pytest_addoption**(*parser*)

markers.meta.**pytest_collection_modifyitems**(*session*, *config*, *items*)

markers.meta.**pytest_configure**(*config*)

markers.meta.**pytest_pycollect_makeitem**(*collector*, *name*, *obj*)

markers.meta.**pytest_runtest_call**(*item*)

markers.meta.**pytest_runtest_setup**(*item*)

markers.meta.**pytest_runtest_teardown**(*item*)

markers.meta.**run_plugins**(*item*, *when*)

markers.requires module

requires_test(test_name_or_nodeid): Mark a test as requiring another test

If another test is required to have run and passed before a suite of tests has any hope of succeeding, such as a smoke test, apply this mark to those tests.

It takes a test name as the only positional argument. In the event that the test name is ambiguous, a full py.test nodeid can be used. A test's nodeid can be found by inspecting the request.node.nodeid attribute inside the required test item.

markers.requires.**pytest_configure**(*config*)

markers.requires.**pytest_runtest_setup**(*item*)

markers.sauce module

sauce: Mark a test to run on sauce

Mark a single test to run on sauce.

markers.sauce.**pytest_addoption**(*parser*)

markers.sauce.**pytest_configure**(*config*)

markers.skipper module

skipper: Autmatically skip tests with certain marks as defined in this module

This doesn't provide any special markers, but it does add behavor to marks defined in `skip_marks`.

markers.skipper.**pytest_addoption**(*parser*)

markers.skipper.**pytest_collection_modifyitems**(*items*)

markers.skipper.**pytest_configure**(*config*)

markers.skipper.**skip_marks = [('long_running', '–long-running'), ('perf', '–perf')]**
List of (mark, commandline flag) tuples. When the given mark is used on a test, it will be skipped unless the commandline flag is used. If the mark is already found in py.test's parsed mark expression, no changes will be made for that mark.

markers.smoke module

smoke: Mark a test as a smoke test to be run as early as possible

Mark a single test as a smoke test, moving it to the beginning of a test run.

The –halt-on-smoke-test-failure command-line argument will halt after running the smoke tests if any smoke tests fail.

This mark must be used with caution, as marked tests must be able to run out of order, and in isolation.

Furthermore, smoke tests are an excellent target for the requires_test mark since they're run first.

class markers.smoke.**SmokeTests**(*reporter*)
Bases: `object`

complete = False

failed_tests = 0

halt_on_fail = False

pytest_runtest_logreport(*report*)

pytest_runtest_teardown(*item*, *nextitem*)

reported = False

run_tests = 0

start_time = 0.0

markers.smoke.**pytest_addoption**(*parser*)

markers.smoke.**pytest_collection_modifyitems**(*session*, *config*, *items*)

markers.smoke.**pytest_configure**(*config*)

markers.stream_excluder module

ignore_stream(*streams): Marker for uncollecting the tests based on appliance stream.

Streams are the first two fields from version of the appliance (5.0, 5.1, ...), the nightly upstream is represented as upstream. If you want to ensure, that the test shall not be collected because it is not supposed to run on 5.0 and 5.1 streams, just put those streams in the parameters and that is enough.

It also provides a facility to check the appliance's version/stream for smoke testing.

markers.stream_excluder.**get_streams_id**()

markers.stream_excluder.**pytest_addoption**(*parser*)

markers.stream_excluder.**pytest_collection_modifyitems**(*session*, *config*, *items*)

markers.stream_excluder.**pytest_configure**(*config*)

markers.stream_excluder.**pytest_itemcollected**(*item*)

markers.uncollect module

uncollect

Used internally to mark a test to be "uncollected"

This mark should be used at any point before or during test collection to dynamically flag a test to be removed from the list of collected tests.

py.test adds marks to test items a few different ways. When marking in a py.test hook that takes an `Item` or `Node` (`Item` is a subclass of `Node`), use `item.add_marker('uncollect')` or `item.add_marker(pytest.mark.uncollect)`

When dealing with the test function directly, using the mark decorator is preferred. In this case, either decorate a test function directly (and have a good argument ready for adding a test that won't run...), e.g. `@pytest.mark.uncollect` before the test `def`, or instantiate the mark decorator and use it to wrap a test function, e.g. `pytest.mark.uncollect()(test_function)`

uncollectif

The `uncollectif` marker is very special and can cause harm to innocent kittens if used incorrectly. The `uncollectif` marker enables the ability to uncollect a specific test if a certain condition is evaluated to `True`. The following is an example:

```
@pytest.mark.uncollectif(lambda: version.current_version() < '5.3')
```

In this case, when pytest runs the modify items hook, it will evaluate the lambda function and if it results in `True`, then the test will be uncollected. Fixtures that are generated by testgen, such as provider_key, provider_data etc, are also usable inside the `collectif` marker, assuming the fixture name is also a prerequisite for the test itself. For example:: python

```
@pytest.mark.uncollectif(lambda provider_type: provider_type != 'virtualcenter')
def test_delete_all_snapshots(test_vm, provider_key, provider_type):
    pass
```

Here, the fixture provider_type is special as it comes from testgen and is passed to the lambda for comparison.

Note: Be aware, that this cannot be used for any other fixture types. Doing so will break pytest and may invalidate your puppies.

---

markers.uncollect.**pytest_collection_modifyitems**(*session*, *config*, *items*)

markers.uncollect.**uncollectif**(*item*)
Evaluates if an item should be uncollected

Tests markers against a supplied lambda from the marker object to determine if the item should be uncollected or not.

markers.uses module

uses_*: Provides a set of fixtures used to mark tests for filtering on the command-line.

Tests using these fixtures directly or indirectly can be filtered using py.test's `-k` filter argument. For example, run tests that use the ssh client:

```
py.test -k uses_ssh
```

Additionally, tests using one of the fixtures listed in `appliance_marks` will be marked with *is_appliance*, for easily filtering out appliance tests, e.g:

```
py.test -k 'not is_appliance'
```

All fixtures created by this module will have the `uses_` prefix.

Note: `is_appliance` is a mark that will be dynamically set based on fixtures used, but is not a fixture itself.

---

markers.uses.**appliance_marks = set(['uses_ssh', 'uses_db'])**
List of fixtures that, when used, indicate an appliance is being tested by applying the `is_appliance` mark.

markers.uses.**pytest_itemcollected**(*item*)
pytest hook that actually does the marking

See: http://pytest.org/latest/plugins.html#_pytest.hookspec.pytest_collection_modifyitems

markers.uses.**uses_blockers**()
Fixture which marks a test with the `uses_blockers` mark

markers.uses.**uses_cloud_providers**(*uses_providers*)
Fixture which marks a test with the `uses_cloud_providers` and `uses_providers` marks

markers.uses.**uses_db**()
Fixture which marks a test with the `uses_db` mark

markers.uses.**uses_event_listener**()
Fixture which marks a test with the uses_event_listener mark

markers.uses.**uses_infra_providers**(*uses_providers*)
Fixture which marks a test with the uses_infra_providers and uses_providers marks

markers.uses.**uses_providers**()
Fixture which marks a test with the uses_providers mark

markers.uses.**uses_pxe**()
Fixture which marks a test with the uses_pxe mark

markers.uses.**uses_soap**()
Fixture which marks a test with the uses_soap mark

markers.uses.**uses_ssh**()
Fixture which marks a test with the uses_ssh mark

3.3.2 Module contents

3.4 metaplugins package

3.4.1 Submodules

metaplugins.blockers module

A generalized framowork for handling test blockers.

Currently handling Bugzilla nad GitHub issues. For extensions, see this file and utils.blockers.

If you want to mark test with blockers, use meta mark blockers and specify a list of the blockers. The blockers can be directly the objects of utils.blockers.Blocker subclasses, but you can use just plain strings that will get resolved into the objects when required.

Example comes:

```
@pytest.mark.meta(
    blockers=[
        BZ(123456),             # Will get resolved to BZ obviously
        GH(1234),               # Will get resolved to GH if you have default repo set
        GH("owner/repo:issue"), # Otherwise you need to use this syntax
        # Generic blocker writing - (<engine_name>#<blocker_spec>)
        # These work for any engine that is in :py:mod:`utils.blockers`
        "BZ#123456",            # Will resolve to BZ
        "GH#123",               # Will resolve to GH (needs default repo specified)
        "GH#owner/repo:123",    # Will resolve to GH
        # Shortcut writing
        123456,                 # Will resolve to BZ
    ]
)
```

Íf you want to unskip, then you have to use the full object (BZ()) and pass it a kwarg called unblock. When the function in unblock resolves to a truthy value, the test won't be skipped. If the blocker does not block, the unblock is not called. There is also a custom_action that will get called if the blocker blocks. if the action does nothing, then it continues with next actions etc., until it gets to the point that it skips the test because there are blockers.

metaplugins.blockers.**kwargify**(*f*)
Convert function having only positional args to a function taking dictionary.

