Pulp Smash Documentation

Release 2017.11.16

Jeremy Audet

Nov 20, 2017

Contents

Pulp Smash is a test suite for Pulp. It lets you execute a workflow like this:

```
pip install pulp-smash
pulp-smash settings create  # generate a settings file
python3 -m unittest discover pulp_smash.tests  # run the tests
```

Pulp Smash is a GPL-licensed Python library, but no knowledge of Python is required to execute the tests. Just install the application, run it, and follow the prompts.

Pulp Smash has a presence on the following websites:

- Documentation is available on ReadTheDocs.
- A Python package is available on PyPi.
- Source code and the issue tracker are available on GitHub.

Documentation contents:

CHAPTER 1

Introductory Video

Location: *Pulp Smash* → *Introductory Video*

This video demonstrates how to install Pulp, demonstrates how to install, configure and run Pulp Smash, shows where to find more information, and covers some additional information. It covers much of the same information as the text-based documentation.

This video is designed to be viewed full-screen.

For more videos on Pulp and Pulp Smash, see the Pulp Vimeo group.

CHAPTER 2

Installation

Location: *Pulp Smash* → *Installation*

Installing Pulp Smash into a virtual environment[1] is recommended. To create and activate a virtual environment:

```
python3 -m venv env
source env/bin/activate  # run `deactivate` to exit environment
```

Pulp Smash can be installed from PyPI or from source. To install from PyPI:

```
pip install pulp-smash  # prepend `python -m` on Python 3.3
```

To install Pulp Smash from source (GitHub):

```
git clone https://github.com/PulpQE/pulp-smash.git
cd pulp-smash
pip install .
```

Pulp Smash can also be installed from source in "develop mode," where changes to source code are reflected in the working environment. The `--editable` flag does this. Also, development dependencies can be installed by requiring the extra "dev" group.

```
git clone https://github.com/PulpQE/pulp-smash.git
cd pulp-smash
pip install --editable .[dev]
```

For an explanation of key concepts and more installation strategies, see Installing Python Modules.

[1] See Virtual Environments and Packages for an explanation of virtual environments. `python3 -m venv` and `virtualenv` are similar, but the former ships with Python as of Python 3.3, whereas the latter is a third party tool.

CHAPTER 3

Usage

Location: *Pulp Smash* → *Usage*

3.1 Configuring Pulp Smash

Pulp Smash needs a configuration file to be present in order to know how to access the Pulp system under test. To do that you can use the following command:

```
pulp-smash settings create
```

Provide the information like the authentication credentials for a Pulp admin user, Pulp version, the system hostname, if the system is published under HTTPS or not, etc.

The `pulp-smash settings create` command will generate a settings file for a Pulp deployment. For more information about how to install Pulp check its installation docs. The generated settings file will assume that both Pulp's Admin Client and Alternate Download Policies are installed and configured.

If you are planning to test a clustered Pulp deployment, then you are going to need to manually edit the settings file. Before getting into the details of the settings file, let's see how Pulp Smash finds the settings file.

Pulp Smash abides by the XDG Base Directory Specification. The configuration file may be placed in any XDG-compliant location. The first configuration file found is used. Settings are not cascaded.

Note: The default settings filename is settings.json and it must be under a directory called pulp_smash. The pulp_smash directory may be placed in any XDG-compliant location.

A non-default configuration file can be selected with an environment variable like so: `PULP_SMASH_CONFIG_FILE=alternate-settings.json python3 -m unittest discover pulp_smash.tests`. This variable should be a file name, not a path.

Note: Setting `PULP_SMASH_CONFIG_FILE` will make Pulp Smash look for an alternate filename but the file must continue be placed under the pulp_smash directory on any XDG-compliant location.

Now that how Pulp Smash finds the settings file is known, let's know how the configuration file format is. Consider the example below:

```
{
    "pulp": {
        "auth": ["username", "password"],
        "version": "2.12.2"
    },
    "systems": [
        {
            "hostname": "pulp.example.com",
            "roles": {
                "amqp broker": {"service": "qpidd"},
                "api": {
                    "scheme": "https",
                    "verify": true
                },
                "mongod": {},
                "pulp cli": {},
                "pulp celerybeat": {},
                "pulp resource manager": {},
                "pulp workers": {},
                "shell": {"transport": "local"},
                "squid": {}
            }
        }
    ]
}
```

Pulp Smash's configuration file format allows configuring either a Pulp deployment (as seen in the example above) or a clustered Pulp deployment where each service can be installed on its own machine. The "pulp" section lets you declare the version of and authentication credentials for the Pulp deployment under test. The "systems" section provides information about each system that composes the deployment. A system must have its hostname and the roles that system has.

Note: The only required role for a system is the `shell` role. Pulp Smash will not be able to test a Pulp deployment if any of the required roles are missing, see *`pulp_smash.config.REQUIRED_ROLES`* for the list of required roles.

Not all roles requires additional information. Currently, only the `amqp broker`, `api` and `shell` roles do. The `amqp broker` expects the service to be defined and it can be either `qpidd` or `rabbitmq`. The `api` role means that `httpd` will be running on the system. The api's `scheme` allows specifying if the API should be accessed using HTTP or HTTPS, `verify` allows specifying if the request SSL certificate should be verified (true or false or a path to a custom certificate file, the path must be local to the system where Pulp Smash is being run). The `shell` role configures how the system will be accessed by using a `local` or `ssh` transport, only set `local` if Pulp Smash is running on that same system.

Note: Pulp Smash can access a system via SSH only if the SSH connection can be made without typing a password. Make sure to configure SSH so just running `ssh $hostname` will access the system. See sshd_config(5).

The example below shows a configuration file that enables Pulp Smash to access a clustered Pulp deployment:

```
{
    "pulp": {
        "auth": ["username", "password"],
        "version": "2.12.1"
    },
    "systems": [
        {
            "hostname": "first.example.com",
            "roles": {
                "amqp broker": {"service": "qpidd"},
                "api": {"scheme": "https", "verify": true},
                "mongod": {},
                "pulp cli": {},
                "pulp celerybeat": {},
                "pulp resource manager": {},
                "pulp workers": {},
                "shell": {"transport": "ssh"},
                "squid": {}
            }
        },
        {
            "hostname": "second.example.com",
            "roles": {
                "api": {"scheme": "https", "verify": false},
                "pulp celerybeat": {},
                "pulp resource manager": {},
                "pulp workers": {},
                "shell": {"transport": "ssh"},
                "squid": {}
            }
        }
    ]
}
```

Note that the roles `mongod` and `amqp broker` is only available on the first system and that the Pulp related roles plus the `squid` are available on both. The example shows how to have a clustered deployment where second system will connect to the first system's `mongod` and `amqp broker`, all the other services will work as a failover redundancy. Like, if first system's `pulp resource manager` goes down than Pulp failover feature will activate and start using the second system's `pulp resource manager`.

Pulp Smash also has two other commands to help with configuration file management: `pulp-smash settings show` and `pulp-smash settings validate` to show the current settings file and validate the settings file format schema respectively. Those commands will take into consideration the environment variables to select an alternate settings file.

3.2 Running the tests

All tests can be run by running the command below:

```
python3 -m unittest discover pulp_smash.tests
```

Any subset of tests may also be selected. For example, you may also run `python3 -m unittest pulp_smash.tests.pulp2.platform.api_v2.test_login`. Consult the unittest documentation for test selection syntax, and consult the *API Documentation* to see which test modules are available, check the tests under the `pulp_smash.tests.*` namespace.

CHAPTER 4

About

Location: *Pulp Smash* → *About*

Why does Pulp Smash exist? What are its goals, and what does it *not* do?

4.1 Why Pulp Smash?

Pulp Smash exists to make testing Pulp easy.

4.2 Scope and Limitations

4.2.1 Portability

Pulp Smash should be usable in any environment that supports:

- one of the Python versions listed in `.travis.yml`,
- the dependencies listed in `setup.py`,
- a *nix-like shell,
- the XDG Base Directory Specification,
- and OpenSSH or a compatible clone.

In addition, we recommend that GNU Make or a compatible clone be available.

This level of portability[1] allows Pulp Smash to be accessible[2].

4.2.2 Provisioning

Pulp Smash is not concerned with provisioning systems. Users must bring their own systems.

4.2.3 Destructiveness

Pulp Smash is highly destructive! You should not use Pulp Smash for testing if you care about the state of the target system. Pulp Smash will do the following to a system under test, and possibly more:

- It will drop databases.
- It will forcefully delete files from the filesystem.
- It will stop and start system services.

Pulp Smash treats the system(s) under test as cattle, not pets.[3]

4.3 Contributing

Contributions are encouraged. The easiest way to contribute is to submit a pull request on GitHub, but patches are welcome no matter how they arrive.

4.3.1 Learning Pulp Smash

Not sure where to start? Consider reading an existing test module, creating a development environment, and tackling an open issue.

The *Introductory Module* is a great candidate for study.

[1] Portable software cannot make assumptions about its environment. It cannot reference `/etc/pki/tls/certs/ca-bundle.crt` or call `yum`. Instead, it must use standardized mechanisms for interacting with its environment. This separation of concerns should lead to an application with fewer responsibilities. Fewer responsibilities means fewer bugs and more focused developers.

[2] An inaccessible project is a dead project. Labeling a project "open source" and licensing it under a suitable terms does not change that fact. People have better things to do than bang their head against a wall.

[3] The "pets vs cattle" analogy is widely attributed to Bill Baker of Microsoft.

Installation provides a recipe for creating a virtualenv-based development environment. To verify the sanity of your development environment, `cd` into the Pulp Smash source code directory and execute `make all`.

The Pulp Smash issues list includes test cases that should be automated and added to the test suite.

4.3.2 Code Standards

Please adhere to the following guidelines:

- Pull requests must pass the Travis CI continuous integration tests. These tests are automatically run whenever a pull request is submitted. If you want to locally verify your changes before submitting a pull request, execute `make all`.
- Test failures must not be introduced. Consider running all new and modified tests and copy-pasting the output from the test run as a comment in the GitHub pull request. The simplest way to run the test suite is with `python3 -m unittest pulp_smash.tests`. See the unittest Command-Line Interface and `python3 -m pulp_smash` for more information.
- Each commit in a pull request must be atomic and address a single issue. Try asking yourself: "can I revert this commit?" Knowing how to rewrite history may help. In addition, please take the time to write a good commit message. While not *strictly* necessary, consider: commits are (nearly) immutable, and getting commit messages right makes for a more pleasant review process, better release notes, and easier use of tools like `git log`, `git blame` or `git bisect`.
- The pull request must not raise any other outstanding concerns. For example, do not author a commit that adds a 10MB binary blob without exceedingly good reason. As another example, do not add a test that makes dozens of concurrent requests to a public service such as docker hub.

4.3.3 Review Process

Changes that meet the *code standards* will be reviewed by a Pulp Smash developer and are likely to be merged.

Though commits are accepted as-is, they are frequently accompanied by a follow-up commit in which the reviewer makes a variety of changes, ranging from simple typo corrections and formatting adjustments to whole-sale restructuring of tests. This can take quite a bit of effort and time. If you'd like to make the review process faster and have more assurance your changes are being accepted with little to no modifications, take the time to really make your changes shine: ensure your code is DRY, matches existing formatting conventions, is organized into easy-to-read blocks, has isolated unit test assertions, and so on.

Join the #pulp IRC channel on freenode if you have further questions.

4.3.4 Labels

Issues are categorized with labels. Pull requests are categorized with GitHub's pull request reviews feature.

The specific meaning of (issue) labels is as follows.

Issue Type: Bug This label denotes an issue that describes a specific counter-productive behaviour. For example, an issue entitled "test X contains an incorrect assertion" is a great candidate for this label.

Issue Type: Discussion This label denotes an issue that broadly discusses some topic. Feature requests should be given this label. If a discussion results in a specific and concrete plan of action, a new issue should be opened, where that issue outlines a specific solution and has a label of "Issue Type: Plan".

Issue Type: Plan This label denotes an issue that outlines a specific, concrete plan of action for improving Pulp Smash. This may include plans for new utilities or refactors of existing tests or other tools. Open-ended discussions (including feature requests) should go into issues labeled "Issue Type:Discussion."

Issue Type: Test Case This label indicates that an issue is asking for a test case to be automated. (Issues with this label are a special type of plan.)

Pulp Version: 3 This label serves to differentiate issues, that otherwise should belong to one of the above issue types, as being related to creation of automated tests or utilities for for Pulp 3. This label is meant to aid pulp-smash developers in filtering issues by major version, as Pulp 3 introduces many breaking changes.

CHAPTER 5

Introductory Module

Location: *Pulp Smash* → *Introductory Module*

`pulp_smash.tests.pulp2.platform.api_v2.test_login` is one of the shortest, simplest and most heavily commented modules in Pulp Smash. Reading through this module will give you a mental framework for understanding other modules and making your own changes. It is included here for convenience.

```
# coding=utf-8
"""Test the API's `authentication`_ functionality.

.. _authentication:
    https://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/
↪authentication.html
"""
# Why The Comments?
# =================
#
# This module serves as an introduction to Pulp Smash. This module was picked
# because it's one of the simplest in the test suite. A "how to understand Pulp
# Smash tests" section in the documentation should link to this module.
#
# Encoding
# --------
#
# The encoding declaration at the beginning of this file tells Python which
# encoding scheme was used when creating this document, and therefore how to
# decode the bytes in this document. If omitted, Python might use an UTF-8
# decoder, or a utf-8 decoder, or something else, and that can be problematic.
# Try running this script with Python in a variety of environments:
#
#     #!/usr/bin/env python3
#     # ...
#     import sys
#     print(sys.getdefaultencoding())
#
# Shebang
```

```
# -------
#
# There is no shebang at the top of this file. That's because this file is not
# meant to be executed directly. Instead, a "test runner" program will import
# this module, and it will decide if and how to execute the code herein.
# Several test runners can run Pulp Smash tests, but with varying degrees of
# compatibility. The "unittest" test runner is best supported. [1]
#
# Docstrings
# ----------
#
# The docstrings throughout this module comply with reStructuredText syntax.
# [2] However, a tool like `rst2html` cannot extract and compile the docstrings
# from this file. Instead, the Sphinx documentation generator must be used. [3]
# It knows how to extract docstrings, and it knows how to treat directives like
# the `:meth:` directive. See the README file in the root of this repository
# for more information.
#
# Package Structure
# -----------------
#
# Why is this module placed where it is?
#
# First, a path such as `pulp_smash.tests.pulp2.platform.api_v2.test_login` is
# human-readable. This module of Pulp Smash tests relies on the base platform's
# API version 2 interface, and it tests logging in.
#
# Second, Pulp has a plug-in architecture, where the base platform is always
# present, and everything else is optional. Pulp Smash's packages are
# structured to reflect that plug-in architecture. This makes it easy for a
# test case to know which plug-ins it may reference. For example, test cases in
# `pulp_smash.tests.pulp2.rpm` may reference the RPM plug-in, but must not
# reference the Puppet plugin. This also makes it easier to automatically skip
# tests. For example, if the RPM plug-in is not installed on the Pulp system
# under test, all test cases in `pulp_smash.tests.pulp2.rpm` are skipped.
#
# No per-plugin test skipping logic is present in this module. Elsewhere, be on
# the look-out for `setUpModule` functions and mentions of the `load_tests`
# protocol. [4]
#
# Imports
# -------
#
# The imports are listed in the order recommended by PEP 8: double-underscore,
# standard library, third party and local. [5][6] In addition, blocks of
# imports are sorted alphabetically, and "import x" statements are placed
# before "from x import y" statements.
#
# [1] https://docs.python.org/3.5/library/unittest.html
# [2] http://docutils.sourceforge.net/docs/user/rst/quickstart.html
# [3] http://www.sphinx-doc.org/en/stable/
# [4] https://docs.python.org/3/library/unittest.html#load-tests-protocol
# [5] https://www.python.org/dev/peps/pep-0008/#imports
# [6] https://docs.python.org/3/library/__future__.html
import unittest

from pulp_smash import api, config, selectors
from pulp_smash.tests.pulp2.constants import (
```

```
    ERROR_KEYS,
    LOGIN_KEYS,
    LOGIN_PATH,
)
from pulp_smash.tests.pulp2.platform.utils import set_up_module as setUpModule  #␣
↪noqa pylint:disable=unused-import


class LoginTestCase(unittest.TestCase):
    """Tests for logging in."""

    # The `TestCase` class provides a lot of functionality, and it's a Good
    # Idea to read the unittest documentation. [1] Right now, you just need to
    # know that, when a test runner executes a `TestCase` class:
    #
    # * The `test*` methods run in alphabetic order.
    # * A failure in one `test*` method doesn't affect any other `test*`
    #   method.
    #
    # [1] https://docs.python.org/3/library/unittest.html

    def test_success(self):
        """Successfully log in to the server.

        Assert that:

        * The response has an HTTP 200 status code.
        * The response body is valid JSON and has correct keys.
        """
        # The object returned by `config.get_config()` tells the new `Client`
        # object where the Pulp server is, what the authentication credentials
        # are, and so on.
        #
        # There are several assertions that can be made about the login API
        # call. Rather than logging in logging in once for every test, we log
        # in just once, and make multiple assertions about that log in.
        response = api.Client(config.get_config()).post(LOGIN_PATH)
        with self.subTest(comment='check response status code'):
            self.assertEqual(response.status_code, 200)
        with self.subTest(comment='check response body'):
            self.assertEqual(frozenset(response.json().keys()), LOGIN_KEYS)

    def test_failure(self):
        """Unsuccessfully log in to the server.

        Assert that:

        * The response has an HTTP 401 status code.
        * The response body is valid JSON and has correct keys.
        """
        # By default, `Client` objects munge every response they receive. They
        # act cautiously, and do things like raise an exception if the response
        # has an HTTP 4XX or 5XX status code. We don't want that to happen in
        # this test case. So, we pass the `Client` object a non-default
        # function with which to munge responses. In addition, we override the
        # default authentication handling for just this one API call.
        #
        # The API and CLI clients are interesting and capable classes. Read
```

```
        # about them.
        cfg = config.get_config()
        response = (
            api.Client(cfg, api.echo_handler).post(LOGIN_PATH, auth=('', ''))
        )
        with self.subTest(comment='check response status code'):
            self.assertEqual(response.status_code, 401)
        # The `version` attribute should correspond to the version of the Pulp
        # server under test. This block of code says "if bug 1412 is not fixed
        # in Pulp version X, then skip this test."
        if selectors.bug_is_testable(1412, cfg.pulp_version):
            with self.subTest(comment='check response body'):
                self.assertEqual(frozenset(response.json().keys()), ERROR_KEYS)
```

CHAPTER 6

API Documentation

Location: *Pulp Smash* → *API Documentation*

This is the Pulp Smash API documentation. It is mostly auto generated from the source code. Beware that Pulp Smash is very new. Its API is **not stable**. This section of the documentation should be treated as a handy reference for developers, not a gospel.

6.1 *pulp_smash*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash* The root of Pulp Smash's namespace.

6.2 *pulp_smash.api*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.api* A client for working with Pulp's API.

Working with an API can require repetitive calls to perform actions like check HTTP status codes. In addition, Pulp's API has specific quirks surrounding its handling of href paths and HTTP 202 status codes. This module provides a customizable client that makes it easier to work with the API in a safe and concise manner.

class `pulp_smash.api.`**`Client`**(*server_config*, *response_handler=None*, *request_kwargs=None*, *pulp_system=None*)

A convenience object for working with an API.