If you pass False or None, a function which always returns False is returned. If you pass True, a function which always returns True is returned.

metaplugins.blockers.**resolve_blockers**(*item*, *blockers*)

metaplugins.server_roles module

Set server roles based on a list of roles attached to the test using metadata plugin.

If you want to specify certain roles that have to be set, you can use this type of decoration:

```
@pytest.mark.meta(server_roles="+automate")
def test_appliance_roles():
    assert foo
```

This takes the current list from cfme_data.yaml and modifies it by the server_roles keyword. If prefixed with + or nothing, it adds, if prefixed with -, it removes the role. It can be combined either in string and in list, so these lines are functionally equivalent:

```
"+automate -foo bar" # (add automate and bar, remove foo)
["+automate", "-foo", "bar"]
```

If you specify the server_roles as `None`, then all roles are flushed and the list contains only user_interface role.

Roles can be pulled from the cfme_data fixture using yaml selectors, which will do a 'set' with the list of roles found at the target path:

```
@pytest.mark.meta(server_roles=('level1', 'sublevel2'), server_roles_mode='cfmedata')
def test_appliance_roles():
    assert len(get_server_roles()) == 3
```

Which corresponds to this yaml layout:

```
level1:
    sublevel2:
        - database_operations
        - user_interface
        - web_services
```

To ensure the appliance has the default roles:

```
@pytest.mark.fixtureconf(server_roles="default")
def test_appliance_roles():
    do(test)
```

For a list of server role names currently exposed in the CFME interface, see keys of `cfme.configure.configuration.server_roles`.

metaplugins.server_roles.**add_server_roles**(*server_roles*, *server_roles_mode='add'*)

metaplugins.skip module

I missed callable based skipper so here it is.

metaplugins.skip.**skip_plugin**(*item*, *skip*, *reason='Skipped'*)

3.4.2 Module contents

3.5 utils package

3.5.1 Subpackages

utils.mgmt_system package

Submodules

utils.mgmt_system.openstack_infra module

Module contents

3.5.2 Submodules

utils.api module

class utils.api.**API**(*entry_point*, *auth*)
Bases: object

api_version(*version*)

delete(*url*, ***payload*)

get(*url*, ***get_params*)

get_entity(*collection_or_name*, *entity_id*)

latest_version

new_id_behaviour
2.0.0 introduced a new id/href difference.

on_latest_version

post(*url*, ***payload*)

version

versions

exception utils.api.**APIException**
Bases: exceptions.Exception

class utils.api.**Action**(*container*, *name*, *method*, *href*)
Bases: object

api

collection

class utils.api.**ActionContainer**(*obj*)
Bases: object

all

collection

reload()

class utils.api.**Collection**(*api*, *href*, *name*, *description=None*)
Bases: object

all

api

count

find_by(***params*)
Search items in collection. Filters based on keywords passed.

get(***params*)

reload(*expand=False*)

reload_if_needed()

subcount

class utils.api.**CollectionsIndex**(*api*, *data*)
Bases: object

all

all_names

class utils.api.**Entity**(*collection*, *data*, *incomplete=False*)
Bases: object

COLLECTION_MAPPING = {'zone_id': 'zones', 'task_id': 'tasks', 'current_group_id': 'groups', 'evm_owner_id': 'users',

SUBCOLLECTIONS = {'service_catalogs': set(['service_templates'])}

TIME_FIELDS = set(['last_scan_attempt_on', 'created_at', 'updated_at', 'state_changed_on', 'lastlogon', 'created_on', '

exists

reload(*expand=None*, *get=True*)

reload_if_needed()

wait_exists(***kwargs*)

wait_for_existence(*existence*, ***kwargs*)

wait_not_exists(***kwargs*)

class utils.api.**SearchResult**(*collection*, *data*)
Bases: object

utils.apidoc module

Sphinx plugin for automatically generating (and optionally cleaning) project api documentation

To enable the optional cleaning, set `clean_autogenerated_docs` to `True` in docs/conf.py

utils.apidoc.**modules_to_document = ['cfme', 'fixtures', 'markers', 'metaplugins', 'utils']**
List of modules/packages to document, paths relative to the project root.

utils.apidoc.**purge_module_apidoc**(*sphinx*, *exception*)

utils.apidoc.**setup**(*sphinx*)
Main sphinx entry point, calls sphinx-apidoc

utils.appliance module

utils.artifactor_start module

utils.artifactor_start.**run**(*port*, *run_id=None*)

utils.async module

class utils.async.**ResultsPool**(**args*, ***kwargs*)
Bases: multiprocessing.pool.Pool

multiprocessing.Pool boilerplate wrapper

- Stores results on a results property
- Task result successes are aggregated by the sucessful property

apply_async(**args*, ***kwargs*)

map_async(**args*, ***kwargs*)

successful

utils.blockers module

class utils.blockers.**BZ**(*bug_id*, ***kwargs*)
Bases: utils.blockers.Blocker

blocks

bugzilla_bug

data

get_bug_url()

class utils.blockers.**Blocker**(***kwargs*)
Bases: object

Base class for all blockers

REQUIRED THING! Any subclass' constructors must accept kwargs and after POPping the values required for the blocker's operation, *call to ''super'* with `**kwargs` must be done! Failing to do this will render some of the functionality disabled ;).

classmethod all_blocker_engines()
Return mapping of name:class of all the blocker engines in this module.

Having this as a separate function will later enable to scatter the engines across modules in case of extraction into a separate library.

blocks = False

kwargs = {}

classmethod parse(*blocker*)
Create a blocker object from some representation

class utils.blockers.**GH**(*description*, ***kwargs*)
Bases: utils.blockers.Blocker

DEFAULT_REPOSITORY = 'foo/bar'

blocks

data

github = <github.MainClass.Github object at 0x7f52e86c5550>

repo

utils.browser module

Core functionality for starting, restarting, and stopping a selenium browser.

class utils.browser.**DuckwebQaClient**
Bases: object

selenium

class utils.browser.**DuckwebQaTestSetup**
Bases: object

A standin for mozwebqa's TestSetup class

Pretends to be a mozwebqa TestSetup so we can uninstall mozwebqa whithout breaking old tests that aren't yet converted.

Note: This should never be used, and places where it's currently used should stop it.

---

default_implicit_wait

selenium

timeout

class utils.browser.**Wharf**(*wharf_url*)
Bases: object

checkin()

checkout()

renew()

utils.browser.**browser**()
callable that will always return the current browser instance

If None, no browser is running.

Returns The current browser instance.

utils.browser.**browser_session**(**args*, ***kwds*)
A context manager that can be used to start and stop a browser.

Usage:

```
with browser_session as browser:
    # do stuff with browser here
# Browser will be closed here
```

utils.browser.**ensure_browser_open**()
Ensures that there is a browser instance currently open

Will reuse an existing browser or start a new one as-needed

Returns The current browser instance.

utils.browser.**firefox_profile_tmpdir** = **None**
After starting a firefox browser, this will be set to the temporary directory where files are downloaded.

utils.browser.**quit**()
Close the current browser

Will silently fail if the current browser can't be closed for any reason.

Note: If a browser can't be closed, it's usually because it has already been closed elsewhere.

utils.browser.**start**(*webdriver_name=None*, *base_url=None*, ***kwargs*)
Starts a new web browser

If a previous browser was open, it will be closed before starting the new browser

Parameters

- **webdriver_name** – The name of the selenium Webdriver to use. Default: 'Firefox'
- **base_url** – Optional, will use `utils.conf.env['base_url']` by default
- ****kwargs** – Any additional keyword arguments will be passed to the webdriver constructor

utils.browser.**wharf**()

utils.bz module

class utils.bz.**BugWrapper**(*bugzilla*, *bug*)
Bases: object

bugzilla

can_test_on_upstream

copies
Returns list of copies of this bug.

copy_of
Returns either id of the bug this is copy of, or None, if it is not a copy.

is_opened

loose

product

qa_whiteboard
Returns a set of QA Whiteboard markers.

It relies on the fact, that our QA Whiteboard uses format foo:bar:baz.

Should be able to handle cases like 'foo::bar', or 'abc:'.

release_flag

upstream_bug

zstream

class utils.bz.**Bugzilla**(***kwargs*)
Bases: object

bug_count

bugs

bugzilla

default_product

classmethod from_config()

get_bug(*id*)

get_bug_variants(*id*)

loose

open_states

product(*product*)

products(**names*)

resolve_blocker(*blocker*, *version=None*, *ignore_bugs=set([])*, *force_block_streams=[]*)

upstream_version

class utils.bz.**Product**(*data*)
Bases: object

default_release

latest_version

milestones

name

releases

versions

utils.bz.**check_fixed_in**(*fixed_in*, *version_series*)

utils.category module

Module used for handling categories of let's say form values and for categorizing them.

class utils.category.**CategoryBase**(*value*)
Bases: object

Base class for categories

Parameters **value** – Value to be categorized.

utils.category.**categorize**(*iterable*, *cat*)
Function taking iterable of values and a dictionary of rules to categorize the values.

Keys of the dictionary are callables, taking one parameter - the current iterable item. If the call on it returns positive, then the value part of dictionary is taken (assumed callable) and it is called with the current item.

Parameters

- **iterable** – Iterable to categorize.
- **cat** – Category specification dictionary

utils.conf module

Configuration YAML loader and cache

utils.conf.**copy**() → a shallow copy of D

utils.conf.**fromkeys**(*S*[, *v*]) → New dict with keys from S and values equal to v.
v defaults to None.

utils.conf.**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

utils.conf.**has_key**(*k*) → True if D has a key k, else False

utils.conf.**items**() → list of D's (key, value) pairs, as 2-tuples

utils.conf.**iteritems**() → an iterator over the (key, value) items of D

utils.conf.**iterkeys**() → an iterator over the keys of D

utils.conf.**itervalues**() → an iterator over the values of D

utils.conf.**keys**() → list of D's keys

utils.conf.**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

utils.conf.**popitem**() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

utils.conf.**setdefault**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

utils.conf.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

utils.conf.**values**() → list of D's values

utils.conf.**viewitems**() → a set-like object providing a view on D's items

utils.conf.**viewkeys**() → a set-like object providing a view on D's keys

utils.conf.**viewvalues**() → an object providing a view on D's values

utils.datafile module

datafile functions, to help reliably datafiles from the data directory.

utils.datafile.**data_path_for_filename**(*filename*, *base_path*, *testmod_path=None*)
Returns the data path for a given file name

utils.datafile.**load_data_file**(*filename*, *replacements=None*)
Opens the given filename, returning a file object

Parameters

- **filename** – If a base_path string is passed, filename will be loaded from there
- **replacements** – If a replacements mapping is passed, the loaded file is assumed to be a template. In this case the replacements mapping will be used in that template's subsitute method.

Returns: A file object.

utils.db module

class utils.db.**Db**(*hostname=None*, *credentials=None*)
Bases: _abcoll.Mapping

Helper class for interacting with a CFME database using SQLAlchemy

Parameters

- **hostname** – base url to be used (default is from current_appliance)
- **credentials** – name of credentials to use from utils.conf.credentials (default database)

Provides convient attributes to common sqlalchemy objects related to this DB, as well as a Mapping interface to access and reflect database tables. Where possible, attributes are cached.

Db objects support getting tables by name via the mapping interface:

```
table = db['table_name']
```

Usage:

```
# Usually used to query the DB for info, here's a common query
for vm in db.session.query(db['vms']).all():
    print vm.name, vm.guid

# List comprehension to get all templates
[(vm.name, vm.guid) for vm in session.query(db['vms']).all() if vm.template is True]

# Use the transaction manager for write operations:
with db.transaction:
    db.session.query(db['vms']).all().delete()
```

Note: Creating a table object requires a call to the database so that SQLAlchemy can do reflection to determine the table's structure (columns, keys, indices, etc). On a latent connection, this can be extremely slow, which will affect methods that return tables, like the mapping interface or values().

---

copy()
Copy this database instance, keeping the same credentials and hostname

db_url
The connection URL for this database, including credentials

This attribute is lazily evaluated and cached.

engine
The Engine for this database

It uses pessimistic disconnection handling, checking that the database is still connected before executing commands.

This attribute is lazily evaluated and cached.

get(*table_name*, *default=None*)
table getter

Parameters

- **table_name** – Name of the table to get

- **default** – Default value to return if `table_name` is not found.

Returns: a table if `table_name` exists, otherwise 'None' or the passed-in default

items()
Iterator of `(table_name, table)` pairs

keys()
Iterator of table names in this db

metadata
`MetaData` for this database

This can be used for introspection of reflected items.

Note:

Tables that haven't been reflected won't show up in metadata. To reflect a table, use `reflect_table()`.

This attribute is lazily evaluated and cached.

reflect_table(*table_name*)
Populate `metadata` with information on a table

Parameters **table_name** – The name of a table to reflect

session
Returns a `Session`

This is used for database queries. For writing to the database, start a `transaction()`.

Note:

This attribute is cached. In cases where a new session needs to be explicitly created, use `sessionmaker()`.

This attribute is lazily evaluated and cached.

sessionmaker
A `sessionmaker`

Used to make new sessions with this database, as needed.

This attribute is lazily evaluated and cached.

table_base
Base class for all tables returned by this database

This base class is created using `declarative_base`.

This attribute is lazily evaluated and cached.

table_names
A sorted list of table names available in this database.

This attribute is lazily evaluated and cached.

transaction
Context manager for simple transaction management

Sessions understand the concept of transactions, and provider context managers to handle conditionally committing or rolling back transactions as needed.

Note: Sessions automatically commit transactions by default. For predictable results when writing to the database, use the transaction manager.

Usage:

```
with db.transaction:
    db.session.do_something()
```

values()
Iterator of tables in this db

utils.db.**cfmedb**()

utils.db.**database_on_server**(**args*, ***kwds*)

utils.db.**db_yamls**(*db=None*, *guid=None*)
Returns the yamls from the db configuration table as a dict

Usage:

```
# Get all the yaml configs
configs = db_yamls

# Get all the yaml names
configs.keys()

# Retrieve a specific yaml (but you should use get_yaml_config here)
vmdb_config = configs['vmdb']
```

utils.db.**get_yaml_config**(*config_name*, *db=None*)
Return a specific yaml from the db configuration table as a dict

Usage:

```
# Retrieve a specific yaml
vmdb_config = get_yaml_config('vmdb')
```

utils.db.**ping_connection**(*dbapi_connection*, *connection_record*, *connection_proxy*)
ping_connection event hook, used to reconnect db sessions that time out

---

Note: See also: *Connection Invalidation*

---

utils.db.**scl_name**()

utils.db.**set_yaml_config**(*config_name*, *data_dict*, *hostname=None*)
Given a yaml name, dictionary and hostname, set the configuration yaml on the server

The configuration yamls must be inserted into the DB using the ruby console, so this function uses SSH, not the database. It makes sense to be included here as a counterpart to `get_yaml_config()`

Parameters

- **config_name** – Name of the yaml configuration file
- **data_dict** – Dictionary with data to set/change
- **hostname** – Hostname/address of the server that we want to set up (default `None`)

Note: If hostname is set to `None`, the default server set up for this session will be used. See :py:class:`utils.ssh.SSHClient` for details of the default setup.

Warning: Manually editing the config yamls is potentially dangerous. Furthermore, the rails runner doesn't return useful information on the outcome of the set request, so errors that arise from the newly loading config file will go unreported.

Usage:

```
# Update the appliance name, for example
vmbd_yaml = get_yaml_config('vmdb')
vmdb_yaml['server']['name'] = 'EVM IS AWESOME'
set_yaml_config('vmdb', vmdb_yaml, '1.2.3.4')
```

utils.db_queries module

utils.db_queries.**check_domain_enabled**(*domain*, *ip_address=None*, *db=None*)

utils.db_queries.**get_configuration_details**(*db=None*, *ip_address=None*)
Return details that are necessary to navigate through Configuration accordions.

Parameters **ip_address** – IP address of the server to match. If None, uses hostname from `conf.env['base_url']`

Returns If the data weren't found in the DB, `NoneType` If the data were found, it returns tuple *(region, server name, server id, server zone id)*

utils.db_queries.**get_host_id**(*hostname*, *ip_address=None*, *db=None*)

utils.db_queries.**get_zone_description**(*zone_id*, *ip_address=None*, *db=None*)

utils.error module

Handles errors based on something beyond the type. You can match error messages with regular expressions. You can also extend the matching behavior however you like. By default, strings are treated as regex and matched against the message of the error. Functions are passed the error and if the function returns 'truthy', then the error is caught.

Usage:

```
import utils.error as error
with error.expected('foo'):
    x = 1
    raise Exception('oh noes foo happened!')  # this will be caught because regex matches

with error.expected('foo'):
    raise Exception('oh noes bar happened!')  # this will bubble up because it doesn't match

with error.expected('foo'):
    pass  # an error will be thrown because we expected an error but there wasn't one.
```

exception utils.error.**UnexpectedSuccessException**
Bases: exceptions.Exception

An error that is thrown when something we expected to fail didn't fail.

utils.error.**expected**(**args*, ***kwds*)
Inverts error handling. If the enclosed block doesn't raise an error, it will raise one. If it raises a matching error, it will return normally. If it raises a non-matching error, that error will be allowed to propagate up the stack.

utils.error.**handler**(**args*, ***kwds*)
Handles errors based on more than just their type. Any matching error will be caught, the rest will be allowed to propagate up the stack.

utils.error.**regex**(*expr*, *e*)
Search the message of the exception using the regex expr

utils.events module

utils.ext_auth module

utils.ext_auth.**disable_external_auth_ipa**()
Unconfigure external auth.

utils.ext_auth.**setup_external_auth_ipa**(***data*)
Sets up the appliance for an external authentication with IPA.