This class is a wrapper around the `requests.api` module provided by Requests. Each of the functions from that module are exposed as methods here, and each of the arguments accepted by Requests' functions are also accepted by these methods. The difference between this class and the Requests functions lies in its configurable request and response handling mechanisms.

This class is flexible enough that it should be usable with any API, but certain defaults have been set to work well with Pulp.

As an example of basic usage, let's say that you'd like to create a user, then read that user's information back from the server. This is one way to do it:

```
>>> from pulp_smash.api import Client
>>> from pulp_smash.config import get_config
>>> client = Client(get_config())
>>> response = client.post('/pulp/api/v2/users/', {'login': 'Alice'})
>>> response = client.get(response.json()['_href'])
>>> print(response.json())
```

Notice how we never call `response.raise_for_status()`? We don't need to because, by default, `Client` instances do this. Handy!

How does this work? Each `Client` object has a callback function, `response_handler`, that is given a chance to munge each server response. How else might this callback be useful? Well, notice how we call `json()` on each server response? That's kludgy. Let's write our own callback that takes care of this for us:

```
>>> from pulp_smash.api import Client
>>> from pulp_smash.config import get_config
>>> def response_handler(server_config, response):
...     response.raise_for_status()
...     return response.json()
>>> client = Client(get_config(), response_handler=response_handler)
>>> response = client.post('/pulp/api/v2/users/', {'login': 'Alice'})
>>> response = client.get(response['_href'])
>>> print(response)
```

Pulp Smash ships with several response handlers. See:

- *pulp_smash.api.echo_handler()*
- *pulp_smash.api.safe_handler()*
- *pulp_smash.api.json_handler()*

As mentioned, this class has configurable request and response handling mechanisms. We've covered response handling mechanisms — let's move on to request handling mechanisms.

When a client is instantiated, a *pulp_smash.config.PulpSmashConfig* must be passed to the constructor, and configuration options are copied from the `PulpSmashConfig` to the client. These options can be overridden on a per-object or per-request basis. Here's an example:

```
>>> from pulp_smash.api import Client
>>> from pulp_smash.config import PulpSmashConfig
>>> cfg = config.PulpSmashConfig(
...     pulp_auth=('username', 'password'),
...     systems=[
...         config.PulpSystem(
...             hostname='example.com',
...             roles={'api': {
...                 'scheme': 'https',
...                 'verify': '~/Documents/my.crt',
...             }}
...         )
...     ]
... )
>>> client = api.Client(cfg)
>>> client.request_kwargs['url'] == 'https://example.com'
True
>>> client.request_kwargs['verify'] == '~/Documents/my.crt'
True
>>> response = client.get('/index.html')  # Use my.crt for SSL verification
>>> response = client.get('/index.html', verify=False)  # Disable SSL
```

```
>>> response = client.get('/index.html')  # Use my.crt for SSL verification
>>> client.request_kwargs['verify'] = None
>>> response = client.get('/index.html')  # Do default SSL verification
```

Anything accepted by the Requests functions may be placed in `request_kwargs` or passed in to a specific call. You can set `verify` for example.

The `url` argument is slightly special. When making a call, it is possible to pass in a relative URL, as shown in the examples above. (e.g. `/index.html`.) The default URL and passed-in URL are joined like so:

```
>>> urljoin(request_kwargs['url'], passed_in_url)
```

This allows one to easily use the hrefs returned by Pulp in constructing new requests.

The remainder of this docstring contains design notes. They are useful to advanced users and developers.

Requests' `requests.api.post` method has the following signature:

```
def post(url, data=None, json=None, **kwargs)
```

Pulp supports only JSON for most of its API endpoints, so it makes sense for us to demote `data` to being a regular kwarg and list `json` as the one and only positional argument.

We make `json` a positional argument for `post()`, `put()` and `patch()`, but not the other methods. Why? Because HTTP OPTIONS, GET, HEAD and DELETE **must not** have bodies. This is stated by the HTTP/1.1 specification, and network intermediaries such as caches are at liberty to drop such bodies.

Why the sentinel? Imagine the following scenario: a user provides a default JSON payload in `self.request_kwargs`, but they want to skip sending that payload for just one request. How can they do that? With `client.post(url, None)`.

delete(*url*, ***kwargs*)
Send an HTTP DELETE request.

get(*url*, ***kwargs*)
Send an HTTP GET request.

head(*url*, ***kwargs*)
Send an HTTP HEAD request.

options(*url*, ***kwargs*)
Send an HTTP OPTIONS request.

patch(*url*, *json=<object object>*, ***kwargs*)
Send an HTTP PATCH request.

post(*url*, *json=<object object>*, ***kwargs*)
Send an HTTP POST request.

put(*url*, *json=<object object>*, ***kwargs*)
Send an HTTP PUT request.

request(*method*, *url*, ***kwargs*)
Send an HTTP request.

Arguments passed directly in to this method override (but do not overwrite!) arguments specified in `self.request_kwargs`.

`pulp_smash.api.`**code_handler**(*server_config*, *response*)
Check the response status code, and return the response.

Unlike *safe_handler()*, this method doesn't wait for asynchronous tasks to complete if `response` has an HTTP 202 status code.

Raises `requests.exceptions.HTTPError` if the response status code is in the 4XX or 5XX range.

`pulp_smash.api.`**`echo_handler`**(*server_config*, *response*)
Immediately return `response`.

`pulp_smash.api.`**`json_handler`**(*server_config*, *response*)
Like `safe_handler`, but also return a JSON-decoded response body.

Do what *pulp_smash.api.safe_handler()* does. In addition, decode the response body as JSON and return the result.

`pulp_smash.api.`**`poll_spawned_tasks`**(*server_config*, *call_report*, *pulp_system=None*)
Recursively wait for spawned tasks to complete. Yield response bodies.

Recursively wait for each of the spawned tasks listed in the given call report to complete. For each task that completes, yield a response body representing that task's final state.

Parameters

- **`server_config`** – A *pulp_smash.config.PulpSmashConfig* object.
- **`call_report`** – A dict-like object with a call report structure.
- **`pulp_system`** – The system from where to pool the task. If `None` is provided then the first system found with api role will be used.

Returns A generator yielding task bodies.

Raises Same as *poll_task()*.

`pulp_smash.api.`**`poll_task`**(*server_config*, *href*, *pulp_system=None*)
Wait for a task and its children to complete. Yield response bodies.

Poll the task at `href`, waiting for the task to complete. When a response is received indicating that the task is complete, yield that response body and recursively poll each child task.

Parameters

- **`server_config`** – A *pulp_smash.config.PulpSmashConfig* object.
- **`href`** – The path to a task you'd like to monitor recursively.
- **`pulp_system`** – The system from where to pool the task. If `None` is provided then the first system found with api role will be used.

Returns An generator yielding response bodies.

Raises ***pulp_smash.exceptions.TaskTimedOutError*** – If a task takes too long to complete.

`pulp_smash.api.`**`safe_handler`**(*server_config*, *response*)
Check status code, wait for tasks to complete, and check tasks.

Inspect the response's HTTP status code. If the response has an HTTP Accepted status code, inspect the returned call report, wait for each task to complete, and inspect each completed task.

Raises `requests.exceptions.HTTPError` if the response status code is in the 4XX or 5XX range.

Raises

- ***pulp_smash.exceptions.CallReportError*** – If the call report contains an error.

- ***pulp_smash.exceptions.TaskReportError*** – If the task report contains an error.

6.3 *pulp_smash.cli*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.cli* A client for working with Pulp hosts via their CLI.

class pulp_smash.cli.**BaseServiceManager**
A base service manager.

Each subclass must implement the abstract methods to provide the service management on a single or multiple hosts.

Subclasses should take advantage of the helper methods offered by this class in order to manage services and check the proper service manager software available on a host.

This base class also offers a context manager to temporary disable SELinux. It is useful when managing services on hosts running RHEL 6 and earlier, which has SELinux issues when running on Jenkins.

Make sure to call this class __init__ method on the subclass __init__ method to ensure the helper methods functionality.

restart(*services*)
Restart the given services.

Parameters **services** – A list or tuple of services to be restarted.

start(*services*)
Start the given services.

Parameters **services** – A list or tuple of services to be started.

stop(*services*)
Stop the given services.

Parameters **services** – A list or tuple of services to be stopped.

class pulp_smash.cli.**Client**(*server_config*, *response_handler=None*, *pulp_system=None*)
A convenience object for working with a CLI.

This class provides the ability to execute shell commands on either the local host or a remote host. Here is a pedagogic usage example:

```
>>> from pulp_smash import cli, config
>>> system = (
...     config.PulpSystem('localhost', {'shell': {'transport': 'local'}})
... )
>>> cfg = config.PulpSmashConfig(systems=[system])
>>> client = cli.Client(cfg, pulp_system=system)
>>> response = client.run(('echo', '-n', 'foo'))
>>> response.returncode == 0
True
>>> response.stdout == 'foo'
True
>>> response.stderr == ''
True
```

The above example shows how various classes fit together. It's also verbose: smartly chosen defaults mean that most real code is much more concise.

You can customize how `Client` objects execute commands and handle responses by fiddling with the two public instance attributes:

machine A Plumbum machine. *`run()`* delegates all command execution responsibilities to this object.

response_handler A callback function. Each time `machine` executes a command, the result is handed to this callback, and the callback's return value is handed to the user.

If `pulp_system.roles['shell']['transport']` is `'local'` or `'ssh'`, `machine` will be set so that commands run locally or over SSH, respectively. If `pulp_system.roles['shell']['transport']` is `None`, the constructor will guess how to set `machine` by comparing the hostname embedded in `pulp_system.hostname` against the current host's hostname. If they match, `machine` is set to execute commands locally; and vice versa.

Parameters

- **server_config** (`pulp_smash.config.PulpSmashConfig`) – Information about the host on which commands will be executed.
- **response_handler** – A callback function. Defaults to *`pulp_smash.cli.code_handler()`*.
- **pulp_system** (`pulp_smash.config.PulpSystem`) – A specific host to target, instead of the first host with the `pulp cli` role.

run(*args*, ***kwargs*)

Run a command and `return self.response_handler(result)`.

This method is a thin wrapper around Plumbum's BaseCommand.run method, which is itself a thin wrapper around the standard library's subprocess.Popen class. See their documentation for detailed usage instructions. See *`pulp_smash.cli.Client`* for a usage example.

class `pulp_smash.cli.`**CompletedProcess**(*args*, *returncode*, *stdout*, *stderr*)

A process that has finished running.

This class is similar to the `subprocess.CompletedProcess` class available in Python 3.5 and above. Significant differences include the following:

- All constructor arguments are required.
- *`check_returncode()`* returns a custom exception, not `subprocess.CalledProcessError`.

All constructor arguments are stored as instance attributes.

Parameters

- **args** – A string or a sequence. The arguments passed to *`pulp_smash.cli.Client.run()`*.
- **returncode** – The integer exit code of the executed process. Negative for signals.
- **stdout** – The standard output of the executed process.
- **stderr** – The standard error of the executed process.

check_returncode()

Raise an exception if `returncode` is non-zero.

Raise *`pulp_smash.exceptions.CalledProcessError`* if `returncode` is non-zero.

Why not raise `subprocess.CalledProcessError`? Because stdout and stderr are not included when str() is called on a CalledProcessError object. A typical message is:

```
"Command '('ls', 'foo')' returned non-zero exit status 2"
```

This information is valuable. One could still make `subprocess.CalledProcessError` work by overloading `args`:

```
>>> if isinstance(args, (str, bytes)):
...     custom_args = (args, stdout, stderr)
... else:
...     custom_args = tuple(args) + (stdout, stderr)
>>> subprocess.CalledProcessError(args, returncode)
```

But this seems like a hack.

In addition, it's generally good for an application to raise expected exceptions from its own namespace, so as to better abstract away dependencies.

class `pulp_smash.cli.`**`GlobalServiceManager`**(*cfg*)

A service manager that manages services on all Pulp hosts.

Each instance of this class manages a single service. When a method like *`start()`* is executed, it will start a service on all hosts that are declared as running that service. For example, imagine that the following is executed:

```
>>> from pulp_smash import cli, config
>>> cfg = config.get_config()
>>> svc_mgr = cli.GlobalServiceManager(cfg)
>>> svc_mgr.start(['httpd'])
```

In this case, the service manager will iterate over all hosts in `cfg`. For each host that is declared as fulfilling the `api` role, Apache (httpd) will be restarted.

When asked to perform an action, this object may talk to each target host and determines whether it is running as root. If not root, all commands are prefixed with "sudo". Please ensure that Pulp Smash can either execute commands as root or can successfully execute `sudo`. You may need to edit your `~/.ssh/config` file.

For conceptual information on why both a *`pulp_smash.cli.ServiceManager`* and a *`pulp_smash.cli.GlobalServiceManager`* are necessary, see *`pulp_smash.config.PulpSmashConfig`*.

Parameters **`cfg`** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp deployment.

Raises ***`pulp_smash.exceptions.NoKnownServiceManagerError`*** – If unable to find any service manager on one of the target hosts.

`restart`(*services*)

Restart the services on every host that has the services.

Parameters **`services`** – An iterable of service names.

Returns A dict mapping the affected hosts' hostnames with a list of *`pulp_smash.cli.CompletedProcess`* objects.

`start`(*services*)

Start the services on every host that has the services.

Parameters **`services`** – An iterable of service names.

Returns A dict mapping the affected hosts' hostnames with a list of *`pulp_smash.cli.CompletedProcess`* objects.

`stop`(*services*)

Stop the services on every host that has the services.

Parameters **`services`** – An iterable of service names.

Returns A dict mapping the affected hosts' hostnames with a list of *pulp_smash.cli.CompletedProcess* objects.

class pulp_smash.cli.**PackageManager**(*cfg*)

A package manager on a host.

Each instance of this class represents the package manager on a host. An example may help to clarify this idea:

```
>>> from pulp_smash import cli, config
>>> pkg_mgr = cli.PackageManager(config.get_config())
>>> completed_process = pkg_mgr.install('vim')
>>> completed_process = pkg_mgr.uninstall('vim')
```

In the example above, the pkg_mgr object represents the package manager on the host referenced by *pulp_smash.config.get_config()*.

Upon instantiation, a *PackageManager* object talks to its target host and uses simple heuristics to determine which package manager is used. As a result, it's possible to manage packages on heterogeneous host with homogeneous commands.

Upon instantiation, this object talks to the target host and determines whether it is running as root. If not root, all commands are prefixed with "sudo". Please ensure that Pulp Smash can either execute commands as root or can successfully execute sudo. You may need to edit your ~/.ssh/config file.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the target host.

Raises ***pulp_smash.exceptions.NoKnownPackageManagerError*** – If unable to find any package manager on the target host.

install(**args*)

Install the named packages.

Return type *pulp_smash.cli.CompletedProcess*

uninstall(**args*)

Uninstall the named packages.

Return type *pulp_smash.cli.CompletedProcess*

upgrade(**args*)

Upgrade the named packages.

Return type *pulp_smash.cli.CompletedProcess*

class pulp_smash.cli.**ServiceManager**(*cfg*, *pulp_system*)

A service manager on a host.

Each instance of this class represents the service manager on a host. An example may help to clarify this idea:

```
>>> from pulp_smash import cli, config
>>> cfg = config.get_config()
>>> pulp_system = cfg.get_services_for_roles(('api',))[0]
>>> svc_mgr = cli.ServiceManager(cfg, pulp_system)
>>> completed_process_list = svc_mgr.stop(['httpd'])
>>> completed_process_list = svc_mgr.start(['httpd'])
```

In the example above, svc_mgr represents the service manager (such as SysV or systemd) on a host. Upon instantiation, a *ServiceManager* object talks to its target host and uses simple heuristics to determine which service manager is available. As a result, it's possible to manage services on heterogeneous hosts with homogeneous commands.

Upon instantiation, this object talks to the target host and determines whether it is running as root. If not root, all commands are prefixed with "sudo". Please ensure that Pulp Smash can either execute commands as root or can successfully execute `sudo`. You may need to edit your `~/.ssh/config` file.

For conceptual information on why both a *pulp_smash.cli.ServiceManager* and a *pulp_smash.cli.GlobalServiceManager* are necessary, see *pulp_smash.config.PulpSmashConfig*.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp application.
- **pulp_system** (pulp_smash.config.PulpSystem) – The host to target.

Raises ***pulp_smash.exceptions.NoKnownServiceManagerError*** – If unable to find any service manager on the target host.

restart (*services*)
Restart the given services.

Parameters **services** – An iterable of service names.

Returns An iterable of *pulp_smash.cli.CompletedProcess* objects.

start (*services*)
Start the given services.

Parameters **services** – An iterable of service names.

Returns An iterable of *pulp_smash.cli.CompletedProcess* objects.

stop (*services*)
Stop the given services.

Parameters **services** – An iterable of service names.

Returns An iterable of *pulp_smash.cli.CompletedProcess* objects.

pulp_smash.cli.**code_handler** (*completed_proc*)
Check the process for a non-zero return code. Return the process.

Check the return code by calling `completed_proc.check_returncode()`. See: *pulp_smash.cli.CompletedProcess.check_returncode()*.

pulp_smash.cli.**echo_handler** (*completed_proc*)
Immediately return `completed_proc`.

6.4 pulp_smash.config

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.config* Tools for managing information about hosts under test.

Pulp Smash needs to know about the Pulp application under test and the hosts that comprise that application. For example, it might need to know which username and password to use when communicating with a Pulp application, or it might need to know which host is hosting the squid service, if any. This module eases the task of managing that information.

pulp_smash.config.**AMQP_SERVICES = {'qpidd', 'rabbitmq'}**
The set of services that can fulfill the `amqp broker` role.

pulp_smash.config.**CONFIG_JSON_SCHEMA = {'additionalProperties': False, 'type': 'object', 'properties': {'pulp': {'add**
The schema for Pulp Smash's configuration file.

pulp_smash.config.**OPTIONAL_ROLES = {'squid', 'pulp cli'}**
Additional roles that can be present in a Pulp application.

class pulp_smash.config.**PulpSmashConfig**(*pulp_auth=None*, *pulp_version=None*, *systems=None*)
Information about a Pulp application.