Keywords: get_groups: Get User Groups from External Authentication (httpd). ipaserver: IPA server address. iparealm: Realm. credentials: Key of the credential in credentials.yaml

utils.ftp module

FTP manipulation library

@author: Milan Falešník <mfalesni@redhat.com>

class utils.ftp.**FTPClient**(*host*, *login*, *password*)
Bases: object

FTP Client encapsulation

This class provides basic encapsulation around ftplib's FTP class. It wraps some methods and allows to easily delete whole directory or walk through the directory tree.

Usage:

```
>>> from utils.ftp import FTPClient
>>> ftp = FTPClient("host", "user", "password")
>>> only_files_with_EVM_in_name = ftp.filesystem.search("EVM", directories=False)
>>> only_files_by_regexp = ftp.filesystem.search(re.compile("regexp"), directories=False)
>>> some_directory = ftp.filesystem.cd("a/b/c") # cd's to this directory
>>> root = some_directory.cd("/")
```

Always going through filesystem property is a bit slow as it parses the structure on each use. If you are sure that the structure will remain intact between uses, you can do as follows to save the time:

```
>>> fs = ftp.filesystem
```

Let's download some files:

```
>>> for f in ftp.filesystem.search("IMPORTANT_FILE", directories=False):
...     f.download()    # To pickup its original name
...     f.download("custom_name")
```

We finished the testing, so we don't need the content of the directory:

```
>>> ftp.recursively_delete()
```

And it's gone.

cdup()
Goes one level up in directory hierarchy (cd ..)

close()
Finish work and close connection

connect()

cwd(*d*)
Enter a directory

Parameters **d** – Directory name

Returns Success of the action

dele(*f*)
Remove a file

Parameters **f** – File name

Returns Success of the action

filesystem
Returns the object structure of the filesystem

Returns Root directory

ls()
Lists the content of a directory.

Returns List of all items in current directory Return format is [(is_dir?, "name", remote_time), ...]

mkd(*d*)
Create a directory

Parameters **d** – Directory name

Returns Success of the action

pwd()
Get current directory

Returns Current directory

Raises `AssertionError` – PWD command fails

recursively_delete(*d=None*)
Recursively deletes content of pwd

WARNING: Destructive!

Parameters

- **d** – Directory to enter (None for not entering - root directory)
- **d** – str or None

Raises `AssertionError` – When some of the FTP commands fail.

retrbinary(*f*, *callback*)
Download file

You need to specify the callback function, which accepts one parameter (data), to be processed.

Parameters

- **f** – Requested file name
- **callback** – Callable with one parameter accepting the data

rmd(*d*)
Remove a directory

Parameters **d** – Directory name

Returns Success of the action

storbinary(*f*, *file_obj*)
Store file

You need to specify the file object.

Parameters

- **f** – Requested file name
- **file_obj** – File object to be stored

tree(*d=None*)
Walks the tree recursively and creates a tree

Base structure is a list. List contains directory content and the type decides whether it's a directory or a file: - tuple: it's a file, therefore it represents file's name and time - dict: it's a directory. Then the dict structure is as follows:

```
dir: directory name
content: list of directory content (recurse)
```

Parameters **d** – Directory to enter(None for no entering - root directory)

Returns Directory structure in lists nad dicts.

Raises `AssertionError` – When some of the FTP commands fail.

update_time_difference()
Determine the time difference between the FTP server and this computer.

This is done by uploading a fake file, reading its time and deleting it. Then the self.dt variable captures the time you need to ADD to the remote time or SUBTRACT from local time.

The FTPFile object carries this automatically as it has .local_time property which adds the client's .dt to its time.

class utils.ftp.**FTPDirectory**(*client*, *name*, *items*, *parent_dir=None*, *time=None*)
Bases: object

FTP FS Directory encapsulation

This class represents one directory. Contains pointers to all child directories (self.directories) and also all files in current directory (self.files)

cd(*path*)
Change to a directory

Changes directory to a path specified by parameter path. There are three special cases: / - climbs by self.parent_dir up in the hierarchy until it reaches root element. . - does nothing .. - climbs one level up in hierarchy, if present, otherwise does the same as preceeding.

Parameters **path** – Path to change

path
Returns – whole path for this directory

search(*by*, *files=True*, *directories=True*)
Recursive search by string or regexp.

Searches throughout all the filesystem structure from top till the bottom until it finds required files or dirctories. You can specify either plain string or regexp. String search does classic `in`, regexp matching is done by exact matching (by.match).

Parameters

- **by** – Search string or regexp
- **files** – Whether look for files
- **directories** – Whether look for directories

Returns List of all objects found in FS

exception utils.ftp.**FTPException**
Bases: exceptions.Exception

class utils.ftp.**FTPFile**(*client*, *name*, *parent_dir*, *time*)
Bases: object

FTP FS File encapsulation

This class represents one file in the FS hierarchy. It encapsulates mainly its position in FS and adds the possibility of downloading the file.

download(*target=None*)
Download file into this machine

Wrapper around self.retr function. It downloads the file from remote filesystem into local filesystem. Name is either preserved original, or can be changed.

Parameters **target** – Target file name (None to preserver the original)

local_time
Returns – time modified to match local computer's time zone

path
Returns – whole path for this file

retr(*callback*)
Retrieve file

Wrapper around ftplib.FTP.retrbinary(). This function cd's to the directory where this file is present, then calls the FTP's retrbinary() function with provided callable and then cd's back where it started to keep it consistent.

Parameters **callback** – Any callable that accepts one parameter as the data

Raises

- `AssertionError` – When any of the CWD or CDUP commands fail.
- `ftplib.error_perm` – When retrbinary call of ftplib fails

utils.hosts module

utils.hosts

utils.hosts.**get_host_data_by_name**(*provider_key*, *host_name*)

utils.hosts.**setup_all_provider_hosts_credentials**()

utils.hosts.**setup_host_creds**(*provider_key*, *host_name*, *ignore_errors=False*)

utils.hosts.**setup_providers_hosts_credentials**(*provider_key*, *ignore_errors=False*)

utils.ipmi module

class utils.ipmi.**IPMI**(*hostname*, *username*, *password*, *interface_type='lan'*, *timeout=30*)
Utility to access IPMI via CLI.

The IPMI utility uses the `ipmitool` package to access the remote management card of a server.

Parameters

- **hostname** – The hostname of the remote management console.
- **username** – The username for the remote management console.
- **password** – The password tied to the username.
- **interface_type** – A string giving the `interface_type` to pass to the CLI.
- **timeout** – The number of seconds to wait before giving up on a command.

Returns: A `IPMI` instnace.

is_power_on()
Checks if the power is on.

Returns: `True` if power is on, `False` if not.

power_off()
Turns the power off.

Returns: `True` if power is off, `False` if not.

power_on()
Turns the power on.

Returns: `True` if power is on, `False` if not.

power_reset()
Turns the power off.

Returns: `True` if power reset initiated, `False` if not.

exception utils.ipmi.**IPMIException**
Bases: `exceptions.Exception`

Raised during `_run_ipmi()` if the error code is non zero.

utils.log module

Logging framework

This module creates the cfme logger, for use throughout the project. This logger only captures log messages explicitly sent to it, not logs emitted by other components (such as selenium). To capture those, consider using the pytest-capturelog plugin.

Example Usage

```
from utils.log import logger

logger.debug('debug log message')
logger.info('info log message')
logger.warning('warning log message')
logger.error('error log message')
logger.critical('critical log message')
```

The above will result in the following output in `cfme_tests/logs/cfme.log`:

```
1970-01-01 00:00:00,000 [D] debug log message (filename.py:3)
1970-01-01 00:00:00,000 [I] info log message (filename.py:4)
1970-01-01 00:00:00,000 [W] warning log message (filename.py:5)
1970-01-01 00:00:00,000 [E] error log message (filename.py:6)
1970-01-01 00:00:00,000 [C] fatal log message (filename.py:7)
```

Additionally, if `log_error_to_console` is True (see below), the following will be written to stderr:

```
[E] error (filename.py:6)
[C] fatal (filename.py:7)
```

Log Message Source

We have added a custom log record attribute that can be used in log messages: `%(source)s` This attribute is included in the default 'cfme' logger configuration.

This attribute will be generated by default and include the filename and line number from where the log message was emitted. It will attempt to convert file paths to be relative to cfme_tests, but use the absolute file path if a relative path can't be determined.

When writting generic logging facilities, it is sometimes helpful to override those source locations to make the resultant log message more useful. To do so, pass the extra `source_file` (str) and `source_lineno` (int) to the log emission:

```
logger.info('info log message', extra={'source_file': 'somefilename.py', 'source_lineno': 7})
```

If `source_lineno` is `None` and `source_file` is included, the line number will be omitted. This is useful in cases where the line number can't be determined or isn't necessary.