This object stores information about Pulp application and its constituent hosts. A single Pulp application may have its services spread across several hosts. For example, one host might run Qpid, another might run MongoDB, and so on. Here's how to model a multi-host deployment where Apache runs on one host, and the remaining components run on another host:

```
>>> import requests
>>> from pulp_smash.config import PulpSmashConfig
>>> cfg = PulpSmashConfig(
...     pulp_auth=('username', 'password'),
...     pulp_version='2.12.2',
...     systems=[  # A misnomer. Think "hosts," not "systems."
...         PulpSystem(
...             hostname='pulp1.example.com',
...             roles={'api': {'scheme': 'https'}},
...         ),
...         PulpSystem(
...             hostname='pulp.example.com',
...             roles={
...                 'amqp broker': {'service': 'qpidd'},
...                 'mongod': {},
...                 'pulp celerybeat': {},
...                 'pulp resource manager': {},
...                 'pulp workers': {},
...                 'shell': {'transport': 'ssh'},
...             },
...         )
...     ]
... )
```

In the simplest case, all of the services that comprise a Pulp applicaiton run on a single host. Here's an example of how this object might model a single-host deployment:

```
>>> import requests
>>> from pulp_smash.config import PulpSmashConfig
>>> cfg = PulpSmashConfig(
...     pulp_auth=('username', 'password'),
...     pulp_version='2.12.2',
...     systems=[  # A misnomer. Think "hosts," not "systems."
...         PulpSystem(
...             hostname='pulp.example.com',
...             roles={
...                 'amqp broker': {'service': 'qpidd'},
...                 'api': {'scheme': 'https'},
...                 'mongod': {},
...                 'pulp cli': {},
...                 'pulp celerybeat': {},
...                 'pulp resource manager': {},
...                 'pulp workers': {},
...                 'shell': {'transport': 'ssh'},
...             },
...         )
...     ]
```

```
... )
```

In the simplest case, Pulp Smash's configuration file resides at `~/.config/pulp_smash/settings.json`. However, there are several ways to alter this path. Pulp Smash obeys the XDG Base Directory Specification. In addition, Pulp Smash responds to the `PULP_SMASH_CONFIG_FILE` environment variable. This variable is a relative path, and it defaults to `settings.json`.

Configuration files contain JSON data structured in a way that resembles what is accepted by this class's constructor. For exact details on the structure of configuration files, see *pulp_smash.config.CONFIG_JSON_SCHEMA*.

Parameters

- **pulp_auth** – A two-tuple. Credentials to use when communicating with the server. For example: `('username', 'password')`.
- **pulp_version** – A string, such as '1.2' or '0.8.rc3'. Defaults to '1!0' (epoch 1, version 0). Must be compatible with the packaging library's `packaging.version.Version` class.
- **systems** (pulp_smash.config.PulpSystem) – A list of the hosts comprising a Pulp application.

default_config_file_path
Build the default config file path.

get_base_url(*pulp_system=None*)
Generate the base URL for a given `pulp_sytem`.

Parameters **pulp_system** (pulp_smash.config.PulpSystem) – One of the hosts that comprises a Pulp application. Defaults to the first host with the `api` role.

get_config_file_path()
Get the config file path.

Raises ***pulp_smash.exceptions.ConfigFileNotFoundError*** – If configuration file cannot be found.

get_requests_kwargs(*pulp_system=None*)
Get kwargs for use by the Requests functions.

This method returns a dict of attributes that can be unpacked and used as kwargs via the `**` operator. For example:

```
>>> cfg = PulpSmashConfig().read()
>>> requests.get(cfg.get_base_url() + '...', **cfg.get_requests_kwargs())
```

This method is useful because client code may not know which attributes should be passed from a `PulpSmashConfig` object to Requests. Consider that the example above could also be written like this:

```
>>> cfg = PulpSmashConfig().get()
>>> requests.get(
...     cfg.get_base_url() + '...',
...     auth=tuple(cfg.pulp_auth),
...     verify=cfg.get_systems('api')[0].roles['api']['verify'],
... )
```

But this latter approach is more fragile. The user must remember to get a host with api role to check for the verify config, then convert `pulp_auth` config to a tuple, and it will require maintenance if `cfg` gains or loses attributes.

get_systems(*role*)
Return a list of hosts fulfilling the given role.

Parameters **role** – The role to filter the available hosts, see *pulp_smash.config.ROLES* for more information.

read(*xdg_config_file=None*, *xdg_config_dir=None*)
Read a configuration file.

Parameters

- **xdg_config_file** – A string. The name of the file to manipulate.
- **xdg_config_dir** – A string. The XDG configuration directory in which the configuration file resides.

Returns A new *pulp_smash.config.PulpSmashConfig* object. The current object is not modified by this method.

Return type *PulpSmashConfig*

Raises warnings.DecprecationWarning if the configuration file uses the old format instead of the new role-based format.

static services_for_roles(*roles*)
Return the services based on the roles.

class pulp_smash.config.**PulpSystem**(*hostname*, *roles*)

hostname
Alias for field number 0

roles
Alias for field number 1

pulp_smash.config.**REQUIRED_ROLES = {'amqp broker', 'api', 'mongod', 'pulp workers', 'shell', 'pulp resource manage**
The set of roles that must be present in a functional Pulp application.

pulp_smash.config.**ROLES = {'pulp workers', 'shell', 'pulp resource manager', 'squid', 'pulp celerybeat', 'amqp broker',**
The set of all roles that may be present in a Pulp application.

pulp_smash.config.**convert_old_config**(*config_dict*)
Convert the old configuration dict representation to the new format.

pulp_smash.config.**get_config**()
Return a copy of the global PulpSmashConfig object.

This method makes use of a cache. If the cache is empty, the configuration file is parsed and the cache is populated. Otherwise, a copy of the cached configuration object is returned.

Returns A copy of the global server configuration object.

Return type *pulp_smash.config.PulpSmashConfig*

pulp_smash.config.**validate_config**(*config_dict*)
Validate an in-memory configuration file.

Given an in-memory configuration file, verify its sanity by validating it against a schema and performing several semantic checks.

Parameters **config_dict** – A dictionary returned by json.load or json.loads after reading the config file.

Raises *pulp_smash.exceptions.ConfigValidationError* – If the any validation error is found.

6.5 pulp_smash.constants

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.constants* Values usable by multiple test modules.

pulp_smash.constants.**DOCKER_IMAGE_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/docker/busybox:latest.**
The URL to a Docker image as created by `docker save`.

pulp_smash.constants.**DOCKER_UPSTREAM_NAME = 'dmage/manifest-list-test'**
The name of a Docker repository.

This repository has several desireable properties:

- It is available via both *DOCKER_V1_FEED_URL* and *DOCKER_V2_FEED_URL*.
- It has a manifest list, where one entry has an architecture of amd64 and an os of linux. (The "latest" tag offers this.)
- It is relatively small.

This repository also has several shortcomings:

- This repository isn't an official repository. It's less trustworthy, and may be more likely to change with little or no notice.
- It doesn't have a manifest list where no list entries have an architecture of amd64 and an os of linux. (The "arm32v7" tag provides schema v1 content.)

One can get a high-level view of the content in this repository by executing:

```
curl --location --silent \
https://registry.hub.docker.com/v2/repositories/$this_constant/tags \
| python -m json.tool
```

pulp_smash.constants.**DOCKER_UPSTREAM_NAME_NOLIST = 'library/busybox'**
The name of a Docker repository without a manifest list.

DOCKER_UPSTREAM_NAME should be used when possible. However, this constant is useful for backward compatibility. If Pulp is asked to sync a repository, and:

- Pulp older than 2.14 is under test.
- The repository is configured to sync schema v2 content.
- The upstream repository has a manifest list.

. . . then Pulp will break when syncing. See Pulp #2384.

pulp_smash.constants.**DOCKER_V1_FEED_URL = 'https://index.docker.io'**
The URL to a V1 Docker registry.

This URL can be used as the "feed" property of a Pulp Docker registry.

pulp_smash.constants.**DOCKER_V2_FEED_URL = 'https://registry-1.docker.io'**
The URL to a V2 Docker registry.

This URL can be used as the "feed" property of a Pulp Docker registry.

pulp_smash.constants.**DRPM = 'drpms/test-alpha-1.1-1_1.1-2.noarch.drpm'**
The path to a DRPM file in one of the DRPM repositories.

This path may be joined with *DRPM_SIGNED_FEED_URL* or *DRPM_UNSIGNED_FEED_URL*.

pulp_smash.constants.**DRPM_SIGNED_FEED_COUNT = 4**
The number of packages available at *DRPM_SIGNED_FEED_URL*.

pulp_smash.constants.**DRPM_SIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/drpm-signed/'**
The URL to a signed DRPM repository.

pulp_smash.constants.**DRPM_SIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/drpm-signed/drpms/test**
The URL to a DRPM file.

Built from *DRPM_SIGNED_FEED_URL* and *DRPM*.

pulp_smash.constants.**DRPM_UNSIGNED_FEED_COUNT = 4**
The number of packages available at *DRPM_UNSIGNED_FEED_URL*.

pulp_smash.constants.**DRPM_UNSIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/drpm-unsigne**
The URL to an unsigned DRPM repository.

pulp_smash.constants.**DRPM_UNSIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/drpm-unsigned/drpm**
The URL to a unsigned DRPM file.

Built from *DRPM_UNSIGNED_FEED_URL* and *DRPM*.

pulp_smash.constants.**FILE_FEED_COUNT = 3**
The number of packages available at *FILE_FEED_URL*.

pulp_smash.constants.**FILE_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/file/'**
The URL to a file repository.

pulp_smash.constants.**FILE_MIXED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/file-mixed/'**
The URL to a file repository containing invalid and valid entries.

pulp_smash.constants.**FILE_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/file/1.iso'**
The URL to an ISO file at *FILE_FEED_URL*.

pulp_smash.constants.**OSTREE_BRANCHES = ['rawhide', 'stable']**
A branch in *OSTREE_FEED*. See OSTree Importer Configuration.

pulp_smash.constants.**OSTREE_FEED = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/ostree/small/'**
The URL to a URL of OSTree branches. See OSTree Importer Configuration.

pulp_smash.constants.**PULP_FIXTURES_BASE_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/'**
A URL at which generated pulp fixtures are hosted.

pulp_smash.constants.**PULP_FIXTURES_KEY_ID = '269d9d98'**
The 32-bit ID of the public key used to sign various fixture files.

To calculate a new key ID, find the public key used by Pulp Fixtures (it should be in the Pulp Fixtures source code repository) and use GnuPG to examine it:

```
$ gpg "$public_key_file"
pub   rsa2048 2016-08-05 [SC]
      6EDF301256480B9B801EBA3D05A5E6DA269D9D98
uid           Pulp QE
sub   rsa2048 2016-08-05 [E]
```

The last 32 bits (8 characters) of the key ID are what Pulp wants — in this example, 269D9D98.

pulp_smash.constants.**PUPPET_FEED_2 = 'http://forge.puppetlabs.com'**
The URL to a repository of Puppet modules.

pulp_smash.constants.**PUPPET_MODULE_1 = {'author': 'pulpqe', 'version': '0.1.0', 'name': 'dummypuppet'}**
Information about a Puppet module available via Pulp Fixtures.

pulp_smash.constants.**PUPPET_MODULE_2 = {'author': 'pulp', 'version': '1.0.0', 'name': 'pulp'}**
Information about a Puppet module available at *PUPPET_FEED_2*.

pulp_smash.constants.**PUPPET_MODULE_URL_1 = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/puppet/pulpqe-du**
The URL to a Puppet module module available via Pulp Fixtures.

Test cases that require a single module should use this URL, and test cases that require a feed should use *PUPPET_MODULE_URL_2*. Doing so shifts load away from the Puppet Forge.

Why do both URLs exist? Because simulating the Puppet Forge's behaviour is unreasonably hard.

Pulp Fixtures is designed to create data that can be hosted by a simple HTTP server, such as `python3 -m http.server`. A dynamic API, such as the Puppet Forge API, cannot be simulated. We could create a static tree of files, where that tree of files is the same as what the Puppet Forge would provide in response to a certain HTTP GET request. However:

- The Puppet Forge API will inevitably change over time as bugs are fixed and features are added. This will make a static facsimile of the Puppet Forge API outdated. This is more than a mere inconvenience: outdated information is also confusing!
- Without an in-depth understanding of each and every file the Puppet Forge yields, it is probable that static fixtures will be wrong from the get-go.

pulp_smash.constants.**PUPPET_MODULE_URL_2 = 'http://forge.puppetlabs.com/v3/files/pulp-pulp-1.0.0.tar.gz'**
The URL to a Puppet module available at *PUPPET_FEED_2*.

pulp_smash.constants.**PUPPET_QUERY_2 = 'pulp-pulp-1.0.0'**
A query that can be used to search for Puppet modules.

Built from *PUPPET_MODULE_2*.

Though the Puppet Forge API supports a variety of search types, Pulp only supports the ability to search for modules. As a result, it is impossible to create a Puppet repository and sync only an exact module or set of modules. This query intentionally returns a small number of Puppet modules. A query which selected a large number of modules would produce tests that took a long time and abused the free Puppet Forge service.

Beware that the Pulp API takes given Puppet queries and uses them to construct URL queries verbatim. Thus, if the user gives a query of "foo bar", the following URL is constructed:

> https://forge.puppet.com/modules.json/q=foo bar

In an attempt to avoid this error, this query is encoded before being submitted to Pulp.

pulp_smash.constants.**PYTHON_EGG_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/python-pypi/packages/3a**
The URL to a Python egg at *PYTHON_PYPI_FEED_URL*.

pulp_smash.constants.**PYTHON_PYPI_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/python-pypi/'**
The URL to the PyPI Python repository.

pulp_smash.constants.**PYTHON_WHEEL_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/python-pypi/packages/**
The URL to a Python egg at *PYTHON_PYPI_FEED_URL*.

pulp_smash.constants.**RPM = 'bear-4.1-1.noarch.rpm'**
The name of an RPM file. See *pulp_smash.constants.RPM_SIGNED_URL*.

pulp_smash.constants.**RPM2 = 'camel-0.1-1.noarch.rpm'**
The name of an RPM. See *pulp_smash.constants.RPM2_UNSIGNED_URL*.

pulp_smash.constants.**RPM2_UNSIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-unsigned/camel-**
The URL to an unsigned RPM file.

Built from *RPM_UNSIGNED_FEED_URL* and *RPM2*.

pulp_smash.constants.**RPM_ALT_LAYOUT_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-alt-layo**
The URL to a signed RPM repository. See *RPM_SIGNED_URL*.

pulp_smash.constants.**RPM_DATA = mappingproxy({'version': '4.1', 'arch': 'noarch', 'name': 'bear', 'metadata': {'sum**
Metadata for an RPM with an associated erratum.

The metadata tags that may be present in an RPM may be printed with:

```
rpm --querytags
```

Metadata for an RPM can be printed with a command like the following:

```
for tag in name epoch version release arch vendor; do
    echo "$(rpm -qp bear-4.1-1.noarch.rpm --qf "%{$tag}")"
done
```

There are three ways to measure the size of an RPM:

installed size The size of all the regular files in the payload.

archive size The uncompressed size of the payload, including necessary CPIO headers.

package size The actual size of an RPM file, as returned by `stat --format='%s' ...`.

For more information, see the Fedora documentation on RPM headers.

pulp_smash.constants.**RPM_ERRATUM_COUNT = 4**
The number of errata listed in *RPM_ERRATUM_URL*.

pulp_smash.constants.**RPM_ERRATUM_ID = 'RHEA-2012:0058'**
The ID of an erratum.

The package contained on this erratum is defined by *pulp_smash.constants.RPM_ERRATUM_RPM_NAME* and the erratum is present on repository which feed is *pulp_smash.constants.RPM_SIGNED_FEED_URL*.

pulp_smash.constants.**RPM_ERRATUM_RPM_NAME = 'gorilla'**
The name of the RPM named by *pulp_smash.constants.RPM_ERRATUM_ID*.

pulp_smash.constants.**RPM_ERRATUM_URL = 'https://repos.fedorapeople.org/repos/pulp/pulp/fixtures/rpm-erratum/erra**
The URL to an JSON erratum file for an RPM repository.

Note: This erratum is also used by several of the RPM repositories referenced in this module.

---

pulp_smash.constants.**RPM_INCOMPLETE_FILELISTS_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixture**
The URL to a repository with an incomplete `filelists.xml` file.

pulp_smash.constants.**RPM_INCOMPLETE_OTHER_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm**
The URL to a repository with an incomplete `other.xml` file.

pulp_smash.constants.**RPM_MIRRORLIST_BAD = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-mirrorlist-bad**
The URL to a mirrorlist file containing only invalid entries.

pulp_smash.constants.**RPM_MIRRORLIST_GOOD = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-mirrorlist-go**
The URL to a mirrorlist file containing only valid entries.

pulp_smash.constants.**RPM_MIRRORLIST_LARGE = 'https://mirrors.fedoraproject.org/metalink?repo=epel-7&arch=x8**
A mirrorlist referencing a large RPM repository.

pulp_smash.constants.**RPM_MIRRORLIST_MIXED = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-mirrorlist-m**
The URL to a mirrorlist file containing invalid and valid entries.

pulp_smash.constants.**RPM_MISSING_FILELISTS_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rp**
A repository that's missing its `filelists.xml` file.

pulp_smash.constants.**RPM_MISSING_OTHER_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-mis**
A repository that's missing its `other.xml` file.

pulp_smash.constants.**RPM_MISSING_PRIMARY_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-**
A repository that's missing its `primary.xml` file.

pulp_smash.constants.**RPM_NAMESPACES = {'metadata/rpm': 'http://linux.duke.edu/metadata/rpm', 'metadata/other':**
Namespaces used by XML-based RPM metadata.

Many of the XML files generated by the `createrepo` utility make use of these namespaces. Some of the files that use these namespaces are listed below:

metadata/common Used by `repodata/primary.xml`.

metadata/filelists Used by `repodata/filelists.xml`.

metadata/other Used by `repodata/other.xml`.

metadata/repo Used by `repodata/repomd.xml`.

metadata/rpm Used by `repodata/repomd.xml`.

pulp_smash.constants.**RPM_PKGLISTS_UPDATEINFO_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/**
A repository whose updateinfo file has multiple `<pkglist>` sections.

pulp_smash.constants.**RPM_SIGNED_FEED_COUNT = 32**
The number of packages available at *RPM_SIGNED_FEED_URL*.

pulp_smash.constants.**RPM_SIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-signed/'**
The URL to a signed RPM repository. See *RPM_SIGNED_URL*.

pulp_smash.constants.**RPM_SIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-signed/bear-4.1-1.no**
The URL to an RPM file.

Built from *RPM_SIGNED_FEED_URL* and *RPM*.

pulp_smash.constants.**RPM_UNSIGNED_FEED_COUNT = 32**
The number of packages available at *RPM_UNSIGNED_FEED_URL*.

pulp_smash.constants.**RPM_UNSIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-unsigned/'**
The URL to an unsigned RPM repository. See *RPM_SIGNED_URL*.

pulp_smash.constants.**RPM_UNSIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-unsigned/bear-4.1**
The URL to an unsigned RPM file.

Built from *RPM_UNSIGNED_FEED_URL* and *RPM*.

pulp_smash.constants.**RPM_UPDATED_INFO_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-upda**
A repository whose updateinfo file has an errata section.

pulp_smash.constants.**RPM_WITH_NON_ASCII_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-with-non**
The URL to an RPM with non-ascii metadata in its header.

pulp_smash.constants.**RPM_WITH_NON_UTF_8_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-with-non**
The URL to an RPM with non-UTF-8 metadata in its header.

pulp_smash.constants.**RPM_WITH_PULP_DISTRIBUTION_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtu**
The URL to a RPM repository with a PULP_DISTRIBUTION.xml file.

pulp_smash.constants.**RPM_WITH_VENDOR = 'rpm-with-vendor-1-1.fc25.noarch.rpm'**
The name of an RPM. See *pulp_smash.constants.RPM_WITH_VENDOR_URL*.

pulp_smash.constants.**RPM_WITH_VENDOR_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-with-v**
A repository whose primary.xml file has an vendor section.

pulp_smash.constants.**RPM_WITH_VENDOR_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/rpm-with-vendor/**
The URL of an RPM with a specified vendor in its header.

pulp_smash.constants.**SRPM = 'test-srpm02-1.0-1.src.rpm'**
An SRPM file at *pulp_smash.constants.SRPM_SIGNED_FEED_URL*.

pulp_smash.constants.**SRPM_SIGNED_FEED_COUNT = 3**
The number of packages available at *SRPM_SIGNED_FEED_URL*.

pulp_smash.constants.**SRPM_SIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/srpm-signed/'**
The URL to a signed SRPM repository.

pulp_smash.constants.**SRPM_SIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/srpm-signed/test-srpm0**
The URL to an SRPM file.

Built from *SRPM_SIGNED_FEED_URL* and *SRPM*.

pulp_smash.constants.**SRPM_UNSIGNED_FEED_COUNT = 3**
The number of packages available at *SRPM_UNSIGNED_FEED_COUNT*.

pulp_smash.constants.**SRPM_UNSIGNED_FEED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/srpm-unsigne**
The URL to an unsigned SRPM repository.

pulp_smash.constants.**SRPM_UNSIGNED_URL = 'https://repos.fedorapeople.org/pulp/pulp/fixtures/srpm-unsigned/test-s**
The URL to an unsigned SRPM file.

Built from *SRPM_UNSIGNED_FEED_URL* and *SRPM*.

6.6 *pulp_smash.exceptions*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.exceptions* Custom exceptions defined by Pulp Smash.

exception pulp_smash.exceptions.**BugStatusUnknownError**
We have encountered a bug whose status is unknown to us.

See *pulp_smash.selectors* for more information on how bug statuses are used.

exception pulp_smash.exceptions.**BugTPRMissingError**
We have encountered a bug with no "Target Platform Release" field.

See *pulp_smash.selectors* for more information.

exception pulp_smash.exceptions.**CallReportError**
A call report contains an error.

For more information about pulp's task handling, see Synchronous and Asynchronous Calls and Task Management.

exception pulp_smash.exceptions.**CalledProcessError**
Indicates a CLI process has a non-zero return code.

See *pulp_smash.cli.CompletedProcess()* for more information.

exception pulp_smash.exceptions.**ConfigFileNotFoundError**
We cannot find the requested Pulp Smash configuration file.

See *pulp_smash.config* for more information on how configuration files are handled.

exception pulp_smash.exceptions.**ConfigFileSectionNotFoundError**
We cannot read the requested Pulp Smash configuration file section.

See *pulp_smash.config* for more information on how configuration files are handled.

exception pulp_smash.exceptions.**ConfigValidationError**(*error_messages*, **args*, ***kwargs*)
The configuration file has validation errors.

See *pulp_smash.config.validate_config()* for more information on how configuration validation is handled.

exception pulp_smash.exceptions.**NoKnownBrokerError**
We cannot determine the AMQP broker used by a system.

An "AMQP broker" is a tool such as RabbitMQ or Apache Qpid.

exception pulp_smash.exceptions.**NoKnownPackageManagerError**
We cannot determine the package manager used by a system.

A "package manager" is a tool such as `yum` or `dnf`.

exception pulp_smash.exceptions.**NoKnownServiceManagerError**
We cannot determine the service manager used by a system.

A "service manager" is a tool such as `systemctl` or `service`.

exception pulp_smash.exceptions.**TaskReportError**(*msg*, *task*, **args*, ***kwargs*)
A task contains an error.

For more information about pulp's task handling, see Synchronous and Asynchronous Calls and Task Management.

exception pulp_smash.exceptions.**TaskTimedOutError**
We timed out while polling a task and waiting for it to complete.

See *pulp_smash.api.poll_spawned_tasks()* and *pulp_smash.api.poll_task()* for more information on how task polling is handled.

6.7 *pulp_smash.pulp_smash_cli*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.pulp_smash_cli* The entry point for Pulp Smash's command line interface.

6.8 *pulp_smash.selectors*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.selectors* Tools for selecting and deselecting tests.

pulp_smash.selectors.**bug_is_testable**(*bug_id*, *pulp_version*)
Tell the caller whether bug `bug_id` should be tested.

Parameters

- **bug_id** – An integer bug ID, taken from https://pulp.plan.io.
- **pulp_version** – A `packaging.version.Version` object telling the version of the Pulp server we are testing.

Returns `True` if the bug is testable, or `False` otherwise.

Raises `TypeError` if `bug_id` is not an integer.

Raises ***pulp_smash.exceptions.BugStatusUnknownError*** – If the bug has a status Pulp Smash does not recognize.

`pulp_smash.selectors.`**`bug_is_untestable`**(*bug_id*, *pulp_version*)
Return the inverse of *bug_is_testable()*.

`pulp_smash.selectors.`**`require`**(*version_string*)
Optionally skip a test method, based on a version string.

This decorator concisely encapsulates a common pattern for skipping tests. It can be used like so:

```
>>> from pulp_smash.config import get_config
>>> from pulp_smash.utils import require
>>> from unittest import TestCase
>>> class MyTestCase(TestCase):
...
...     @classmethod
...     def setUpClass(cls):
...         cls.cfg = get_config()
...
...     @require('2.7')  # References `self.cfg`
...     def test_foo(self):
...         pass  # Add a test for Pulp 2.7+ here.
```

Notice that `cls.cfg` is assigned to. This is a **requirement**.

Parameters **`version_string`** – A PEP 440 compatible version string.

`pulp_smash.selectors.`**`skip_if`**(*func*, *var_name*, *result*)
Optionally skip a test method, based on a condition.

This decorator checks to see if `func(getattr(self, var_name))` equals `result`. If so, a `unittest.SkipTest` exception is raised. Otherwise, nothing happens, and the decorated test method continues as normal. Here's an example:

```
>>> import unittest
>>> from pulp_smash.selectors import skip_if
>>> class MyTestCase(unittest.TestCase):
...
...     @classmethod
...     def setUpClass(cls):
...         cls.my_var = False
...
...     @skip_if(bool, 'my_var', False)
...     def test_01_skips(self):
...         pass
...
...     def test_02_runs(self):
...         type(self).my_var = True
...
...     @skip_if(bool, 'my_var', False)
...     def test_03_runs(self):
...         pass
```

Parameters **`var_name`** – A valid variable name.

6.9 *pulp_smash.tests*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests* Tests for Pulp.

This package contain functional tests for Pulp. These tests should be run against live Pulp systems. These tests may target all of the different interfaces exposed by Pulp, including its API and CLI. These tests are entirely different from the unit tests in *tests*, which test Pulp Smash itself.

The tests are organized into a hierarchy in the following form: `major_pulp_version.plugin.interface`. For example, Pulp 2 tests for the RPM plugin's API interface are in *pulp_smash.tests.pulp2.ostree.api_v2*.

There are several factors that determine whether or not a given test should run. These factors include:

- Which version of Pulp is under test? For example, version 2.13.2, 2.14.1, 3, and so on.
- Given that a certain version of Pulp is under test, which bugs affect that Pulp installation? For example, does https://pulp.plan.io/issues/2144 affect the Pulp installation under test?
- Which plugins are installed? For example, docker, ostree, puppet, and so on.

These are all common reasons that a given test might be skipped. From a user's perspective, this is all automatic. They need only install Pulp Smash, configure it, and point a test runner at *pulp_smash.tests*.

From a Pulp Smash developer's perspective, some work needs to be done to make this happen. There are two especially common ways to make this happen. First, bug-specific skipping logic can be implemented with the methods in *pulp_smash.selectors*, especially *pulp_smash.selectors.bug_is_testable()* and it's sibling. Second, a setUpModule function must be present in **every** `test*` module. In the simplest case, this can be done with pre-defined functions. For example, *pulp_smash.tests.pulp2.rpm.api_v2.test_broker* might do the following:

```
from pulp_smash.tests.pulp2.rpm.utils import set_up_module as setUpModule
```

By default, Requests refuses to make insecure HTTPS connections. You can override this behaviour by passing `verify=False`. For example:

```
requests.get('https://insecure.example.com', verify=False)
```

If you do this, Requests will make the connection. However, an `InsecureRequestWarning` will be raised. This is problematic. If a user has explicitly stated that they want insecure HTTPS connections and they are pestered about that fact, the user is effectively trained to ignore warnings.

This module attempts to solve that problem. When this module is imported, a filter is prepended to the warning module's list of filters. The filter states that `InsecureRequestWarning` warnings should be ignored. Note that this has no effect on whether SSL verification is performed. Insecure HTTPS connections are still blocked by default. The filter only has an effect on whether `InsecureRequestWarning` messages are displayed.

This filtering has a drawback: the warning module's filtering system is process-wide. Imagine an application which uses both Pulp Smash and some other library, and imagine that the other library generates `InsecureRequestWarning` warnings. The filter created here will suppress the warnings raised by that application. The `warnings.catch_warnings` context manager is not a good solution to this problem, as it is thread-unsafe.

This filtering is also limited: any Python code which executes after this module is imported can also manipulate the warning module's list of filters, and can therefore override the effects of this module. Unfortunately, the unittest test runner from the standard library does just that. As of this writing (with Python 3.5.3), the following filter (among others) is prepended to the warning module's filter list whenever a test case is executed:

```
>>> from warnings import filters
>>> filters[1]
('default', None, <class 'Warning'>, None, 0)
```

This filter matches all warnings — including `InsecureRequestWarning` — and causes them to be emitted. In theory, this does not occur if `-W 'ignore::requests.packages.urllib3.exceptions.InsecureRequestWarning'` is passed when calling `python -m unittest`. A more efficacious solution is to use a different test runner.

6.10 *pulp_smash.tests.pulp2*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2* Tests for Pulp 2.

6.11 *pulp_smash.tests.pulp2.constants*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.constants* Constants for Pulp 2 tests.

pulp_smash.tests.pulp2.constants.**CALL_REPORT_KEYS = frozenset({'result', 'error', 'spawned_tasks'})**
See: Call Report.

pulp_smash.tests.pulp2.constants.**CONSUMERS_ACTIONS_CONTENT_REGENERATE_APPLICABILITY_PATH = '**
See: Content Applicability.

pulp_smash.tests.pulp2.constants.**CONSUMERS_CONTENT_APPLICABILITY_PATH = '/pulp/api/v2/consumers/co**
See: Content Applicability.

pulp_smash.tests.pulp2.constants.**CONSUMERS_PATH = '/pulp/api/v2/consumers/'**
See: Consumer APIs.

pulp_smash.tests.pulp2.constants.**CONTENT_SOURCES_PATH = '/etc/pulp/content/sources/conf.d'**
See: Content Sources.

pulp_smash.tests.pulp2.constants.**CONTENT_UNITS_PATH = '/pulp/api/v2/content/units/'**
See: Search for Units.

pulp_smash.tests.pulp2.constants.**CONTENT_UPLOAD_PATH = '/pulp/api/v2/content/uploads/'**
See: Creating an Upload Request.

pulp_smash.tests.pulp2.constants.**ERROR_KEYS = frozenset({'_href', 'error', 'traceback', 'error_message', 'except**
See: Exception Handling.

No `href` field should be present. See Issue #1310.

pulp_smash.tests.pulp2.constants.**GROUP_CALL_REPORT_KEYS = frozenset({'_href', 'group_id'})**
See: Group Call Report.

pulp_smash.tests.pulp2.constants.**LOGIN_KEYS = frozenset({'key', 'certificate'})**
See: User Certificates.

pulp_smash.tests.pulp2.constants.**LOGIN_PATH = '/pulp/api/v2/actions/login/'**
See: Authentication.

pulp_smash.tests.pulp2.constants.**ORPHANS_PATH = 'pulp/api/v2/content/orphans/'**
See: Orphaned Content.

pulp_smash.tests.pulp2.constants.**PLUGIN_TYPES_PATH = '/pulp/api/v2/plugins/types/'**
See: Retrieve All Content Unit Types.

pulp_smash.tests.pulp2.constants.**PULP_SERVICES = {'httpd', 'pulp_celerybeat', 'pulp_workers', 'pulp_resource_**
Core Pulp services.

There are services beyond just these that Pulp depends on in order to function correctly. For example, an AMQP broker such as RabbitMQ or Qpid is integral to Pulp's functioning. However, if resetting Pulp (such as in *pulp_smash.utils.reset_pulp()*), this is the set of services that should be restarted.

pulp_smash.tests.pulp2.constants.**REPOSITORY_EXPORT_DISTRIBUTOR = 'export_distributor'**
A `distributor_type_id` to export a repository.

See: Export Distributors.

pulp_smash.tests.pulp2.constants.**REPOSITORY_GROUP_EXPORT_DISTRIBUTOR = 'group_export_distributor'**
A `distributor_type_id` to export a repository group.

See: Export Distributors.

pulp_smash.tests.pulp2.constants.**REPOSITORY_GROUP_PATH = '/pulp/api/v2/repo_groups/'**
See: Repository Group APIs

pulp_smash.tests.pulp2.constants.**REPOSITORY_PATH = '/pulp/api/v2/repositories/'**
See: Repository APIs.

pulp_smash.tests.pulp2.constants.**TASKS_PATH = '/pulp/api/v2/tasks/'**
See: Tasks APIs.

pulp_smash.tests.pulp2.constants.**USER_PATH = '/pulp/api/v2/users/'**
See: User APIs.

6.12 *pulp_smash.tests.pulp2.docker*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker* This package contains tests for the Docker plugin.

6.13 *pulp_smash.tests.pulp2.docker.api_v2*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2* This package contains tests for the Docker plugin with the Pulp v2 API.

6.14 *pulp_smash.tests.pulp2.docker.api_v2.test_copy*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_copy* Tests for copying docker units between repositories.

class pulp_smash.tests.pulp2.docker.api_v2.test_copy.**CopyV1ContentTestCase**(*methodName='runTest'*)
Copy data between Docker repositories with schema v1 content.

classmethod setUpClass()
Create class-wide variables.

classmethod tearDownClass()
Clean up resources.

test_01_set_up()
Create a repository and populate with with schema v1 content.

test_02_copy_images()
Copy tags from one repository to another.

Assert the same number of images are present in both repositories.

class pulp_smash.tests.pulp2.docker.api_v2.test_copy.**CopyV2ContentTestCase**(*methodName='runTest'*)
Copy data between Docker repositories with schema v2 content.

classmethod setUpClass()
Create class-wide variables.

classmethod tearDownClass()
Clean up resources.

test_01_set_up()
Create a repository and populate with with schema v2 content.

test_02_copy_manifest_lists()
Copy manifest lists from one repository to another.

Assert the same number of manifest lists are present in both repositories. This test targets:

- Pulp #2384
- Pulp #2385

test_02_copy_manifests()
Copy manifests from one repository to another.

Assert the same number of manifests are present in both repositories.

test_02_copy_tags()
Copy tags from one repository to another.

Assert the same number of tags are present in both repositories.

6.15 *pulp_smash.tests.pulp2.docker.api_v2.test_crud*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_crud* Test CRUD for Docker repositories.

This module contains tests for creating Docker repositories. It is intended to also contain read, update, and delete tests.

class pulp_smash.tests.pulp2.docker.api_v2.test_crud.**CrudTestCase**(*methodName='runTest'*)
CRUD a minimal Docker repository.

static create_body()
Return a dict for creating a repository.

static update_body()
Return a dict for creating a repository.

class pulp_smash.tests.pulp2.docker.api_v2.test_crud.**CrudWithFeedTestCase**(*methodName='runTest'*)
CRUD a Docker repository with a feed.

static create_body()
Return a dict, with a feed, for creating a repository.

class pulp_smash.tests.pulp2.docker.api_v2.test_crud.**UpdateTestCase**(*methodName='runTest'*)
Show it is possible to update a distributor for a docker repository.

classmethod setUpClass()
Create a docker repo with a distributor, and update the distributor.

Do the following:

1. Create a docker repository and add a distributor.
2. Update the distributor. Use the distributor's href in the request.
3. Update the distributor. Use the repository's href in the request, and ensure the distributor is updated by packing certain data in the request body.

test_status_codes()
Verify each of the server's responses has a correct status code.

test_update_accepted()
Verify the information sent to the server can be read back.

pulp_smash.tests.pulp2.docker.api_v2.test_crud.**setUpModule()**
Skip tests on Pulp versions lower than 2.8.

6.16 pulp_smash.tests.pulp2.docker.api_v2.test_duplicate_uploads

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_duplicate_uploads* Tests for how well Pulp can deal with duplicate uploads.

This module targets Pulp #1406 and Pulp Smash #81. The test procedure is as follows:

1. Create a new feed-less repository.
2. Upload content and import it into the repository. Assert the upload and import was successful.
3. Upload identical content and import it into the repository.

The second upload should silently fail for all Pulp releases in the 2.x series.

class pulp_smash.tests.pulp2.docker.api_v2.test_duplicate_uploads.**DuplicateUploadsTestCase**(*me*
Test how well Pulp can deal with duplicate content unit uploads.

classmethod setUpClass()
Create a Docker repository.

6.17 pulp_smash.tests.pulp2.docker.api_v2.test_sync

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_sync* Tests that sync docker repositories.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync.**UpstreamNameTestsMixin**
Provides tests that sync a repository and override `upstream_name`.

Any class inheriting from this mixin must also inherit from *`pulp_smash.utils.BaseAPITestCase`*.

test_invalid_upstream_name()
Sync the repository and pass an invalid `upstream_name`.

Verify the sync request is rejected with an HTTP 400 status code.

test_valid_upstream_name()
Sync the repository and pass a valid `upstream_name`.

Verify the sync succeeds.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync.**UpstreamNameV1TestCase**(*methodName='runTest'*)
Sync a v1 docker repository with various upstream_name options.

This test targets Pulp #2230.

classmethod setUpClass()
Create a docker repository with an importer.

The importer has no upstream_name set. it must be passed via override_config when a sync is requested.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync.**UpstreamNameV2TestCase**(*methodName='runTest'*)
Sync a v2 docker repository with various upstream_name options.

This test targets Pulp #2230.

classmethod setUpClass()
Create a docker repository with an importer.

The importer has no upstream_name set. it must be passed via override_config when a sync is requested.

pulp_smash.tests.pulp2.docker.api_v2.test_sync.**setUpModule**()
Skip tests on Pulp versions lower than 2.8.

6.18 pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish* Tests for syncing and publishing docker repositories.

pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**MANIFEST_LIST_V2 = {'description': 'Derived f**
A schema for docker manifest lists.

pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**MANIFEST_V1 = {'description': 'Derived from: h**
A schema for docker v2 image manifests, schema 1.

pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**MANIFEST_V2 = {'description': 'Derived from: h**
A schema for docker v2 image manifests, schema 2.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**NoAmd64LinuxTestCase**(*methodName='*
Sync a Docker image with no amd64/linux build.

A manifest list lets a single Docker repository contain multiple images. This is useful in the case where an image contains platform-specific code, and an image must be built for each each supported architecture, OS, etc.

When a modern Docker client fetches an image, it does the following:

1. Get a manifest list.
2. Look through the list of available images.
3. Pick out an image that functions on the current host's platform.
4. Get a manifest for that image.
5. Use the information in the manifest to get the image layers.