Configuration

```
# in env.yaml
logging:
    # Can be one of DEBUG, INFO, WARNING, ERROR, CRITICAL
    level: INFO
```

```
    # Maximum logfile size, in bytes, before starting a new logfile
    # Set to 0 to disable log rotation
    max_logfile_size: 0
    # Maximimum backup copies to make of rotated log files (e.g. cfme.log.1, cfme.log.2, ...)
    # Set to 0 to keep no backups
    max_logfile_backups: 0
    # If True, messages of level ERROR and CRITICAL are also written to stderr
    errors_to_console: False
    # Default file format
    file_format: "%(asctime)-15s [%(levelname).1s] %(message)s (%(source)s)"
    # Default format to console if errors_to_console is True
    stream_format: "[%(levelname)s] %(message)s (%(source)s)"
```

Additionally, individual logger configurations can be overridden by defining nested configuration values using the logger name as the configuration key. Note that the name of the logger objects exposed by this module don't obviously line up with their key in `cfme_data`. The 'name' attribute of loggers can be inspected to get this value:

```
>>> utils.log.logger.name
'cfme'
>>> utils.log.perflog.logger.name
'perf'
```

Here's an example of those names being used in `env.local.yaml` to configure loggers individually:

```
logging:
    cfme:
        # set the cfme log level to debug
        level: DEBUG
    perf:
        # make the perflog a little more "to the point"
        file_format: "%(message)s"
```

Notes:

- The `cfme` and `perf` loggers are guaranteed to exist when using this module.
- The name of a logger is used to generate its filename, and will usually not have the word "log" in it.
  - `perflog`'s logger name is `perf` for this reason, resulting in `log/perf.log` instead of `log/perflog.log`.
  - Similarly, `logger`'s' name is `cfme`, to prevent having `log/logger.log`.

Warning: Creating a logger with the same name as one of the default configuration keys, e.g. `create_logger('level')` will cause a rift in space-time (or a ValueError).
Do not attempt.

Message Format

```
year-month-day hour:minute:second,millisecond [Level] message text
(file:linenumber)
```

`[Level]`:

One letter in square brackets, where `[I]` corresponds to INFO, `[D]` corresponds to DEBUG, and so on.

`(file:linenumber)`:

The relative location from which this log message was emitted. Paths outside

Members

class utils.log.**ArtifactorLoggerAdapter**(*logger*, *extra*)
Bases: logging.LoggerAdapter

Logger Adapter that hands messages off to the artifactor before logging

art_log(*level_name*, *message*, *kwargs*)

artifactor

critical(*msg*, **args*, ***kwargs*)

debug(*msg*, **args*, ***kwargs*)

error(*msg*, **args*, ***kwargs*)

exception(*msg*, **args*, ***kwargs*)

info(*msg*, **args*, ***kwargs*)

log(*lvl*, *msg*, **args*, ***kwargs*)

process(*msg*, *kwargs*)

slaveid

trace(*msg*, **args*, ***kwargs*)

warning(*msg*, **args*, ***kwargs*)

class utils.log.**NamedLoggerAdapter**(*logger*, *extra*)
Bases: utils.log.TraceLoggerAdapter

An adapter that injects a name into log messages

process(*message*, *kwargs*)

class utils.log.**Perflog**(*perflog_name='perf'*)
Bases: object

Performance logger, useful for timing arbitrary events by name

Logged events will be written to log/perf.log by default, unless a different log file name is passed to the Perflog initializer.

Usage:

```
from utils.log import perflog
perflog.start('event_name')
# do stuff
seconds_taken = perflog.stop('event_name')
# seconds_taken is also written to perf.log for later analysis
```

start(*event_name*)
Start tracking the named event

Will reset the start time if the event is already being tracked

stop(*event_name*)
Stop tracking the named event

Returns A float value of the time passed since start was last called, in seconds, *or* None if start was never called.

tracking_events = {}

class utils.log.**SyslogMsecFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

A custom Formatter for the syslogger which changes the log timestamps to have millisecond resolution for compatibility with splunk.

static converter()
timestamp[, tz] -> tz's local time from POSIX timestamp.

formatTime(*record*, *datefmt=None*)

class utils.log.**TraceLogger**(*name*, *level=0*)
Bases: logging.Logger

A trace-loglevel-aware Logger

trace(*msg*, **args*, ***kwargs*)
Log 'msg % args' with severity 'TRACE'.

class utils.log.**TraceLoggerAdapter**(*logger*, *extra*)
Bases: logging.LoggerAdapter

A trace-loglevel-aware LoggerAdapter

trace(*msg*, **args*, ***kwargs*)
Delegate a trace call to the underlying logger, after adding contextual information from this adapter instance.

utils.log.**create_logger**(*logger_name*, *filename=None*, *max_file_size=None*, *max_backups=None*)
Creates and returns the named logger

If the logger already exists, it will be destroyed and recreated with the current config in env.yaml

utils.log.**create_sublogger**(*logger_sub_name*, *logger_name='cfme'*)

utils.log.**format_marker**(*mstring*, *mark='-'*)
Creates a marker in log files using a string and leader mark.

This function uses the constant MARKER_LEN to determine the length of the marker, and then centers the message string between padding made up of leader_mark characters.

Parameters

- **mstring** – The message string to be placed in the marker.
- **leader_mark** – The marker character to use for leading and trailing.

Returns: The formatted marker string.

Note: If the message string is too long to fit one character of leader/trailer and a space, then the message is returned as is.

class utils.log.**logger_wrap**(**args*, ***kwargs*)
Bases: object

Sets up the logger by default, used as a decorator in utils.appliance

If the logger doesn't exist, sets up a sensible alternative

utils.log.**nth_frame_info**(*n*)
Inspect the stack to determine the filename and lineno of the code running at the "n"th frame

Parameters **n** – Number of the stack frame to inspect

Raises IndexError if the stack doesn't contain the nth frame (the caller should know this)

Returns a frameinfo namedtuple as described in inspect

utils.miq_soap module

SOAP wrapper for CFME.

Enables to operate Infrastructure objects. It has better VM provisioning code. OOP encapsulated.

class utils.miq_soap.**BelongsToCluster**(*id*)
Bases: utils.miq_soap.BelongsToProvider

cluster

class utils.miq_soap.**BelongsToProvider**(*id*)
Bases: utils.miq_soap.MiqInfraObject

provider

class utils.miq_soap.**HasManyDatastores**(*id*)
Bases: utils.miq_soap.MiqInfraObject

datastores

class utils.miq_soap.**HasManyEMSs**(*id*)
Bases: utils.miq_soap.MiqInfraObject

emss

class utils.miq_soap.**HasManyHosts**(*id*)
Bases: utils.miq_soap.MiqInfraObject

hosts

class utils.miq_soap.**HasManyResourcePools**(*id*)
Bases: utils.miq_soap.MiqInfraObject

resource_pools

class utils.miq_soap.**HasManyVMs**(*id*)
Bases: utils.miq_soap.MiqInfraObject

vms

class utils.miq_soap.**MiqCluster**(*id*)
Bases: utils.miq_soap.HasManyDatastores, utils.miq_soap.HasManyHosts, utils.miq_soap.HasManyVMs, utils.miq_soap.HasManyResourcePools, utils.miq_soap.BelongsToProvider

GETTER_FUNC = 'FindClusterById'

TAG_PREFIX = 'Cluster'

classmethod all()

default_resource_pool

class utils.miq_soap.**MiqDatastore**(*id*)
Bases: utils.miq_soap.HasManyHosts, utils.miq_soap.HasManyEMSs

GETTER_FUNC = 'FindDatastoreById'

TAG_PREFIX = 'Datastore'

classmethod all()

class utils.miq_soap.**MiqEms**(*id*)
Bases: utils.miq_soap.HasManyDatastores, utils.miq_soap.HasManyHosts, utils.miq_soap.HasManyVMs, utils.miq_soap.HasManyResourcePools

GETTER_FUNC = 'FindEmsByGuid'

TAG_PREFIX = 'Ems'

classmethod all()

clusters

direct_connection
Returns an API from mgmt_system.py targeted at this provider

This attribute is lazily evaluated and cached.

classmethod find_by_name(*name*)

host_name

ip_address

port

class utils.miq_soap.**MiqHost**(*id*)
Bases: utils.miq_soap.HasManyDatastores, utils.miq_soap.HasManyVMs, utils.miq_soap.HasManyResourcePools, utils.miq_soap.BelongsToCluster

GETTER_FUNC = 'FindHostByGuid'

TAG_PREFIX = 'Host'

classmethod all()

class utils.miq_soap.**MiqInfraObject**(*id*)
Bases: object

Base class for all infrastructure objects.

Parameters **id** – GUID or ID of the object, it depends on what does the particular SOAP function wants.

GETTER_FUNC = None

TAG_PREFIX = None

add_tag(*tag*)
Add tag to the object

Parameters **tag** – Tuple with tag specification.

exists

id

name

object
Accesses SOAP object

Accesses network.

tags
Return tags as an array of MiqTag objects.

ws_attributes
Processes object.ws_attributes into builtin types

class utils.miq_soap.**MiqResourcePool**(*id*)
Bases: utils.miq_soap.HasManyHosts, utils.miq_soap.HasManyEMSs

GETTER_FUNC = 'FindResourcePoolById'

TAG_PREFIX = 'ResourcePool'

classmethod all()

store_type

class utils.miq_soap.**MiqTag**(*category*, *category_dn*, *tag_name*, *tag_dname*, *tag_path*, *dn*)
Bases: object

class utils.miq_soap.**MiqVM**(*id*)
Bases: utils.miq_soap.HasManyDatastores, utils.miq_soap.BelongsToCluster

GETTER_FUNC = 'FindVmByGuid'

TAG_PREFIX = 'Vm'

delete()
Delete the VM from VMDB. To completely delete, use direct_connection.

description

host

is_powered_off

is_powered_on

is_suspended

power_off()

power_on()

classmethod provision_from_template(*template_name*, *vm_name*, *wait_min=None*, *cpus=1*, *memory=1024*, *vlan=None*, *first_name='Shadowman'*, *last_name='RedHat'*, *email='shadowm@n.redhat.com'*)
Provision VM from template.

Works independently on the management system, tags appropriate VMDB objects to provision without problems.

Parameters

- **template_name** – Name of the template to use.
- **vm_name** – VM Name.
- **wait_min** – How many minutes of wait for the provisioning to finish.
- **cpus** – How many CPUs should the VM have.
- **memory** – How much memory (in MB) should the VM have.
- **vlan** – Where to connect the VM. Obligatory for RHEV
- **first_name** – Name of the requestee
- **last_name** – Surname of the requestee
- **email** – Email of the requestee

Returns: MiqVM object with freshly provisioned VM.

suspend()

vendor

wait_powered_off(*wait_time=120*)

wait_powered_on(*wait_time=120*)

wait_suspended(*wait_time=160*)

utils.miq_soap.**get_client**()

utils.miq_soap.**is_datastore_banned**(*datastore_name*)
Checks whether the datastore is in the list of datastores not allowed to use

Parameters **datastore_name** – Name of the datastore

Returns: bool

utils.miq_soap.**set_client**(*client*)

utils.net module

utils.net.**ip_echo_socket**(*port=32123*)
A simple socket server, for use with my_ip_address()

utils.net.**my_ip_address**(*http=False*)
Get the ip address of the host running tests using the service listed in cfme_data['ip_echo']

The ip echo endpoint is expected to write the ip address to the socket and close the connection. See a working example of this in ip_echo_socket().

utils.net.**net_check**(*port*, *addr=None*, *force=False*)
Checks the availablility of a port

utils.net.**net_check_remote**(*port*, *addr=None*, *machine_addr=None*, *ssh_creds=None*, *force=False*)
Checks the availability of a port from outside using another machine (over SSH)

utils.net.**random_port**(*tcp=True*)
Get a random port number for making a socket

Parameters **tcp** – Return a TCP port number if True, UDP if False

This may not be reliable at all due to an inherent race condition. This works by creating a socket on an ephemeral port, inspecting it to see what port was used, closing it, and returning that port number. In the time between closing the socket and opening a new one, it's possible for the OS to reopen that port for another purpose.

In practical testing, this race condition did not result in a failure to (re)open the returned port number, making this solution squarely "good enough for now".

utils.net.**resolve_hostname**(*hostname*, *force=False*)
Cached DNS resolver. If the hostname does not resolve to an IP, returns None.

utils.net.**resolve_ips**(*host_iterable*, *force_dns=False*)
Takes list of hostnames, ips and another things. If the item is not an IP, it will be tried to be converted to an IP. If that succeeds, it is appended to the set together with original hostname. If it can't be resolved, just the original hostname is appended.

utils.pagestats module

Functions and PageStat object for performance testing of the UI.

class utils.pagestats.**PageStat**(*request=''*, *status=''*, *seleniumtime=0*, *completedintime=0*, *viewstime=0*, *activerecordtime=0*, *selectcount=0*, *cachedcount=0*, *uncachedcount=0*)

Bases: object

Object that represents page statistics and a list of any associated slow queries.

class utils.pagestats.**PageStatLists**

Bases: object

utils.pagestats.**analyze_page_stat**(*pages*, *soft_assert*)

utils.pagestats.**any_in**(*items*, *thing*)

utils.pagestats.**generate_tree_paths**(*tree_contents*, *path*, *paths*)

utils.pagestats.**navigate_accordions**(*accordions*, *page_name*, *ui_bench_pg_limit*, *ui_worker_pid*, *prod_tail*, *soft_assert*)

utils.pagestats.**navigate_quadicons**(*q_names*, *q_type*, *page_name*, *nav_limit*, *ui_worker_pid*, *prod_tail*, *soft_assert*, *acc_topbars=[]*)

utils.pagestats.**navigate_split_table**(*table*, *page_name*, *nav_limit*, *ui_worker_pid*, *prod_tail*, *soft_assert*)

utils.pagestats.**pages_to_csv**(*pages*, *file_name*)

utils.pagestats.**pages_to_statistics_csv**(*pages*, *filters*, *report_file_name*)

utils.pagestats.**perf_bench_read_tree**(*tree*)

utils.pagestats.**perf_click**(*uiworker_pid*, *tailer*, *measure_sel_time*, *clickable*, **args*)

utils.pagestats.**standup_perf_ui**(*ui_worker_pid*, *soft_assert*)

utils.path module

Project path helpers

Contains py.path.local objects for accessing common project locations.

Paths rendered below will be different in your local environment.

utils.path.**conf_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/conf')**

conf yaml storage, cfme_tests/conf/

utils.path.**data_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/data')**

datafile storage, cfme_tests/data/

utils.path.**docs_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/docs')**

doc root, where these file came from! cfme_tests/docs/

utils.path.**get_rel_path**(*absolute_path_str*)

Get a relative path for object in the project root

Parameters **absolute_path_str** – An absolute path to a file anywhere under *project_path*

Note: This will be a no-op for files that are not in *project_path*

utils.path.**log_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/log')**
log storage, cfme_tests/log/

utils.path.**project_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest')**
The project root, cfme_tests/

utils.path.**scripts_data_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest**
interactive scripts' data, cfme_tests/scripts/data

utils.path.**scripts_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/scripts**
interactive scripts, cfme_tests/scripts/

utils.path.**template_path = local('/home/docs/checkouts/readthedocs.org/user_builds/cfme-user1/checkouts/latest/data/**
jinja2 templates, use with jinja2.FileSystemLoader

utils.perf module

Functions that performance tests use.

utils.perf.**collect_log**(*ssh_client*, *log_prefix*, *local_file_name*, *strip_whitespace=False*)
Collects all of the logs associated with a single log prefix (ex. evm or top_output) and combines to single gzip log file. The log file is then scp-ed back to the host.

utils.perf.**convert_top_mem_to_mib**(*top_mem*)
Takes a top memory unit from top_output.log and converts it to MiB

utils.perf.**generate_statistics**(*the_list*, *decimals=2*)
Returns comma seperated statistics over a list of numbers.

Returns: list of samples(runs), minimum, average, median, maximum, stddev, 90th(percentile), 99th(percentile)

utils.perf.**get_worker_pid**(*worker_type*)
Obtains the pid of the first worker with the worker_type specified

utils.perf.**set_rails_loglevel**(*level*, *validate_against_worker='MiqUiWorker'*)
Sets the logging level for level_rails and detects when change occured.

utils.perf_message_stats module

Functions for performance analysis/charting of the backend messages and top_output from an appliance.

class utils.perf_message_stats.**MiqMsgBucket**
Bases: object

class utils.perf_message_stats.**MiqMsgLists**
Bases: object

class utils.perf_message_stats.**MiqMsgStat**
Bases: object

class utils.perf_message_stats.**MiqWorker**
Bases: object

utils.perf_message_stats.**evm_to_messages**(*evm_file*, *filters*)

utils.perf_message_stats.**evm_to_workers**(*evm_file*)

utils.perf_message_stats.**generate_appliance_charts**(*top_appliance*, *charts_dir*, *start_index*, *end_index*)

utils.perf_message_stats.**generate_hourly_charts_and_csvs**(*hourly_buckets*, *charts_dir*)

utils.perf_message_stats.**generate_raw_data_csv**(*rawdata_dict*, *csv_file_name*)

utils.perf_message_stats.**generate_total_time_charts**(*msg_cmds*, *charts_dir*)

utils.perf_message_stats.**generate_worker_charts**(*workers*, *top_workers*, *charts_dir*)

utils.perf_message_stats.**get_first_miqtop**(*top_log_file*)

utils.perf_message_stats.**get_msg_args**(*log_line*)

utils.perf_message_stats.**get_msg_cmd**(*log_line*)

utils.perf_message_stats.**get_msg_del**(*log_line*)

utils.perf_message_stats.**get_msg_deq**(*log_line*)

utils.perf_message_stats.**get_msg_id**(*log_line*)

utils.perf_message_stats.**get_msg_timestamp_pid**(*log_line*)

utils.perf_message_stats.**hour_bucket_init**(*init*)

utils.perf_message_stats.**line_chart_render**(*title*, *xtitle*, *ytitle*, *x_labels*, *lines*, *fname*, *stacked=False*)

utils.perf_message_stats.**messages_to_hourly_buckets**(*messages*, *test_start*, *test_end*)

utils.perf_message_stats.**messages_to_statistics_csv**(*messages*, *statistics_file_name*)

utils.perf_message_stats.**perf_process_evm**(*evm_file*, *top_file*)

utils.perf_message_stats.**provision_hour_buckets**(*test_start*, *test_end*, *init=True*)

utils.perf_message_stats.**split_appliance_charts**(*top_appliance*, *charts_dir*)

utils.perf_message_stats.**top_to_appliance**(*top_file*)

utils.perf_message_stats.**top_to_workers**(*workers*, *top_file*)

utils.ports module

utils.pretty module

class utils.pretty.**Pretty**
Bases: object

A mixin that prints repr as <MyClass field1=..., field2=...>. The fields that will be printed should be stored in the class's pretty_attrs attribute (none by default).