Older Docker clients aren't aware of manifest lists, and when they go to fetch an image, they just ask for any old manifest from a repository. When a Docker registry receives such a request, it does the following:

1. Look through the list of available images.

2. If an image with an `architecture` of `amd64` and an `os` of `linux` is available, return its manifest. Otherwise, return an HTTP 404.

This test case verifies Pulp's behaviour in the case where an upstream Docker repository has content described by a manifest list.

This test case doesn't verify Pulp's behaviour in the case where an upstream Docker repository has content described by a v2 manifest or v1 manifest. In these cases, the correct behaviour of a Docker registry is not well defined. See the backward compatibility documentation.

classmethod **setUpClass**()
Create class-wide variables.

classmethod **tearDownClass**()
Clean up resources.

test_01_set_up()
Create, sync and publish a Docker repository.

Specifically, do the following:

1. Create, sync and publish a Docker repository. Let the repository's upstream name reference a repository that has an image with a manifest list and no amd64/linux build.
2. Make Crane immediately re-read the metadata files published by Pulp. (Restart Apache.)

test_02_get_manifest_list()
Get a manifest list.

Assert that:

- The response headers include a content-type of `accept:application/vnd.docker.distribution.manifest.list.v2+json`. (See *test_02_get_manifest_list()*.
- The response body matches *MANIFEST_LIST_V2*.
- The returned manifest list doesn't include any entry where `architecture` of `amd64` and `os` is `linux`.

test_02_get_manifest_v1()
Get a v1 manifest. Assert that an HTTP 404 is returned.

test_02_get_manifest_v2()
Get a v2 manifest. Assert that an HTTP 404 is returned.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**NonNamespacedImageTestCase**(*metho*
Work with an image whose name has no namespace.

test_all()
Work with an image whose name has no namespace.

Create, sync and publish a Docker repository whose `UPSTREAM_NAME` doesn't include a namespace. A typical Docker image has a name like "library/busybox." When a non-namespaced image name like "busybox" is given, a prefix of "library" is assumed.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**RepoRegistryIdTestCase**(*methodName*
Show Pulp can publish repos with varying `repo_registry_id` values.

do_test(*cfg*, *repo_registry_id*)
Execute the test with the given `repo_registry_id`.

test_all()
Show Pulp can publish repos with varying `repo_registry_id` values.

The `repo_registry_id` setting defines a Docker repository's name as seen by clients such as the Docker CLI. It's traditionally a two-part name such as `docker/busybox`, but according to Pulp #2368, it can contain an arbitrary number of slashes. This test case verifies that the `repo_registry_id` can be set to values containing one, two and three slashes.

Also see: Pulp #2723.

class `pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.`**`SyncPublishMixin`**

Tools for test cases that sync and publish Docker repositories.

This class must be mixed in to a class that inherits from `unittest.TestCase`.

static **`adjust_url`**(*url*)

Return a URL that can be used for talking with Crane.

The URL returned is the same as `url`, except that the scheme is set to HTTP, and the port is set to (or replaced by) 5000.

Parameters **`url`** – A string, such as `https://pulp.example.com/foo`.

Returns A string, such as `http://pulp.example.com:5000/foo`.

static **`make_crane_client`**(*cfg*)

Make an API client for talking with Crane.

Create an API client for talking to Crane. The client returned by this method is similar to the following `client`:

```
>>> client = api.Client(cfg, api.json_handler)
```

However:

- The client's base URL is adjusted as described by *`adjust_url()`*.
- The client will send an `accept:application/json` header with each request.

Parameters **`cfg`** (`pulp_smash.config.PulpSmashConfig`) – Information about a Pulp deployment.

Returns An API client for talking with Crane.

Return type *pulp_smash.api.Client*

class `pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.`**`V1RegistryTestCase`**(*methodName='run*

Create, sync, publish and interact with a v1 Docker registry.

classmethod **`setUpClass`**()

Create class-wide variables.

classmethod **`tearDownClass`**()

Clean up resources.

`test_01_set_up`()

Create, sync and publish a repository.

Specifically, do the following:

1. Create, sync and publish a Docker repository. Let the repository's feed reference a v1 Docker registry.
2. Make Crane immediately re-read the metadata files published by Pulp. (Restart Apache.)

`test_02_get_crane_repositories`()

Issue an HTTP GET request to `/crane/repositories`.

Assert that the response is as described by Crane Admin.

test_02_get_crane_repositories_v1()
Issue an HTTP GET request to `/crane/repositories/v1`.

Assert that the response is as described by Crane Admin.

verify_v1_repo(*repo*)
Implement the assertions for the `test_02*` methods.

class pulp_smash.tests.pulp2.docker.api_v2.test_sync_publish.**V2RegistryTestCase**(*methodName='run*
Create, sync, publish and interact with a v2 Docker registry.

classmethod setUpClass()
Create class-wide variables.

classmethod tearDownClass()
Clean up resources.

test_01_set_up()
Create, sync and publish a Docker repository.

Specifically, do the following:

1. Create, sync and publish a Docker repository. Let the repository's feed reference a v2 Docker registry, and let the repository's upstream name reference an image with a manifest list.
2. Make Crane immediately re-read the metadata files published by Pulp. (Restart Apache.)

test_02_get_crane_repositories_v2()
Issue an HTTP GET request to `/crane/repositories/v2`.

Assert that the response is as described by Crane Admin.

test_02_get_manifest_list()
Issue an HTTP GET request to `/v2/{repo_id}/manifests/latest`.

Pass a header of `accept:application/vnd.docker.distribution.manifest.list.v2+json` Assert that:

- The response body matches *MANIFEST_LIST_V2*.
- The response has a content-type equal to what was requested. (According to Docker's *backward compatiblity* specification, if a registry is asked for a manifest list but doesn't have a manifest list, it may return a manifest instead. But this test targets manifest lists, and it will fail if that happens.)

test_02_get_manifest_v1()
Issue an HTTP GET request to `/v2/{repo_id}/manifests/latest`.

Pass each of the followng headers in turn:

- (none)
- `accept:application/json`
- `accept:application/vnd.docker.distribution.manifest.v1+json`

Assert the response matches *MANIFEST_V1*.

This test targets Pulp #2336.

test_02_get_manifest_v2()
Issue an HTTP GET request to `/v2/{repo_id}/manifests/latest`.

Pass a header of `accept:application/vnd.docker.distribution.manifest.v2+json`. Assert that the response body matches *MANIFEST_V2*.

This test targets Pulp #2336.

6.19 *pulp_smash.tests.pulp2.docker.api_v2.test_tags*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.test_tags* Tests that work with tags on docker repositories.

class pulp_smash.tests.pulp2.docker.api_v2.test_tags.**DockerTagTestCase**(*methodName='runTest'*)
Tests for docker repository tagging feature.

get_latest_tags()
Return all tags in this test's repo with a name of "latest".

Starting with Pulp 2.13, docker schema v2 is supported. As a result, a tag or manifest may be included in search results twice, with a differing schema_version. For convenience, if the Pulp system under test is version 2.13 or newer, only schema v1 is targeted. (The choice of schema version 1 is arbitrary.) See Pulp #2099.

Returns A list of docker tags.

get_manifests()
Return all manifests in this test's repo.

If the Pulp system under test is version 2.13 or newer, only return schema v1 manifests. See *get_latest_tags()*.

setUp()
Create and sync a docker repository.

test_create_tag()
Check if a tag can be created.

test_update_tag()
Check if a tag can be updated to a new manifest.

Do the following:

1. Find the tag in this test's docker repository whose name is "latest." Make note of the manifest it references.
2. Pick some other manifest. Update the repository so that the "latest" tag references the chosen manifest.
3. Find the tag in this test's docker repository whose name is "latest." Assert it references the chosen manifest.

test_update_tag_another_repo()
Check if tagging fail for a manifest from another repo.

test_update_tag_invalid_manifest()
Check if tagging fail for a invalid manifest.

pulp_smash.tests.pulp2.docker.api_v2.test_tags.**create_docker_repo**(*cfg*, *upstream_name*, *use_v1=False*)
Create a docker repository.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **upstream_name** – The Docker container upstream name.
- **use_v1** – If `True` use Docker V1 feed URL else use Docker V2 feed URL.

Returns Detailed information about the created repository.

pulp_smash.tests.pulp2.docker.api_v2.test_tags.**import_upload**(*cfg*, *repo*, *params*)
Create or update a docker repository.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **repo** – A dict of information about the targed repository.
- **params** – A dict of information to pass as import_upload body.

Returns A dict of information about the creation/update report.

pulp_smash.tests.pulp2.docker.api_v2.test_tags.**setUpModule**()
Skip tests on Pulp versions lower than 2.12.

6.20 pulp_smash.tests.pulp2.docker.api_v2.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.api_v2.utils* Utility functions for Docker API tests.

pulp_smash.tests.pulp2.docker.api_v2.utils.**gen_distributor**()
Return a semi-random dict for use in creating a Docker distributor.

pulp_smash.tests.pulp2.docker.api_v2.utils.**gen_repo**()
Return a semi-random dict that used for creating a Docker repo.

6.21 pulp_smash.tests.pulp2.docker.cli

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.cli* Tests that communicate with the server via the pulp-admin CLI client.

6.22 pulp_smash.tests.pulp2.docker.cli.test_crud

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.cli.test_crud* CRUD tests for docker repositories.

These tests can also be accomplished via the API. However, there is value in showing that the pulp-admin CLI client correctly interfaces with the API.

class pulp_smash.tests.pulp2.docker.cli.test_crud.**CreateTestCase**(*methodName='runTest'*)
Create docker repositories, both successfully and unsuccessfully.

setUp()
Provide a server config and a repository ID.

tearDown()
Delete created resources.

test_basic()
Create a docker repository. Only provide a repository ID.

Assert the return code is 0.

test_duplicate_ids()
Create two docker repositories with identical IDs.

Assert only the first repository is created.

test_with_feed_upstream_name()
Create a docker repository. Provide a feed and upstream name.

Assert the return code is 0.

class pulp_smash.tests.pulp2.docker.cli.test_crud.**DeleteV2TestCase**(*methodName='runTest'*)
Delete a populated v2 repository.

There was a bug in the Docker server plugin that caused a traceback when repos were deleted. This test ensures that that operation works correctly.

https://pulp.plan.io/issues/1296

classmethod setUpClass()
Provide a server config and a repository ID.

test_data_gone()
Assert that the published data was cleaned up.

test_repo_gone()
Assert that the repo is not in the pulp-admin output anymore.

test_success()
Assert that the CLI reported success.

class pulp_smash.tests.pulp2.docker.cli.test_crud.**UpdateDistributorTestCase**(*methodName='runTest'*)
Update a docker repository and use the `--repo-registry-id` flag.

This test case targets Pulp issue 1710. According to this bug, updating and using the `--repo-registry-id` flag would trigger a traceback.

classmethod setUpClass()
Provide a server config and a repository ID.

classmethod tearDownClass()
Delete created resources.

test_repo_registry_id_flag()
Verify the information sent to the server can be read back.

test_stdout()
Inspect the stdout emitted by `pulp-admin` when updating.

class pulp_smash.tests.pulp2.docker.cli.test_crud.**UpdateEnableV1TestCase**(*methodName='runTest'*)
Update a docker repository's –enable-v1 flag.

There was a bug in pulp-admin wherein the –enable-v1 flag could be set during repository creation, but not while updating repositories. This test ensures that behavior functions correctly.

classmethod setUpClass()
Provide a server config and a repository ID.

classmethod tearDownClass()
Delete created resources.

test_change_enable_v1_flag()
Test that the the –enable-v1 flag was successful.

test_success()
Assert that the CLI reported success.

class pulp_smash.tests.pulp2.docker.cli.test_crud.**UpdateEnableV2TestCase**(*methodName='runTest'*)
Update a docker repository's –enable-v2 flag.

There was a bug in pulp-admin wherein the –enable-v2 flag could be set during repository creation, but not while updating repositories. This test ensures that behavior functions correctly.

classmethod setUpClass()
Provide a server config and a repository ID.

classmethod tearDownClass()
Delete created resources.

test_change_enable_v2_flag()
Test that the the –enable-v2 flag was successful.

test_success()
Assert that the CLI reported success.

6.23 pulp_smash.tests.pulp2.docker.cli.test_sync_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.cli.test_sync_publish* Tests for syncing and publishing docker repositories.

class pulp_smash.tests.pulp2.docker.cli.test_sync_publish.**InvalidFeedTestCase**(*methodName='runTes*
Show Pulp behaves correctly when syncing a repo with an invalid feed.

test_all()
Create a docker repo with an invalid feed and sync it.

pulp_smash.tests.pulp2.docker.cli.test_sync_publish.**setUpModule**()
Execute `pulp-admin login`.

6.24 pulp_smash.tests.pulp2.docker.cli.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.cli.utils* Utility functions for docker CLI tests.

All of the functions in this module share a common structure. The first argument is a *pulp_smash.config.PulpSmashConfig*, and all other arguments correspond to command-line options. Most arguments are named after a flag. For example, an argument `to_repo_id` corresponds to the flag `--to-repo-id`.

For the meaning of each argument, see pulp-admin.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_copy**(*server_config*, *unit_type*, *from_repo_id=None*, *to_repo_id=None*)
Execute `pulp-admin docker repo copy {unit_type}`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_create**(*server_config*, *enable_v1=None*, *enable_v2=None*, *feed=None*, *repo_id=None*, *repo_registry_id=None*, *upstream_name=None*)
Execute `pulp-admin docker repo create`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_delete**(*server_config*, *repo_id*)
Execute `pulp-admin docker repo delete`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_list**(*server_config*, *repo_id=None*, *details=False*)
Execute `pulp-admin docker repo list`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_publish**(*server_config*, *repo_id*, *bg=None*, *force_full=None*)
Execute `pulp-admin docker repo publish run`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_search**(*server_config*, *unit_type*, *fields=None*, *repo_id=None*)
Execute `pulp-admin docker repo search {unit_type}`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_sync**(*server_config*, *repo_id*)
Execute `pulp-admin docker repo sync run`.

pulp_smash.tests.pulp2.docker.cli.utils.**repo_update**(*server_config*, *enable_v1=None*, *enable_v2=None*, *feed=None*, *repo_id=None*, *upstream_name=None*, *repo_registry_id=None*)
Execute `pulp-admin docker repo update`.

6.25 pulp_smash.tests.pulp2.docker.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.docker.utils* Utilities for Docker tests.

pulp_smash.tests.pulp2.docker.utils.**get_upstream_name**(*cfg*)
Return a Docker upstream name.

Return *pulp_smash.constants.DOCKER_UPSTREAM_NAME_NOLIST* if Pulp is older than version 2.14. Otherwise, return *pulp_smash.constants.DOCKER_UPSTREAM_NAME*. See the documentation for those constants for more information.

pulp_smash.tests.pulp2.docker.utils.**set_up_module**()
Skip tests if the Docker plugin is not installed.

See *pulp_smash.tests* for more information.

6.26 pulp_smash.tests.pulp2.ostree

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree* Functional tests for Pulp's ostree plugin.

6.27 pulp_smash.tests.pulp2.ostree.api_v2

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.api_v2* Tests that communicate with the server via the v2 API.

6.28 pulp_smash.tests.pulp2.ostree.api_v2.test_copy

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.api_v2.test_copy* Tests that copy content between OSTree repositories.

class pulp_smash.tests.pulp2.ostree.api_v2.test_copy.**FilterTestCase**(*methodName='runTest'*)
Copy content between OSTree repositories with a filter.

test_all()
Copy content between OSTree repositories with a filter.

Do the following:

1. Create a pair of repositories, and populate the first.
2. Randomly select a unit from the first repository, and copy it to the second repository.
3. Verify that the selected unit is the only one in the second repository.

6.29 pulp_smash.tests.pulp2.ostree.api_v2.test_crud

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.api_v2.test_crud* Test the CRUD API endpoints OSTree repositories.

class pulp_smash.tests.pulp2.ostree.api_v2.test_crud.**CreateDistributorsTestCase**(*methodName='run*
Show Pulp can create OSTree distributors and prevent path conflicts.

This test case targets:

- Pulp #1106
- Pulp #2769

classmethod setUpClass()
Create a pair of repositories.

Ensure the first repo has a distributor with a relative path. Succeeding tests will give the second repository distributors with relative paths, where those paths may or may not conflict with the first repository's distributor's relative path. This test splits the distributors across two repositories to ensure that Pulp correctly checks new relative paths against the existing relative paths in all repositories.

test_invalid_v1()
Create a distributor whose relative path is invalid.

Re-use the same relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `foo/bar`.

test_invalid_v2()
Create a distributor whose relative path is invalid.

Extend an existing relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `foo/bar/biz`.

test_invalid_v3()
Create a distributor whose relative path is invalid.

Prepend a slash onto an existing relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `/foo/bar`.

test_valid_v1()
Create a distributor whose relative path is valid.

Create a unique relative path. For example, if an existing relative path is `foo/bar`, then this relative path might be `biz/baz`.

test_valid_v2()
Create a distributor whose relative path is valid.

Create a relative path that contains three segments. Most other tests in this module have relative paths with two segments.

class pulp_smash.tests.pulp2.ostree.api_v2.test_crud.**CrudTestCase**(*methodName='runTest'*)
CRUD a minimal OSTree repository.

static create_body()
Return a dict for creating a repository.

static update_body()
Return a dict for creating a repository.

class pulp_smash.tests.pulp2.ostree.api_v2.test_crud.**CrudWithFeedTestCase**(*methodName='runTest'*)
CRUD an OSTree repository with a feed.

static create_body()
Return a dict, with a feed, for creating a repository.

class pulp_smash.tests.pulp2.ostree.api_v2.test_crud.**UpdateDistributorsTestCase**(*methodName='run*
Test the update of ostree distributors.

This test case targets:

- Pulp #1106
- Pulp #2769

classmethod setUpClass()
Create a pair of repositories.

Ensure each repo has a distributor with a relative path. Succeeding tests will update the second repository's distributor with varying relative paths, where those paths may or may not conflict with the first repository's distributor's relative path. This test splits the distributors across two repositories to ensure that Pulp correctly checks new relative paths against the existing relative paths in all repositories.

test_invalid_v1()
Update a distributor's relative path with an invalid value.

Re-use an existing relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `foo/bar`.

test_invalid_v2()
Update a distributor's relative path with an invalid value.

Extend an existing relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `foo/bar/biz`.

test_invalid_v3()
Update a distributor's relative path with an invalid value.

Prepend a slash to an existing relative path. For example, if an existing relative path is `foo/bar`, then this relative path would be `/foo/bar`.

test_valid_v1()
Update a distributor's relative path with a valid value.

Use a unique value for the new relative path. For example, if an existing relative path is `foo/bar`, then the new relative path might be `biz/baz`.

test_valid_v2()
Update a distributor's relative path with a valid value.

Use a three-segment value for the new relative path. Most other tests in this module have relative paths with two segments.

6.30 pulp_smash.tests.pulp2.ostree.api_v2.test_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.api_v2.test_publish* Tests that publish OSTree repositories.

class pulp_smash.tests.pulp2.ostree.api_v2.test_publish.**PublishTestCase**(*methodName='runTest'*)
Create, sync and publish an OSTree repository.

test_all()
Create, sync and publish an OSTree repository.

Verify that:

- The distributor's last_publish attribute is None after the sync. This demonstrates that auto_publish correctly defaults to False.
- The distributor's last_publish attribute is not None after the publish.