pretty_attrs = []

utils.pretty.**attr_repr**(*o*, *attr*)
Return the string repr of the attribute attr on the object o

utils.pretty.**pr_obj**(*attrs*)

utils.pretty.**pretty_repr**(*attrs*, *o*)

utils.providers module

utils.pytest_shortcuts module

utils.pytest_shortcuts.**extract_fixtures_values**(*item*)
Extracts names and values of all the fixtures that the test has.

Parameters **item** – py.test test item

Returns `dict` with fixtures and their values.

utils.signals module

Simple callback handler routine

Signals are a simple way of notifying the framework that something has happened. Currently we are using them to notify and take action when we _know_ that certain caches have become stale and need to be invalidated. The example below shows this.

```
import signals
from fixtures.pytest_store import store

def invalidate_server_details():
    del store.current_appliance.configuration_details
    del store.current_appliance.zone_description


signals.register_callback('server_details_changed', invalidate_server_details)
```

Or by using a decorator:

```
from signals import on_signal
from fixtures.pytest_store import store

@on_signal("server_details_changed")
def invalidate_server_details():
    del store.current_appliance.configuration_details
    del store.current_appliance.zone_description
```

Here we create a function to do the work of invalidating the cache and register it to the signal name 'server_details_changed'. Now whenever something in the framework changes anything to do with server details it will use the fire function like so.

```
import signals

signals.fire('server_details_changed')
```

The user who fires off the signal doesn't need to worry about what should happen when the server details change. They fire the signal and the framework will take the appropriate action as defined in the callback handler.

Mutliple callbacks can be assigned to the same signal, and can be augmented with args and kwargs to be able to pass extra information to the callback function.

Current list of signals defined and their usage

Name	Usage
server_details_changed	Signal used when the main details of a server has been changed, name etc
server_config_changed	Signal used when the main configuration yaml has been altered

utils.signals.**fire**(*signal*)
Fires the signal, invoking all callbacks in the library for the signal.

Parameters **signal** – Name of signal to be invoked.

utils.signals.**on_signal**(*signal*, **args*, ***kwargs*)
Decorator for register_callback usage.

utils.signals.**register_callback**(*signal*, *cb_func*, **args*, ***kwargs*)
Register a callback function to a signal name

Parameters

- **signal** – The name of the signal.
- **cb_func** – The function object to be called.
- **args** – Any args, passed to the cb_func on calling
- **kwargs** – Any kwargs, passed to the cb_func on calling

Returns: A callback object.

utils.signals.**unregister_callback**(*cb_obj*)
Unregisters a callback object from the library.

Given a callback object, an attempt will be made to remove it from the callback library.

Parameters **cb_obj** – A callback object to be removed.

utils.smtp_collector_client module

class utils.smtp_collector_client.**SMTPCollectorClient**(*host='localhost'*, *port=1026*)
Bases: object

Client for smtp_collector.py script

Parameters

- **host** – Host where collector runs (Default: localhost)
- **port** – Port where the collector query interface listens (Default: 1026)

clear_database()
Clear the database in collector

Returns: bool

get_emails(***filter*)
Get emails. Eventually apply filtering on SQLite level

Time variables can be passed as instances of utils.timeutil.parsetime. That carries out the necessary conversion automatically.

_like args - see SQLite's LIKE operator syntax

Keywords: from_address: E-mail matches. to_address: E-mail matches. subject: Subject matches exactly. subject_like: Subject is LIKE. time_from: E-mails arrived since this time. time_to: E-mail arrived before this time. text: Text matches exactly. text_like: Text is LIKE.

Returns: List of dicts with e-mails matching the criteria.

get_html_report()

set_test_name(*test_name*)
Set the test name for folder name in the collector.

Parameters **test_name** – Name to set

Returns: `bool` with result.

utils.snmp_client module

This module provides a client class for the SNMP listener

It automatically detects whether the listener is installed and if it is not, it installs it automatically.

class `utils.snmp_client.`**`SNMPClient`**(*addr*, *port=8765*)
Bases: `object`

Class for accessing the SNMP traps stored in the appliance listener

Parameters

- **addr** – Address of the appliance
- **port** – port to contact, 8765 by default

`get_all`()
Get all traps that were caught.

Returns: List of dicts.

`install`()
Install the listener to the appliance

`setup`

Checks for presence of the listener on the appliance. If it is not present, it then installs it.

This attribute is lazily evaluated and cached.

utils.soap module

class `utils.soap.`**`MiqClient`**(*url*, ***kwargs*)
Bases: `suds.client.Client`

static **`pipeoptions`**(*options_dict*)
Convert a flat dict into pipe-separated key=value pairs

Handy helper for making argument strings that the CFME soap API wants

Doesn't handle pipes in keys or values, so don't put any in them.

`utils.soap.`**`soap_client`**()
SoapClient to EVM based on base_url

utils.sprout module

class `utils.sprout.`**`APIMethodCall`**(*client*, *method_name*)
Bases: `object`

exception `utils.sprout.`**`AuthException`**
Bases: `utils.sprout.SproutException`

class `utils.sprout.`**`SproutClient`**(*protocol='http'*, *host='localhost'*, *port=8000*, *entry='appliances/api'*, *auth=None*)
Bases: `object`

api_entry

call_method(*name*, **args*, ***kwargs*)

classmethod from_config(***kwargs*)

exception utils.sprout.**SproutException**
Bases: exceptions.Exception

utils.ssh module

class utils.ssh.**SSHClient**(*stream_output=False*, ***connect_kwargs*)
Bases: paramiko.client.SSHClient

paramiko.SSHClient wrapper

Allows copying/overriding and use as a context manager Constructor kwargs are handed directly to paramiko.SSHClient.connect()

appliance_has_netapp()

client_address()

close()

connect(*hostname=None*, ***kwargs*)
See paramiko.SSHClient.connect

connected

get_build_date()

get_build_datetime()

get_file(*remote_file*, *local_path=''*, ***kwargs*)

get_transport(**args*, ***kwargs*)

is_appliance_downstream()

put_file(*local_file*, *remote_file='.'*, ***kwargs*)

run_command(*command*, *timeout=1200.0*)

run_rails_command(*command*, *timeout=1200.0*)

run_rake_command(*command*, *timeout=1200.0*)

status
Parses the output of the service evmserverd status.

Returns A dictionary containing servers and workers, both lists. Each of the lists contains dictionaries, one per line. You can refer inside the dictionary using the headers.

uptime()

class utils.ssh.**SSHResult**
Bases: tuple

SSHResult(rc, output)

output
Alias for field number 1

rc
Alias for field number 0

class utils.ssh.**SSHTail**(*remote_filename*, ***connect_kwargs*)
Bases: utils.ssh.SSHClient

set_initial_file_end()

utils.ssh.**keygen**()
Generate temporary ssh keypair for appliance SSH auth

Intended not only to simplify ssh access to appliances, but also to simplify SSH access from one appliance to another in multi-appliance setups

utils.stats module

utils.stats.**tol_check**(*ref*, *compare*, *min_error=0.05*, *low_val_correction=3.0*)
Tolerance check

The tolerance check is very simple. In essence it checks to ensure that the compare value is within min_error percentage of the ref value. However there are special conditions.

If the ref value is zero == the compare value we will alwys return True to avoid calculation overhead.

If the ref value is zero we check if the compare value is below the low_val_correction threshold.

The low value correction is also used if ref is small. In this case, if one minus the difference of the ref and low value correction / reference value yields greater error correction, then this is used.

For example, if the reference was 1 and the compare was 2, with a min_error set to the default, the tolerance check would return False. At low values this is probably undesirable and so, the low_val_correction allows for a greater amount of error at low values. As an example, with the lvc set to 3, the allowe error would be much higher, allowing the tolerance check to pass.

The lvc will only take effect if the error it produces is greater than the min_error.

Parameters

- **ref** – The reference value
- **compare** – The comparison value
- **min_error** – The minimum allowed error
- **low_val_correction** – A correction value for lower values

utils.storage_managers module

utils.storage_managers.**objects_from_config**(**keys*)

utils.storage_managers.**set_roles_for_sm**()

utils.storage_managers.**setup_storage_manager**(*key*)

utils.storage_managers.**setup_storage_managers**()

utils.testgen module

utils.timeutil module

This module should contain all things associated with time or date that can be shared.

utils.timeutil.**nice_seconds**(*t_s*)
Return nicer representation of seconds

class utils.timeutil.**parsetime**
Bases: datetime.datetime

Modified class with loaders for our datetime formats.

classmethod from_american_date_only(*time_string*)
Convert the string representation of the time into parsetime()

CFME's format here is 'mm/dd/yy'

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

classmethod from_american_minutes(*time_string*)
Convert the string representation of the time into parsetime()

CFME's format here is 'mm/dd/yy hh:mm'

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

classmethod from_american_with_utc(*time_string*)
Convert the string representation of the time into parsetime()

CFME's format here is 'mm/dd/yy hh:mm:ss UTC'

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

classmethod from_iso_date(*time_string*)
Convert the string representation of the time into parsetime()

Format here is 'YYYY-MM-DD'

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

classmethod from_iso_with_utc(*time_string*)
Convert the string representation of the time into parsetime()

CFME's format here is 'mm-dd-yy hh:mm:ss UTC'

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

classmethod from_request_format(*time_string*)
Convert the string representation of the time into parsetime()

Format here is 'YYYY-MM-DD-HH-MM-SS'. Used for transmitting data over http

Parameters **time_string** – String with time to parse

Returns: :py:class'utils.timeutil.datetime()' object

to_american_date_only()
Convert the this object to string representation in american date only format.

CFME's format here is 'mm/dd/yy'

Returns: :py:class'str' object

to_american_minutes()
Convert the this object to string representation in american with just minutes.

CFME's format here is 'mm/dd/yy hh:mm'

Returns: :py:class'str' object

to_american_with_utc()
Convert the this object to string representation in american with UTC.

CFME's format here is 'mm/dd/yy hh:mm:ss UTC'

Returns: :py:class'str' object

to_iso_date()
Convert the this object to string representation in ISO format.

Format here is 'YYYY-MM-DD'

Returns: :py:class'str' object

to_iso_with_utc()
Convert the this object to string representation in american with UTC.

CFME's format here is 'mm-dd-yy hh:mm:ss UTC'

Returns: :py:class'str' object

to_request_format()
Convert the this object to string representation in http request.

Format here is 'YYYY-MM-DD-HH-MM-SS'

Returns: :py:class'str' object

utils.trackerbot module

utils.update module

class utils.update.**Updateable**
Bases: object

A mixin that helps make an object easily updateable. Two Updateables are equal if all their public fields are equal.

utils.update.**all_public_fields_equal**(*a*, *b*)

utils.update.**public_fields**(*o*)
Returns: a dict of fields whose name don't start with underscore.

utils.update.**update**(**args*, ***kwds*)
Update an object and then sync it with an external application.

It will deepcopy the object into whatever is named in the 'as' clause, run the 'with' code block (which presumably alters the object). Then the update() method on the original object will be called with a dict containing only changed fields, and kwargs passed to this function.

If an exception is thrown by update(), the original object will be restored, otherwise the updated object will be returned.

Usage:

```
with update(myrecord):
   myrecord.lastname = 'Smith'
   myrecord.address.zipcode = '27707'
```

utils.update.**updates**(*old*, *new*)
Return a dict of fields that are different between old and new.

utils.version module

utils.version.**SPTuple**
alias of StreamProductTuple

class utils.version.**Version**(*vstring*)
Bases: object

Version class based on distutil.version.LooseVersion

component_re = <_sre.SRE_Pattern object at 0x7f52ef2d0768>

is_in_series(*series*)
This method checks whether the version belongs to another version's series.

Eg.: `Version("5.2.5.2").is_in_series("5.2")` returns `True`

Parameters **series** – Another `Version` to check against. If string provided, will be converted to `Version`

classmethod latest()

classmethod lowest()

parse(*vstring*)

product_version()

series(*n=2*)

stream()

utils.version.**appliance_build_date**()

utils.version.**appliance_build_datetime**()

utils.version.**appliance_has_netapp**()

utils.version.**appliance_is_downstream**()

utils.version.**before_date_or_version**(*date=None*, *version=None*)
Function for deciding based on the build date and version.

Usage:

```
* If both date and version are set, then two things can happen. If the appliance is
    downstream, both date and version are checked, otherwise only the date.
* If only date is set, then only date is checked.
* if only version is set, then it checks the version if the appliance is downstream,
    otherwise it returns ``False``
```

The checks are in form `appliance_build_date() < date` and `current_version() < version`. Therefore when used in `if` statement, the truthy value signalizes 'older' version and falsy signalizes 'newer' version.

utils.version.**current_stream**()

utils.version.**current_version**()
A lazy cached method to return the appliance version.

Do not catch errors, since generally we cannot proceed with testing, without knowing the server version.

utils.version.**dependent**(*default_function*)

utils.version.**get_product_version**(*ver*)
Return product version for given Version obj or version string

utils.version.**get_stream**(*ver*)
Return a stream name for given Version obj or version string

utils.version.**get_version**(*obj=None*)
Return a Version based on obj. For CFME, 'master' version means always the latest (compares as greater than any other version)

If obj is None, the version will be retrieved from the current appliance

utils.version.**parsedate**(*o*)

utils.version.**pick**(*v_dict*)
Collapses an ambiguous series of objects bound to specific versions by interrogating the CFME Version and returning the correct item.

utils.version.**product_version_dispatch**(**_args*, ***_kwargs*)
Dispatch function for use in multimethods that just ignores arguments and dispatches on the current product version.

utils.version.**since_date_or_version**(**args*, ***kwargs*)
Opposite of `before_date_or_version()`

utils.video module

Video recording library

Configuration for this module + fixture: .. code-block:: yaml

logging:

video: enabled: True dir: video display: ":99" quality: 10

class utils.video.**Recorder**(*filename*, *display=None*, *quality=None*)
Bases: `object`

Recorder class

Usage:

```
with Recorder(filename):
    # do something

# or
r = Recorder(filename)
r.start()
```

```
# do something
r.stop()
```

The first way is preferred, obviously

start()

stop()

utils.video.**process_running**(*pid*)
Check whether specified process is running

utils.virtual_machines module

utils.wait module

3.5.3 Module contents

utils.**at_exit**(*f*, **args*, ***kwargs*)
Diaper-protected atexit handler registering. Same syntax as atexit.register()

utils.**classproperty**(*f*)
Enables properties for whole classes:

Usage:

```
>>> class Foo(object):
...     @classproperty
...     def bar(cls):
...         return "bar"
...
>>> print Foo.bar
baz
```

utils.**deferred_verpick**(*version_d*)
This turns a dictionary for verpick to a class property.

Useful for verpicked constants.

class utils.**kwargify**(*function*)
Bases: object

utils.**lazycache**(*wrapped_method*)
method decorator to create a lazily-evaluated and cached property

lazycache'd properties are complete object descriptors, supporting get, set, and del, though del will clear a property's cache rather than destroy the property entirely

Usage:

```
>>> from utils import lazycache
>>> class Example(object):
...     @lazycache
...     def lazyprop(self):
...             return '42'
...
```

```
>>> ex = Example()
>>> value = ex.lazyprop
>>> print value
42
>>> print value is ex.lazyprop
# lazyprop guarantees this to be True, normal properties do not.
True
>>> ex.lazyprop = '99'
>>> print ex.lazyprop
# setting works!
99
>>> del(ex.lazyprop)
>>> print ex.lazyprop
# deleting clears the cache, so the value is recomputed on the next call
42
```

Values are stored in a private attribute of the same name as the method being decorated, e.g. a decorated method named `lazyprop` will store its cached value in an attr called `_lazyprop`

`utils.`**`normalize_text`**(*text*)

`utils.`**`property_or_none`**([*fget*[, *fset*[, *fdel*[, *doc*]]]])

Property decorator that turns AttributeErrors into None returns

Useful for chained attr lookups where some links in the chain are None

---

Note: This delegates back to the `property` builtin and inherits its signature; thus it can be used interchangeably with `property`.

`utils.`**`read_env`**(*file*)

Given a py.path.Local file name, return a dict of exported shell vars and their values

---

Note: This will only include shell variables that are exported from the file being parsed

---

Returns a dict of varname: value pairs. If the file does not exist or bash could not parse the file, this dict will be empty.

`utils.`**`tries`**(*num_tries*, *exceptions*, *f*, **args*, ***kwargs*)

Tries to call the function multiple times if specific exceptions occur.

Parameters

- **num_tries** – How many times to try if exception is raised
- **exceptions** – Tuple (or just single one) of exceptions that should be treated as repeat.
- **f** – Callable to be called.
- ***args** – Arguments to be passed through to the callable
- ****kwargs** – Keyword arguments to be passed through to the callable

Returns What `f` returns.

Raises What `f` raises if the try count is exceeded.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

C

f

m

u

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z