6.31 pulp_smash.tests.pulp2.ostree.api_v2.test_sync

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.api_v2.test_sync* Tests that sync OSTree repositories.

class pulp_smash.tests.pulp2.ostree.api_v2.test_sync.**SyncInvalidBranchesTestCase**(*methodName='ru*
Create an OSTree repository with invalid branches and sync it.

classmethod setUpClass()
Create and sync an OSTree repository.

class pulp_smash.tests.pulp2.ostree.api_v2.test_sync.**SyncInvalidFeedTestCase**(*methodName='runTest*
Create an OSTree repository with an invalid feed and sync it.

classmethod setUpClass()
Set cls.body.

class pulp_smash.tests.pulp2.ostree.api_v2.test_sync.**SyncMissingAttrsTestCase**(*methodName='runTes*
Create an OSTree repository with no feed or branches and sync it.

classmethod setUpClass()
Create and sync an OSTree repository.

class pulp_smash.tests.pulp2.ostree.api_v2.test_sync.**SyncTestCase**(*methodName='runTest'*)
Create an OSTree repository with a valid feed and branch, and sync it.

The sync should complete with no errors reported.

classmethod setUpClass()
Create an OSTree repository with a valid feed and branch.

test_task_progress_report()
Assert no task's progress report contains error details.

6.32 pulp_smash.tests.pulp2.ostree.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.ostree.utils* Utilities for interacting with OS tree.

pulp_smash.tests.pulp2.ostree.utils.**gen_distributor**()
Return a semi-random dict for use in creating an OSTree distributor.

For more information, see the generic repository CRUD documentation and the OSTree distributor configuration documentation.

pulp_smash.tests.pulp2.ostree.utils.**gen_repo**()
Return a semi-random dict for use in creating an OSTree repository.

pulp_smash.tests.pulp2.ostree.utils.**set_up_module**()
Skip tests if the OSTree plugin is not installed.

See *pulp_smash.tests* for more information.

6.33 pulp_smash.tests.pulp2.platform

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform* Functional tests for the Pulp platform.

According to the documentation:

> Pulp can be viewed as consisting of two parts, the platform (which includes both the server and client applications) and plugins (which provide support for a particular set of content types).

This package contains tests for the Pulp platform. These tests target plugin-agnostic functionality. These tests should not rely on any plugin-specific functionality, such as the ability to work with RPMs or ISOs.

6.34 pulp_smash.tests.pulp2.platform.api_v2

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2* Tests that communicate with the server via the v2 API.

6.35 pulp_smash.tests.pulp2.platform.api_v2.test_consumer

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_consumer* Test the consumer API endpoints.

class pulp_smash.tests.pulp2.platform.api_v2.test_consumer.**BindConsumerTestCase**(*methodName='run*
Show that one can bind a consumer to a repository.

test_all()
Bind a consumer to a distributor.

Do the following:

1. Create a repository with a distributor.
2. Create a consumer.
3. Bind the consumer to the distributor.

Assert that:

- The response has an HTTP 200 status code.
- The response body contains the correct values.

6.36 *pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability*
Test the API's *content applicability* functionality.

When a user asks Pulp to regenerate content applicability, Pulp responds with a call report and starts executing tasks in series. Starting with Pulp 2.8, it is possible for a user to explicitly request that Pulp execute tasks in parallel instead. This functionality is only available for certain API calls, and when this is done, Pulp returns a group call report instead of a regular call report. (See *pulp_smash.tests.pulp2.constants.CALL_REPORT_KEYS* and *pulp_smash.tests.pulp2.constants.GROUP_CALL_REPORT_KEYS*.) *SeriesTestCase* and *ParallelTestCase* test these two use cases, respectively.

class pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability.**FailureTestCase**(*method*
Fail to generate content applicability for consumers and repos.

classmethod setUpClass()
Make calls to the server and save the responses.

test_body()
Assert each response is JSON and doesn't look like a call report.

test_status_code()
Assert each response has an HTTP 400 status code.

class pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability.**ParallelTestCase**(*meth*
Ask Pulp to regenerate content applicability for consumers and repos.

Do so in parallel. See *pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability*.

setUp()
Ensure this test only runs on Pulp 2.8 and later.

classmethod setUpClass()
Make calls to the server and save the responses.

test_body()
Assert each response is JSON and has a correct structure.

class pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability.**SeriesTestCase**(*methodN*
Ask Pulp to regenerate content applicability for consumers and repos.

Do so in series. See *pulp_smash.tests.pulp2.platform.api_v2.test_content_applicability*.

classmethod setUpClass()
Make calls to the server and save the responses.

test_body()
Assert each response is JSON and has a correct structure.

6.37 pulp_smash.tests.pulp2.platform.api_v2.test_login

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_login* Test the API's authentication functionality.

class pulp_smash.tests.pulp2.platform.api_v2.test_login.**LoginTestCase**(*methodName='runTest'*)
Tests for logging in.

test_failure()
Unsuccessfully log in to the server.

Assert that:

- The response has an HTTP 401 status code.
- The response body is valid JSON and has correct keys.

test_success()
Successfully log in to the server.

Assert that:

- The response has an HTTP 200 status code.
- The response body is valid JSON and has correct keys.

6.38 pulp_smash.tests.pulp2.platform.api_v2.test_repository

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_repository* Test the repository API endpoints.

The assumptions explored in this module have the following dependencies:

```
It is possible to create an untyped repository.
- It is impossible to create a repository with a duplicate ID
|   or other invalid attributes.
- It is possible to read a repository, including its importers and
|   distributors.
- It is possible to update a repository.
- It is possible to delete a repository.
```

class pulp_smash.tests.pulp2.platform.api_v2.test_repository.**CreateFailureTestCase**(*methodName=*
Establish that repositories are not created in documented scenarios.

classmethod setUpClass()
Create several repositories.

Each repository is created to test a different failure scenario. The first repository is created in order to test duplicate ids.

test_body_status_code()
Assert that each response body has the expected HTTP status code.

test_exception_keys_json()
Assert the JSON body returned contains the correct keys.

test_location_header()
Assert that the Location header is correctly set in the response.

test_status_code()
Assert that each response has the expected HTTP status code.

class pulp_smash.tests.pulp2.platform.api_v2.test_repository.**CreateSuccessTestCase**(*methodName=*
Establish we can create repositories.

classmethod setUpClass()
Create several repositories.

Create one repository with the minimum required attributes, and a second with all available attributes except importers and distributors.

test_attributes()
Assert that each repository has the requested attributes.

test_location_header()
Assert the Location header is correctly set in each response.

According to RFC 7231, the HTTP Location header may be either an absolute or relative URL. Thus, given this request:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

These two responses are equivalent:

```
HTTP/1.1 302 FOUND
Location: http://www.example.com/index.php
```

```
HTTP/1.1 302 FOUND
Location: /index.php
```

This test abides by the RFC and allows Pulp to return either absolute or relative URLs.

test_status_code()
Assert each response has an HTTP 201 status code.

class pulp_smash.tests.pulp2.platform.api_v2.test_repository.**ReadUpdateDeleteTestCase**(*methodNa*
Establish we can read, update and delete repositories.

This test case assumes the assertions in *CreateSuccessTestCase* hold true.

classmethod setUpClass()
Create three repositories and read, update and delete them.

test_read()
Assert the "read" response body contains the correct attributes.

test_read_details()
Assert the read with details has the correct attributes.

test_read_imp_distrib()
Assert reading with importers/distributors returns correct attrs.

test_status_code()
Assert each response has a correct HTTP status code.

test_update_attributes_result()
Assert the "update" response body has the correct attributes.

test_update_spawned_tasks()
Assert the "update" response body mentions no spawned tasks.

6.39 pulp_smash.tests.pulp2.platform.api_v2.test_search

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_search* Test Pulp's Searching facilities.

The tests in this module make use of the User APIs. However, few user-specific references are made. These tests could be rewritten to use repositories or something else with only minimal changes. Thus, the name of this module.

Each test case executes one or more pairs of semantically identical POST and GET requests. Each pair of search results should match exactly.

Most test cases assume that the assertions in some other test case hold true. The assumptions explored in this module have the following dependencies:

```
It is possible to ask for all resources of a kind.
- It is possible to sort search results.
- It is possible to ask for a single field in search results.
- It is possible to ask for several fields in search results.
- It is possible to ask for a resource with a specific ID.
    - It is possible to ask for a resource with one of several IDs.
        - It is possible to skip some search results.
        - It is possible to limit how many search results are returned.
```

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**FieldTestCase**(*methodName='runTest'*)

Ask for a single field in search results.

GET	`{'field': 'name'}` (urlencoded)
POST	`{'criteria': {'fields': 'name'}}`

classmethod setUpClass()

Create one user. Execute searches.

test_field()

Only the requested key should be in each response.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**FieldsTestCase**(*methodName='runTest'*)

Ask for several fields in search results.

GET	`field=login&field=roles`
POST	`{'criteria': {'fields': ['login', 'roles']}}`

classmethod setUpClass()

Create one user. Execute searches.

test_fields()

Only the requested keys should be in each response.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**FiltersIdTestCase**(*methodName='runTest'*)

Ask for a resource with a specific ID.

There is no specification for executing these searches with GET.

GET	`{'filters': {'id': '...'}}` (urlencoded)
POST	`{'criteria': {'filters': {'id': '...'}}}`

classmethod setUpClass()

Search for exactly one user.

test_result_ids()

Assert the search results contain the correct IDs.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**FiltersIdsTestCase**(*methodName='runTest'*)
Ask for resources with one of several IDs.

There is no specification for executing these searches with GET.

GET	`{'filters': {'id': {'$in': ['...', '...']}}}`
POST	`{'criteria': {'filters': {'id': {'$in': ['...', '...']}}}}`

classmethod setUpClass()
Search for exactly two users.

test_result_ids()
Assert the search results contain the correct IDs.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**LimitSkipTestCase**(*methodName='runTest'*)
Ask for search results to be limited or skipped.

There is no specification for executing these searches with GET.

GET	`{'filters': {'id': {'$in': [id1, id2]}}, 'limit': 1}`
GET	`{'filters': {'id': {'$in': [id1, id2]}}, 'skip': 1}`
POST	`{'criteria': {'filters': {'id': {'$in': [id1, id2]}}, 'limit': 1}}`
POST	`{'criteria': {'filters': {'id': {'$in': [id1, id2]}}, 'skip': 1}}`

classmethod setUpClass()
Create two users. Execute searches.

test_results()
Check that one of the two created users has been found.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**MinimalTestCase**(*methodName='runTest'*)
Ask for all resources of a certain kind.

GET	no query parameters
POST	`{'criteria': {}}`

classmethod setUpClass()
Create one user. Execute searches.

test_user_found()
Assert each search result should include a user we created.

class pulp_smash.tests.pulp2.platform.api_v2.test_search.**SortTestCase**(*methodName='runTest'*)
Ask for sorted search results.

There is no specification for executing these searches with GET.

POST	`{'criteria': {'sort': [['id', 'ascending']]}}`
POST	`{'criteria': {'sort': [['id', 'descending']]}}`

classmethod setUpClass()
Create two users. Execute searches.

test_ascending()
Assert ascending results are ordered from low to high.

test_descending()
Assert descending results are ordered from high to low.

pulp_smash.tests.pulp2.platform.api_v2.test_search.**setUpModule**()
Create several users, each with a randomized login name.

Test cases may search for these users or otherwise perform non-destructive actions on them. Test cases should **not** change these users.

pulp_smash.tests.pulp2.platform.api_v2.test_search.**tearDownModule**()
Delete the users created by *setUpModule()*.

6.40 pulp_smash.tests.pulp2.platform.api_v2.test_user

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.api_v2.test_user* Test the user API endpoints.

The assumptions explored in this module have the following dependencies:

```
It is possible to create a user.
- It is impossible to create a duplicate user.
- It is possible to read a user.
- It is possible to update a user.
|   - It is possible to search for a (updated) user.
- It is possible to delete a user.
```

class pulp_smash.tests.pulp2.platform.api_v2.test_user.**CreateTestCase**(*methodName='runTest'*)
Establish that we can create users. No prior assumptions are made.

classmethod **setUpClass**()
Create several users.

Create one user with the minimum required attributes, and another with all available attributes.

test_attrs()
Assert that each user has the requested attributes.

test_password()
Assert that responses do not contain passwords.

test_status_code()
Assert that each response has an HTTP 201 status code.

class pulp_smash.tests.pulp2.platform.api_v2.test_user.**ReadUpdateDeleteTestCase**(*methodName='run*
Establish that we can read, update and delete users.

This test case assumes that the assertions in *CreateTestCase* are valid.

classmethod **setUpClass**()
Create three users and read, update and delete them respectively.

test_create_duplicate_user()
Verify one cannot create a user with a duplicate login.

test_password_not_in_response()
Ensure read and update responses do not contain a password.

Target https://bugzilla.redhat.com/show_bug.cgi?id=1020300.

test_status_code()
Ensure read, update and delete responses have 200 status codes.

test_updated_user()
Assert the updated user has the assigned attributes.

test_updated_user_password()
Assert one can log in with a user with an updated password.

test_use_deleted_user()
Assert one cannot read, update or delete a deleted user.

class pulp_smash.tests.pulp2.platform.api_v2.test_user.**SearchTestCase**(*methodName='runTest'*)
Establish we can search for users.

This test case assumes the assertions in *ReadUpdateDeleteTestCase* are valid.

classmethod setUpClass()
Create a user and add it to the 'super-users' role.

Search for:

- Nothing at all:
- All users having only the super-users role.
- All users having no roles.
- A user by their login.
- A non-existent user by their login.

test_global_search()
Assert the global search includes our user.

test_login_filter_exclusion()
Search for a non-existent user via the "login" filter.

test_login_filter_inclusion()
Search for a user via the "login" filter.

test_roles_filter_exclusion()
Assert that the "roles" filter can be used for exclusion.

test_roles_filter_inclusion()
Assert that the "roles" filter can be used for inclusion.

test_status_codes()
Assert each response has an HTTP 200 status code.

6.41 pulp_smash.tests.pulp2.platform.cli

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.cli* Tests that communicate with the server via the CLI.

6.42 pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db* Tests for the `pulp-manage-db` executable.

`pulp-manage-db` should only run when *REQUIRED_SERVICES* are running and when *CONFLICTING_SERVICES* are stopped. See:

- Pulp #2186
- Pulp Smash #487

class pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**BaseTestCase**(*methodName='runTest'*)
An abstract base class for the test cases in this module.

classmethod setUpClass()
Maybe skip this test case.

tearDown()
Start all of Pulp's services.

pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**CONFLICTING_SERVICES = frozenset({'pulp_**
If any of these services are running, pulp-manage-db will abort.

class pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**NegativeTestCase**(*methodName='runT*
Assert pulp-manage-db doesn't run when inappropriate.

test_celerybeat_running()
Test with pulp_celerybeat running.

test_conflicting_running()
Test with *CONFLICTING_SERVICES* running.

test_required_stopped()
Test with *REQUIRED_SERVICES* stopped.

test_resource_manager_running()
Test with pulp_resource_manager running.

This test targets Pulp #2684.

test_workers_running()
Test with pulp_workers running.

This test targets Pulp #2684.

class pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**PositiveTestCase**(*methodName='runT*
Assert pulp-manage-db runs when appropriate.

test_conflicting_stopped()
Test with *CONFLICTING_SERVICES* stopped.

test_dry_run()
Make sure pulp-manage-db runs if –dry-run is passed.

pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**REQUIRED_SERVICES = frozenset({'mongod'})**
If any of these services are stopped, pulp-manage-db will abort.

pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**setUpModule**()
Log in.

pulp_smash.tests.pulp2.platform.cli.test_pulp_manage_db.**tearDownModule**()
Reset Pulp, in case one of the test cases breaks Pulp.

6.43 *pulp_smash.tests.pulp2.platform.cli.test_selinux*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.cli.test_selinux* Tests to verify that Pulp has proper SELinux permissions.

pulp_smash.tests.pulp2.platform.cli.test_selinux.**CELERY_LABEL = ':system_r:celery_t:s0'**
A label commonly applied to celery processes.

The "user" segment of the label is intentionally omitted, as it is known to vary. See Pulp Smash #444 (comment).

class pulp_smash.tests.pulp2.platform.cli.test_selinux.**FileLabelsTestCase**(*methodName='runTest'*)
Test that files have correct SELinux labels.

This test case targets Pulp Smash #442.

classmethod setUpClass()
Create a CLI client.

test_pulp_celery_fc()
Test files listed in `pulp-celery.fc`.

test_pulp_server_fc()
Test files listed in `pulp-server.fc`.

test_pulp_streamer_fc()
Test files listed in `pulp-streamer.fc`.

pulp_smash.tests.pulp2.platform.cli.test_selinux.**HTTPD_LABEL = ':system_r:httpd_t:s0'**
A label commonly applied to httpd processes.

The "user" segment of the label is intentionally omitted, as it is known to vary. See Pulp Smash #444 (comment).

pulp_smash.tests.pulp2.platform.cli.test_selinux.**PS_FIELDS = ('label', 'args')**
The fields that `ps` should display. See `man ps` for details.

class pulp_smash.tests.pulp2.platform.cli.test_selinux.**Process**(*label*, *args*)
A single line of output from `ps`.

args
Alias for field number 1

label
Alias for field number 0

class pulp_smash.tests.pulp2.platform.cli.test_selinux.**ProcessLabelsTestCase**(*methodName='runTest*
Test that Pulp processes have correct SELinux labels.

This test case targets Pulp Smash #444.

classmethod setUpClass()
Get all of the processes running on the target Pulp system.

test_celerybeat()
Verify the label of the Pulp celerybeat process.

It should have a label of *CELERY_LABEL*.

test_httpd()
Verify the labels of the `wsgi:pulp*` processes.

They should have a label of *HTTPD_LABEL*.

test_reserved_resource_worker_0()
Verify the labels of the `reserved_resource_worker-0` processes.

They should have a label of *CELERY_LABEL*.

test_resource_manager()
Verify the labels of the `resource_manager` processes.

They should have a label of *CELERY_LABEL*.

6.44 *pulp_smash.tests.pulp2.platform.utils*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.platform.utils* Utilities for platform tests.

pulp_smash.tests.pulp2.platform.utils.**set_up_module**()
Skip tests if Pulp 2 isn't under test.

6.45 pulp_smash.tests.pulp2.puppet

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet* Functional tests for Pulp's Puppet plugin.

6.46 pulp_smash.tests.pulp2.puppet.api_v2

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2* Tests that communicate with the server via the v2 API.

6.47 pulp_smash.tests.pulp2.puppet.api_v2.test_crud

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2.test_crud* Tests that CRUD Puppet repositories.

class pulp_smash.tests.pulp2.puppet.api_v2.test_crud.**CRUDTestCase**(*methodName='runTest'*)
Test that one can create, update, read and delete a test case.

static create_body()
Return a dict for creating a repository.

static update_body()
Return a dict for creating a repository.

6.48 pulp_smash.tests.pulp2.puppet.api_v2.test_duplicate_uploads

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2.test_duplicate_uploads* Tests for how well Pulp can deal with duplicate uploads.

This module targets Pulp #1406 and Pulp Smash #81. The test procedure is as follows:

1. Create a new feed-less repository.
2. Upload content and import it into the repository. Assert the upload and import was successful.
3. Upload identical content and import it into the repository.

The second upload should silently fail for all Pulp releases in the 2.x series.

class pulp_smash.tests.pulp2.puppet.api_v2.test_duplicate_uploads.**DuplicateUploadsTestCase**(*me*
Test how well Pulp can deal with duplicate content unit uploads.

classmethod setUpClass()
Create a Puppet repository.

6.49 *pulp_smash.tests.pulp2.puppet.api_v2.test_install_distributor*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2.test_install_distributor* Tests for puppet_install_distributor.

For more information check puppet_install_distributor

class `pulp_smash.tests.pulp2.puppet.api_v2.test_install_distributor.`**`InstallDistributorTestCas`**
Test Puppet install distributor.

`test_all``()`
Test puppet_install_distributor.

Do the following:

1. Create a puppet repository with a puppet_install_distributor
2. Upload a puppet module
3. Publish the repository
4. Check if the puppet_install_distributor config was properly used

class `pulp_smash.tests.pulp2.puppet.api_v2.test_install_distributor.`**`InstallDistributorThrowsC`**
Test Puppet install distributor.

`test_all``()`
Creating a repo with an invalid distributor should throw an error.

This test targets Pulp #1237. Do the following:

1. Create a puppet repo
2. **Make an API call to create a distributor WITHOUT non-optional** install_path
3. Assert that an error is thrown
4. Assert that no repo is created

6.50 *pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish* Test the API endpoints for puppet repositories.

class `pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.`**`CreateTestCase`**`(`*methodName='runTest'*`)`
Create two puppet repos, with and without feed URLs respectively.

classmethod **`setUpClass`**`()`
Create two puppet repositories, with and without feed URLs.

`test_id_notes``()`
Validate the `id` and `notes` attributes for each repo.

`test_importer_config``()`
Validate the `config` attribute of each importer.

`test_importer_type_id``()`
Validate the `importer_type_id` attribute of each importer.

`test_number_importers``()`
Each repository should have only one importer.

class pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.**PublishTestCase**(*methodName='runTest*
Test repository syncing, publishing and data integrity.

Test uploading custom puppet module to repository, copying content between repositories, querying puppet modules from pulp server and integrity of downloaded modules. Three query formats are tested as are provided by different puppet versions:

- puppet version <= 3.3
- 3.3 < puppet version < 3.6
- puppet version > 3.6

classmethod setUpClass()
Upload puppet module to a repo, copy it to another, publish and download.

Create two puppet repositories, both without feeds. Upload an module to the first repository. Copy its content to the second repository. Add distributors to the repositories, publish repositories and download modules back from them.

test_call_report_errors()
Verify each call report is error-free.

test_call_report_keys()
Verify each call report has a sane structure.

- Import into a Repository
- Copying Units Between Repositories

test_free()
Verify the response body for ending an upload.

Delete an Upload Request

test_malloc()
Verify the response body for creating an upload request.

test_publish_errors()
Verify publishing a call report doesn't generate any errors.

test_publish_keys()
Verify publishing a repository generates a call report.

test_repo_units_consistency()
Verify the two puppet repositories have the same content units.

test_status_code()
Verify the HTTP status code of each server response.

test_unit_integrity()
Verify the integrity of the puppet modules downloaded from Pulp.

test_upload()
Verify the response body for uploading bits.

class pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.**SyncInvalidFeedTestCase**(*methodNam*
If an invalid feed is given a sync should complete with errors.

classmethod setUpClass()
Create a puppet repository with an invalid feed and sync it.

test_error_details()
Assert each task's progress report contains error details.

test_number_tasks()
Assert only one task was spawned.

test_status_code()
Assert the call to sync a repository returns an HTTP 202.

test_task_error_traceback()
Assert the task's "error" and "traceback" fields are non-null.

class pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.**SyncNoFeedTestCase**(*methodName='run*
Create and sync a puppet repository with no feed.

At least one of the sync tasks should fail. The task should fail in a graceful manner, without e.g. an internal tracebacks. This test targets Pulp #2628.

test_all()
Create and sync a puppet repository with no feed.

class pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.**SyncValidFeedTestCase**(*methodName=*
Create and sync puppet repositories with valid feeds.

sync_repo(*repo*)
Sync a repository, and verify no tasks contain an error message.

test_matching_query()
Sync a repository with a query that matches units.

Assert that:

- None of the sync tasks has an error message.
- Searching for module *pulp_smash.constants.PUPPET_MODULE_2* yields one result.
- The synced-in module can be downloaded.

test_non_matching_query()
Sync a repository with a query that doesn't match any units.

Assert that:

- None of the sync tasks has an error message.
- Searching for module *pulp_smash.constants.PUPPET_MODULE_2* yields no results.

class pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish.**SyncValidManifestFeedTestCase**(*me*
A valid Puppet manifest should sync correctly.

classmethod setUpClass()
Create repository with the feed pointing to a valid manifest.

test_number_tasks()
Assert only one task was spawned.

test_status_code()
Assert the call to sync a repository returns an HTTP 202.

test_task_progress_report()
Assert each task's progress report shows no errors.

6.51 *pulp_smash.tests.pulp2.puppet.api_v2.utils*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.api_v2.utils* Utility functions for Puppet API tests.

pulp_smash.tests.pulp2.puppet.api_v2.utils.**gen_distributor**()
Return a semi-random dict for use in creating a Puppet distributor.

pulp_smash.tests.pulp2.puppet.api_v2.utils.**gen_install_distributor**()
Return a semi-random dict used for creating a Puppet install distributor.

The caller must fill the install_path distributor_config option otherwise Pulp will throw an error when creating the distributor.

pulp_smash.tests.pulp2.puppet.api_v2.utils.**gen_repo**()
Return a semi-random dict that used for creating a puppet repo.

6.52 *pulp_smash.tests.pulp2.puppet.cli*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.cli* Tests that communicate with the server via the CLI.

6.53 *pulp_smash.tests.pulp2.puppet.cli.test_sync*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.cli.test_sync* Tests that sync Puppet repositories.

class pulp_smash.tests.pulp2.puppet.cli.test_sync.**SyncDownloadedContentTestCase**(*methodName='run*
Test whether Pulp can associate already-downloaded content with a repo.

Consider the following scenario:

1. Create a repository with a feed and sync it.
2. Create a second repository with the same feed and sync it.

When the second repository is synced, Pulp should recognize that the needed content units are already present, and it should associate them with the second repository. However, according to Pulp #1937, Pulp fails to do this, and the second repository will not be populated.

This test case tests Pulp #1937 and the corresponding Pulp Smash issue, Pulp Smash #269.

test_sync_downloaded_content()
Create two repositories with the same feed, and sync them serially.

More specifically, this test creates two puppet repositories with identical feeds, syncs them serially, and verifies that both have equal non-zero content unit counts.

class pulp_smash.tests.pulp2.puppet.cli.test_sync.**SyncFromPuppetForgeTestCase**(*methodName='runTes*
Test whether one can sync modules from the Puppet Forge.

According to Pulp #1846, Pulp sometimes fails to sync modules available on the Puppet forge. There is no consistency regarding which modules fail to sync: according to one comment, Pulp only detects 40 modules (of ~1,800), and according to another comment, Pulp syncs 4,089 of 4,128 modules.

How do we test this? The ideal test case is to repeatedly sync the entire Puppet Forge and check for sync failures, until the probability of another random failure is low. This is problematic: we're abusing the Puppet Forge and extending the test time. A more realistic test is to sync a small number of modules and ensure that no errors occur. This provides much less assurance, but it does at least show that *a* sync from the Puppet Forge can complete.

pulp_smash.tests.pulp2.puppet.api_v2.test_sync_publish already syncs from the Puppet Forge, but Pulp #1846 specifically uses the CLI.

The end result is a test case that syncs an unknown number of Puppet modules and which provides only minimal assurance that syncs from the Puppet Forge work. Unfortunately, we cannot do better.

test_sync_puppet_forge()
Create a Puppet repository and trigger a sync.

pulp_smash.tests.pulp2.puppet.cli.test_sync.**get_num_units_in_repo**(*server_config*, *repo_id*)
Tell how many puppet modules are in a repository.

Parameters

- **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp server being targeted.
- **repo_id** – A Puppet repository ID.

Returns The number of puppet modules in a repository, as an int.

pulp_smash.tests.pulp2.puppet.cli.test_sync.**setUpModule**()
Skip this module of tests if appropriate.

See Pulp #2574.

6.54 pulp_smash.tests.pulp2.puppet.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.puppet.utils* Utilities for Puppet tests.

pulp_smash.tests.pulp2.puppet.utils.**set_up_module**()
Skip tests if the Puppet plugin is not installed.

See *pulp_smash.tests* for more information.

6.55 pulp_smash.tests.pulp2.python

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python* Functional tests for Pulp's Python plugin.

6.56 pulp_smash.tests.pulp2.python.api_v2

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.api_v2* Tests that communicate with the server via the v2 API.

6.57 pulp_smash.tests.pulp2.python.api_v2.test_crud

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.api_v2.test_crud* Tests that CRUD Python repositories.

class pulp_smash.tests.pulp2.python.api_v2.test_crud.**CRUDTestCase**(*methodName='runTest'*)
Test that one can create, update, read and delete a test case.

static create_body()
Return a dict for creating a repository.

static update_body()
Return a dict for creating a repository.

6.58 pulp_smash.tests.pulp2.python.api_v2.test_duplicate_uploads

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.api_v2.test_duplicate_uploads* Tests for how well Pulp can deal with duplicate uploads.

This module targets Pulp #1406 and Pulp Smash #81. The test procedure is as follows:

1. Create a new feed-less repository.
2. Upload content and import it into the repository. Assert the upload and import was successful.
3. Upload identical content and import it into the repository.

The second upload should silently fail for all Pulp releases in the 2.x series.

class pulp_smash.tests.pulp2.python.api_v2.test_duplicate_uploads.**DuplicateUploadsTestCase**(*me*
Test how well Pulp can deal with duplicate content unit uploads.

classmethod setUpClass()
Create a Python repo. Upload a Python package into it twice.

6.59 pulp_smash.tests.pulp2.python.api_v2.test_sync_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.api_v2.test_sync_publish* Test the sync and publish API endpoints for Python repositories.

class pulp_smash.tests.pulp2.python.api_v2.test_sync_publish.**BaseTestCase**(*methodName='runTest'*)
A base class for the test cases in this module.

Test cases derived from this class (should) do the following:

1. Create and populate a Python repository. The procedure for populating the repository varies in each child class.
2. Create a second Python repository, and sync it from the first.

In each step, the `verify_*` methods are used if appropriate.

classmethod setUpClass()
Create class-wide variables.

classmethod tearDownClass()
Delete fixtures and orphans.

test_01_first_repo()
Create, populate and publish a Python repository.

Subclasses must override this method.

test_02_second_repo()
Create a second Python repository, and sync it from the first.

See:

- Pulp #140
- Pulp Smash #493

Note that, for Pulp #140 to be fully tested, an additional test case should be created wherein one Pulp application syncs from another completely independent Pulp application.

verify_package_types(*cfg*, *repo*)
Assert sdist and bdist_wheel shelf-reader packages were synced.

This test targets Pulp #1883.

verify_sync(*cfg*, *call_report*)
Verify the call to sync a Python repository succeeded.

Assert that:

- The call report has an HTTP 202 status code.
- None of the tasks spawned by the "sync" request contain errors.

class pulp_smash.tests.pulp2.python.api_v2.test_sync_publish.**SyncTestCase**(*methodName='runTest'*)
Test whether content can be synced into a Python repository.

test_01_first_repo()
Create, sync content into and publish a Python repository.

See:

- Pulp #135
- Pulp Smash #494

class pulp_smash.tests.pulp2.python.api_v2.test_sync_publish.**UploadTestCase**(*methodName='runTest'*)
Test whether content can be uploaded to a Python repository.

test_01_first_repo()
Create, upload content into and publish a Python repository.

See:

- Pulp #136
- Pulp #2334
- Pulp Smash #492

pulp_smash.tests.pulp2.python.api_v2.test_sync_publish.**get_details**(*cfg*, *repo*)
Return detailed information about a Python repository.

pulp_smash.tests.pulp2.python.api_v2.test_sync_publish.**get_repo_path**(*cfg*, *repo*)
Return the root path to a published Python repository.

6.60 *pulp_smash.tests.pulp2.python.api_v2.utils*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.api_v2.utils* Utility functions for Python API tests.

pulp_smash.tests.pulp2.python.api_v2.utils.**gen_distributor**()
Return a semi-random dict for use in creating a Python distributor.

pulp_smash.tests.pulp2.python.api_v2.utils.**gen_repo**()
Return a semi-random dict for use in creating a Python repository.

6.61 pulp_smash.tests.pulp2.python.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.python.utils* Utilities for Python tests.

pulp_smash.tests.pulp2.python.utils.**set_up_module**()
Skip tests if the Python plugin is not installed.

See *pulp_smash.tests* for more information.

6.62 pulp_smash.tests.pulp2.rpm

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm* Functional tests for Pulp's RPM plugin.

6.63 pulp_smash.tests.pulp2.rpm.api_v2

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2* Tests that communicate with the server via the v2 API.

6.64 pulp_smash.tests.pulp2.rpm.api_v2.test_broker

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_broker* Tests for Pulp's "broker reconnect" feature.

Tests for #55:

> Pulp offers a collection of behaviors known as "reconnect support" for the Pulp Broker. Here are the expected behaviors:
>
> - If you start a Pulp service that connects to the broker and the broker is not running or is not network accessible for some reason, the Pulp services will wait-and-retry. It has a backoff behavior, but the important part is that Pulp services don't exit if they can't connect due to availability, and when the availability problem is resolved, the Pulp services reconnect.
> - If you have a Pulp service connected to the broker and the broker shuts down, the Pulp services need the wait-and-retry as described above. Once the broker becomes available again the Pulp services should reconnect.

There are two scenarios to test here:

- support for initially connecting to a broker, and
- support for reconnecting to a broker that goes missing.

Both scenarios are executed by *pulp_smash.tests.pulp2.rpm.api_v2.test_broker.BrokerTestCase*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_broker.**BrokerTestCase**(*methodName='runTest'*)
Test Pulp's support for broker connections and reconnections.

health_check()
Execute step three of the test plan.

setUp()
Provide a server config and Pulp services to stop and start.

tearDown()
Ensure Pulp services and AMQP broker are running.

Stop all relevant services, then start them again. This approach is slow, but see when broker reconnect test fails, all following tests fail.

test_broker_connect()
Test Pulp's support for initially connecting to a broker.

Do the following:

1. Stop both the broker and several other services.
2. Start the several other resources, wait, and start the broker.
3. Test Pulp's health. Create an RPM repository, sync it, add a distributor, publish it, and download an RPM.

test_broker_reconnect()
Test Pulp's support for reconnecting to a broker that goes missing.

Do the following:

1. Start both the broker and several other services.
2. Stop the broker, wait, and start it again.
3. Test Pulp's health. Create an RPM repository, sync it, add a distributor, publish it, and download an RPM.

This test targets:

- Pulp #1635
- Pulp #2613

6.65 *pulp_smash.tests.pulp2.rpm.api_v2.test_character_encoding*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_character_encoding* Tests for Pulp's character encoding handling.

RPM files have metadata embedded in their headers. This metadata should be encoded as utf-8, and Pulp should gracefully handle cases where invalid byte sequences are encountered.

class pulp_smash.tests.pulp2.rpm.api_v2.test_character_encoding.**UploadNonAsciiTestCase**(*methodN*
Test whether one can upload an RPM with non-ascii metadata.

Specifically, do the following:

1. Create an RPM repository.
2. Upload and import *pulp_smash.constants.RPM_WITH_NON_ASCII_URL* into the repository.

test_all()
Test whether one can upload an RPM with non-ascii metadata.

class pulp_smash.tests.pulp2.rpm.api_v2.test_character_encoding.**UploadNonUtf8TestCase**(*methodNa*
Test whether one can upload an RPM with non-utf-8 metadata.

Specifically, do the following:

1. Create an RPM repository.
2. Upload and import *pulp_smash.constants.RPM_WITH_NON_UTF_8_URL* into the repository.

This test case targets Pulp #1903.

test_all()
Test whether one can upload an RPM with non-ascii metadata.

6.66 pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml* Verify an RPM repository's `comps.xml` file.

Each RPM repository has a `repodata` directory, in which various XML files containing metadata are present. This module houses test cases which verify the `comps.xml` file. For a sample `comps.xml` file, search through *pulp_smash.constants.RPM_SIGNED_FEED_URL*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml.**SyncRepoTestCase**(*methodName='runTest'*)
Sync in content from another RPM repository and publish it.

More specifically, this test case does the following:

1. Create a repository with a feed and a distributor.
2. Sync and publish the repository.
3. Verify the `comps.xml` file available in the published repository.

classmethod setUpClass()
Create, sync and publish a repository. Fetch its `comps.xml`.

test_first_level_element()
Verify the top-level element is named "comps".

test_langpacks_element()
Verify a `langpacks` element is in `comps.xml`.

Support for package langpacks has been added in Pulp 2.9. Consequently, this test is skipped on earlier versions of Pulp.

test_second_level_elements()
Verify the correct second-level elements are present.

class pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml.**UploadPackageGroupsTestCase**(*methodName=*
Upload custom package groups to an RPM repository and publish it.

More specifically, this test case does the following:

1. Create a repository without a feed.
2. Add yum distributor to the repository.
3. Generate several custom package groups. Upload each of them to Pulp, and import them into the repository.
4. Publish the repository.
5. Verify the `comps.xml` file available in the published repository.

classmethod setUpClass()
Create an RPM repository, upload package groups, and publish.

test_conditional_requires()
Assert `requires` attributes are correct on conditional packages.

This test assumes *test_packagelist_values()* has passed.

test_count()
Assert there is one "group" element per imported group unit.

test_default_default()
Assert that the default value of `default` tag is 'false'.

test_default_uservisible()
Assert that the default value of `uservisible` tag is 'false'.

test_display_order_occurences()
Assert `display_order` occurs once if omitted from the unit.

test_display_order_value()
Assert `display_order` is "1024" if omitted from the unit.

This test may be skipped if Pulp #1787 is open.

test_has_groups()
Assert that each imported group unit appears in the XML.

test_ids_alone()
Assert each "group" element has one "id" child element.

test_ids_unique()
Assert each group ID is unique.

test_one_task_per_import()
Assert only one task is spawned per package group upload.

test_packagelist_values()
Assert packagelist contains packagereq elements with correct text.

This test verifies that, for each of the 4 possible types of package in a group, the packagelist in the group XML contains exactly the package names in the uploaded unit.

test_root()
Assert the root element of the tree has a tag of "comps".

test_single_elements()
Assert that certain tags appear under groups exactly once.

test_tasks_result()
Assert each task's result success flag (if present) is true.

This test assumes *test_one_task_per_import()* passes.

test_tasks_state()
Assert each task's state is "finished".

This test assumes *test_one_task_per_import()* passes.

test_translated_string_count()
Assert that the XML has correct number of translated strings.

Some fields (name, description) are translatable. The tags for these fields are expected to appear once per translation, plus once for the untranslated string. This test verifies that this is the case.

test_translated_string_values()
Assert that the XML has correct values for translated strings.

Some fields (name, description) are translatable. The tags for these fields are expected to appear once per translation, plus once for the untranslated string. This test verifies that each translated string matches exactly the string provided when the group unit was imported.

test_verbatim_boolean_fields()
Assert boolean fields on a unit appear correctly in generated XML.

This test is similar to *test_verbatim_string_fields()*, but additionally verifies that boolean values are serialized as expected in the XML (i.e. as text 'true' or 'false').

test_verbatim_string_fields()
Assert string fields on a unit appear unmodified in generated XML.

This test covers fields from a group unit which are expected to be serialized as-is into top-level tags under a <group>. For example, this test asserts that the 'name' attribute on a group unit will appear in the generated XML as:

```
<group>
    <name>some-value</name>
    ...
</group>
```

class pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml.**UploadTwiceTestCase**(*methodName='runTest'*)
Upload a package group twice.

The first upload should create a new package group, and the second should update the existing package group. See: Pulp #2514.

test_all()
Upload a package group to a repository twice.

pulp_smash.tests.pulp2.rpm.api_v2.test_comps_xml.**setUpModule**()
Possibly skip the tests in this module.

Skip tests if Pulp #2277 affects us.

6.67 pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability* Tests for Pulp's content applicability feature.

class pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability.**BasicTestCase**(*methodName='ru*
Perform simple applicability generation tasks.

classmethod setUpClass()
Create and sync a repository.

The regular test methods that run after this can create consumers that bind to this repository.

classmethod tearDownClass()
Delete the repository created by *setUpClass()*.

test_negative()
Verify content isn't made available when appropriate.

Do the same as *test_positive()*, except that both packages' versions are equal to what's offered by the repository.

test_positive()
Verify content is made available when appropriate.

Specifically, do the following:

1. Create a consumer.

2. Bind the consumer to the repository created in *setUpClass()*.
3. Create a consumer profile where:
   - two packages are installed,
   - both packages' versions are lower than what's offered by the repository,
   - one of the corresponding packages in the repository has an applicable erratum, and
   - the other corresponding package in the repository doesn't have an applicable erratum.
4. Regenerate applicability for the consumer.
5. Fetch applicability for the consumer. Verify that both packages are listed as eligible for an upgrade.

pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability.**CONTENT_APPLICABILITY_REPORT_S**
A schema for a content applicability report for a consumer.

pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability.**RPM_WITHOUT_ERRATUM_METADATA =**
Metadata for an RPM without an associated erratum.

pulp_smash.tests.pulp2.rpm.api_v2.test_content_applicability.**RPM_WITH_ERRATUM_METADATA = map**
Metadata for an RPM with an associated erratum.

6.68 *pulp_smash.tests.pulp2.rpm.api_v2.test_content_sources*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_content_sources* Tests for Pulp's content sources feature.

class pulp_smash.tests.pulp2.rpm.api_v2.test_content_sources.**InvalidHeadersTestCase**(*methodName*
Define a content source with invalid headers.

See:

- Pulp #1282
- Pulp Smash #643

classmethod setUpClass()
Create a content source with invalid headers.

classmethod tearDownClass()
Destroy the created content source.

test_content_sources()
List Pulp's content sources.

Assert that the list doesn't include an entry corresponding to the newly defined content source configuration file.

test_refresh_content_sources()
Refresh Pulp's content sources.

Assert that the refresh completes successfully. It should complete successfully because the broken content source is completely skipped.

class pulp_smash.tests.pulp2.rpm.api_v2.test_content_sources.**ValidHeadersTestCase**(*methodName='*
Define a content source with valid headers.

See:

- Pulp #1282

- Pulp Smash #643

classmethod **setUpClass**()
Create a content source with valid headers.

classmethod **tearDownClass**()
Destroy the created content source.

test_list_content_sources()
List Pulp's content sources.

Assert that:

1. The list includes an entry corresponding to the newly defined content source configuration file.
2. This list entry includes the headers defined in the configuration file.

test_refresh_content_sources()
Refresh Pulp's content sources.

Assert that the refresh completes successfully.

6.69 *pulp_smash.tests.pulp2.rpm.api_v2.test_copy*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_copy* Test cases that copy content units.

class pulp_smash.tests.pulp2.rpm.api_v2.test_copy.**CopyErrataRecursiveTestCase**(*methodName='runTes*
Test that recursive copy of erratas copies RPM packages.

test_all()
Test that recursive copy of erratas copies RPM packages.

This test targets the following issues:

- Pulp Smash #769
- Pulp #3004

Do the following:

1. Create and sync a repository with errata, and RPM packages.
2. Create second repository.
3. Copy units from from first repository to second repository using `recursive` as true, and filter `type_id` as `erratum`.
4. Assert that RPM packages were copied.

class pulp_smash.tests.pulp2.rpm.api_v2.test_copy.**MtimeTestCase**(*methodName='runTest'*)
Test whether copied files retain their original mtime.

test_all()
Test whether copied files retain their original mtime.

This test targets the following issues:

- Pulp #2783
- Pulp Smash #720

Do the following:

1. Create, sync and publish a repository, with `generate_sqlite` set to true.

2. Get the `mtime` of the sqlite files.
3. Upload an RPM package into the repository, and sync the repository.
4. Get the `mtime` of the sqlite files again. Verify that the mtimes are the same.

6.70 *pulp_smash.tests.pulp2.rpm.api_v2.test_crud*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_crud* Tests that CRUD RPM repositories.

For information on repository CRUD operations, see Creation, Deletion and Configuration.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_crud.`**`CrudTestCase`**(*methodName='runTest'*)
CRUD a minimal RPM repository.

static `create_body`()
Return a dict for creating a repository.

static `update_body`()
Return a dict for updating a repository.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_crud.`**`CrudWithFeedTestCase`**(*methodName='runTest'*)
CRUD an RPM repository with a feed URL.

static `create_body`()
Return a dict for creating a repository.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_crud.`**`FeedURLUnquoteTestCase`**(*methodName='runTest'*)
Check that feed URLs are unquoted.

See https://pulp.plan.io/issues/2520.

`test_all`()
Ensure Pulp unquote feed URLs.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_crud.`**`LastUnitAddedTestCase`**(*methodName='runTest'*)
Tests for ensuring proper last_unit_added behavior.

`setUp`()
Perform common set-up tasks.

`test_update_on_copy`()
Check if copying units into a repo updates `last_unit_added`.

Do the following:

1. Create a repository with a feed and sync it.
2. Create a second repository. Assert the second repository's `last_unit_added` attribute is null.
3. Copy a content unit from first repository to the second. Assert the second repository's `last_unit_added` attribute is non-null.
4. Publish the second repository. Assert its `last_unit_added` attribute is non-null.

`test_update_on_sync`()
Check if syncing a repo updates `last_unit_added`.

Do the following:

1. Create a repository with a feed.
2. Assert the repository's `last_unit_added` attribute is null.

3. Sync the repository.
4. Assert the repository's `last_unit_added` attribute is non-null.

class pulp_smash.tests.pulp2.rpm.api_v2.test_crud.**NonExistentRepoTestCase**(*methodName='runTest'*)
Perform actions on non-existent repositories.

This test targets Pulp Smash #157.

setUp()
Set variables used by each test case.

test_delete()
Delete a non-existent repository.

test_update()
Update a non-existent repository.

class pulp_smash.tests.pulp2.rpm.api_v2.test_crud.**PulpDistributionTestCase**(*methodName='runTest'*)
Check if a feed with PULP_DISTRIBUTION.xml syncs properly.

See https://pulp.plan.io/issues/1086

test_all()
Check for content synced from a feed with PULP_DISTRIBUTION.xml.

class pulp_smash.tests.pulp2.rpm.api_v2.test_crud.**RPMDistributorTestCase**(*methodName='runTest'*)
RPM distributor tests.

test_update_checksum_type()
Check if RPM distributor can receive null checksum_type.

See: https://pulp.plan.io/issues/2134.

class pulp_smash.tests.pulp2.rpm.api_v2.test_crud.**RepositoryGroupCrudTestCase**(*methodName='runTes*
CRUD a minimal RPM repositories' groups.

For information on repositories' groups CRUD operations, see *Creation, Delete, and Update <http://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/repo/groups/cud.html>*

classmethod setUpClass()
Create, update, read and delete a repository group.

test_create()
Assert the created repository group has all requested attributes.

Walk through each of the attributes present on the create body and verify the attribute is present in the repository group.

test_read()
Assert the repo group update response has the requested changes.

test_status_codes()
Assert each response has a correct status code.

test_update()
Assert the repo group update response has the requested changes.

6.71 *pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies* Tests for Pulp's download policies, such as "background" and "on demand".

Beware that the test cases for the "on demand" download policy will fail if Pulp's Squid server is not configured to return an appropriate hostname or IP when performing redirection.

class pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies.**BackgroundTestCase**(*methodName='*
Ensure the "background" download policy works.

classmethod setUpClass()
Create an RPM repository with a valid feed and sync it.

Do the following:

1. Reset Pulp, including the Squid cache.
2. Create a repository with the "background" download policy.
3. Sync and publish the repository.
4. Download an RPM from the repository.

test_repo_local_units()
Assert that all content is downloaded for the repository.

test_request_history()
Assert that the request was serviced directly by Pulp.

If Pulp did not have the content available locally, it would redirect the client to the streamer and the rpm request would contain a history entry for that redirect.

test_rpm_checksum()
Assert the checksum of the downloaded RPM matches the metadata.

test_spawned_download_task()
Assert that a download task was spawned as a result of the sync.

class pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies.**FixFileCorruptionTestCase**(*metho*
Ensure the "on demand" download policy can fix file corruption.

classmethod get_rpm_abs_path()
Return the absolute path to *pulp_smash.constants.RPM*.

classmethod setUpClass()
Create an RPM repository and issue a task to download the repo.

Do the following:

1. Reset Pulp.
2. Create a repository with the "on demand" download policy.
3. Sync and publish the repository.
4. Trigger a repository download.
5. Corrupt a file in the repository.
6. Trigger a repository download, without unit verification.
7. Trigger a repository download, with unit verification.

test_corruption_occurred()
Assert corrupting a unit changes its checksum.

This is to ensure we actually corrupted the RPM and validates further testing.

test_start_end_checksums()
Verify Pulp's behaviour when `verify_all_units` is true.

Assert that the pre-corruption checksum of the unit is the same as the post-redownload checksum of the unit.

test_units_after_download()
Assert all units are downloaded after download_repo finishes.

test_units_before_download()
Assert no content units were downloaded besides metadata units.

test_verify_all_units_false()
Verify Pulp's behaviour when `verify_all_units` is false.

Assert that the checksum of the corrupted unit is unchanged, indicating that Pulp did not verify (or re-download) the checksum of the corrupted unit.

test_verify_all_units_true()
Verify Pulp's behaviour when `verify_all_units` is true.

Assert that the checksum of the corrupted unit is changed, indicating that Pulp did verify the checksum of the corrupted unit, and subsequently re-downloaded the unit.

class pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies.**OnDemandTestCase**(*methodName='run*
Ensure the "on demand" download policy works.

classmethod setUpClass()
Create an RPM repository with a valid feed and sync it.

Do the following:

1. Reset Pulp, including the Squid cache.
2. Create a repository with the "on demand" download policy.
3. Sync and publish the repository.
4. Download an RPM from the published repository.
5. Download the same RPM to ensure it is served by the cache.

test_local_units()
Assert no content units were downloaded besides metadata.

test_repository_units()
Assert there is at least one content unit in the repository.

test_request_history()
Assert the initial request received a 302 Redirect.

test_rpm_cache_control_header()
Assert the request has the Cache-Control header set.

test_rpm_cache_lookup_header()
Assert the first request resulted in a cache miss from Squid.

test_rpm_checksum()
Assert the checksum of the downloaded RPM matches the metadata.

test_same_rpm_cache_header()
Assert the second request resulted in a cache hit from Squid.

test_same_rpm_checksum()
Assert the checksum of the second RPM matches the metadata.

class pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies.**SwitchPoliciesTestCase**(*methodNa*
Ensure that repo's download policy can be updated and works.

Each test exercises a different download policy permutation by doing the following:

1. Create a repository configuring to one download policy
2. Read the repository and check if the download policy was properly set.
3. Update the repository to a different download policy.
4. Read the repository and check if the download policy was updated.
5. Sync the repository
6. Assert that the final download policy was used and works properly.

assert_background(*repo*, *tasks*)
Assert that background download policy is properly working.

assert_immediate(*repo*, *tasks*)
Assert that immediate download policy is properly working.

assert_on_demand(*repo*)
Assert that on_demand download policy is properly working.

repository_setup(*first*, *second*)
Set up a repository for download policy switch test.

Create a repository using the first download policy, assert it was set, update to the second download policy, assert it was set, then sync the repository and finally poll the spawned tasks.

Return a tuple with the repository and tasks.

setUp()
Make sure Pulp and Squid are reset.

test_background_to_immediate()
Check if switching from background to immediate works.

test_background_to_on_demand()
Check if switching from background to on_demand works.

test_immediate_to_background()
Check if switching from immediate to background works.

test_immediate_to_on_demand()
Check if switching from immediate to on_demand works.

test_on_demand_to_background()
Check if switching from on_demand to background works.

test_on_demand_to_immediate()
Check if switching from on_demand to immediate works.

pulp_smash.tests.pulp2.rpm.api_v2.test_download_policies.**setUpModule**()
Skip tests if the RPM plugin is not installed.

6.72 pulp_smash.tests.pulp2.rpm.api_v2.test_duplicate_uploads

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_duplicate_uploads* Tests for how well Pulp can deal with duplicate uploads.

class pulp_smash.tests.pulp2.rpm.api_v2.test_duplicate_uploads.**DuplicateUploadsTestCase**(*method*
Test how well Pulp can deal with duplicate unit uploads.

do_test (*feed*, *type_id*, *body*, *unit_key=None*)
Test how well Pulp can deal with duplicate unit uploads.

Do the following:

1. Create a new feed-less repository.
2. Upload content and import it into the repository. Assert the upload and import was successful.
3. Upload identical content and import it into the repository.

The second upload should silently fail for all Pulp releases in the 2.x series.

classmethod setUpClass()
Set a class-wide variable.

test_iso()
Upload duplicate ISO content. See *do_test()*.

This test targets the following issues:

- Pulp Smash #582
- Pulp #2274

test_rpm()
Upload duplicate RPM content.See *do_test()*.

This test targets the following issues:

- Pulp Smash #81
- Pulp #1406

6.73 pulp_smash.tests.pulp2.rpm.api_v2.test_errata

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_errata* Test whether applying an erratum will update an RPM package version.

class pulp_smash.tests.pulp2.rpm.api_v2.test_errata.**ApplyErratumTestCsae**(*methodName='runTest'*)
Test whether applying an erratum will install referenced RPMs.

test_all()
Test whether applying an erratum will install referenced RPMs.

It does the following:

1. Create, sync and publish a repository with errata and RPM packages. Two of these RPMs have the same name and different versions. The newest version is referenced by one of the errata.
2. Install the older version of the aforementioned RPM.
3. Apply the erratum, and verify that newer version of the RPM is installed.

This test targets Pulp Smash #760.

6.74 pulp_smash.tests.pulp2.rpm.api_v2.test_export

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_export* Test the API's *Export Distributors* feature.

class pulp_smash.tests.pulp2.rpm.api_v2.test_export.**BaseExportChecksumTypeTestCase**(**args*, ***kwargs*)

Base class for repo and repo group export with checksum type tests.

get_export_entity()
Provide the export entity.

The entity can be either a repository or a repository group.

get_export_entity_type()
Provide the export entity type.

Return `repos` for repository and `repo_group` for repository group.

get_repomd_iso_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path within an exported ISO.

Repository and repository group exports have a different path to repomd.xml file within the exported ISO.

get_repomd_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path.

Repository and repository group exports have a different path to repomd.xml file.

classmethod setUpClass()
Create and sync a repository.

Also skips the test if Pulp version less than 2.9.

Children of this base class must provide a `distributors` attribute with the list of distributors to export a repository or repository group.

test_publish_to_dir_checksum_type()
Publish to a directory choosing the checksum type.

test_publish_to_web_checksum_type()
Publish to web choosing the checksum type.

class pulp_smash.tests.pulp2.rpm.api_v2.test_export.**ExportChecksumTypeTestCase**(**args*, ***kwargs*)

Publish a repository choosing the distributor checksum type.

get_export_entity()
Provide the export entity.

get_export_entity_type()
Provide the export entity type.

get_repomd_iso_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path within an exported ISO.

get_repomd_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path.

classmethod setUpClass()
Create some distributors.

Each distributor is configured with a valid checksum type.

class pulp_smash.tests.pulp2.rpm.api_v2.test_export.**ExportDirMixin**(**args*, ***kwargs*)

Mixin with repo export to dir utilities.

A mixin with methods for managing an export directory on a Pulp server. This mixin is designed to support the following work flow:

1. Create a directory. (See *create_export_dir()*.)
2. Export a repository to the directory.
3. Make the directory readable. (See *change_export_dir_owner()*.)
4. Inspect the contents of the directory.

publish_to_dir() conveniently executes steps 1–3. Consider using the other methods only if more granularity is needed.

A class attribute named `cfg` must be present. It should be a *pulp_smash.config.PulpSmashConfig*.

change_export_dir_owner(*export_dir*)
Change the owner to the running Pulp Smash user.

Update the remote path `dir` owner to the user running Pulp Smash.

create_export_dir()
Create a directory, and ensure Pulp can export to it.

Create a directory, and set its owner and group to `apache`. If Pulp issue 616 affects the current Pulp system, disable SELinux, and schedule a clean-up command that re-enables SELinux.

> **Warning:** Only call this method from a unittest `test*` method. If called from elsewhere, SELinux may be left disabled.

Returns The path to the created directory, as a string.

publish_to_dir(*entity_href*, *distributor_id*)
Create an export directory, publish to it, and change its owner.

For details, see *ExportDirMixin*.

Parameters

- **entity_href** – The path to an entity such as a repository or a repository group.
- **distributor_id** – The ID of the distributor to use when exporting.

Returns The path to the export directory.

sudo()
Return either `''` or `'sudo '`.

Return the former if root, and the latter if not.

class pulp_smash.tests.pulp2.rpm.api_v2.test_export.**ExportDistributorTestCase**(**args*, ***kwargs*)
Establish we can publish a repository using an export distributor.

setUp()
Optionally create an export distributor.

Create an export distributor only if one is not already present. (See *setUpClass()*.)

classmethod setUpClass()
Create and sync a repository. Optionally create a distributor.

Skip creating the distributor if we are testing Pulp 2.9 and it is affected by Pulp #1928.

test_publish_to_dir()
Publish the repository to a directory on the Pulp server.

gerify that *pulp_smash.constants.RPM* is present and has a correct checksum.

This test is skipped if selinux is installed and enabled on the target system an Pulp issue 616 is open.

test_publish_to_web()
Publish the repository to the web, and fetch the ISO file.

The ISO file should be available over both HTTP and HTTPS. Fetch it from both locations, and assert that the fetch was successful.

class pulp_smash.tests.pulp2.rpm.api_v2.test_export.**RepoGroupExportChecksumTypeTestCase**(**args*, ***kwa*

Publish a repository group choosing the distributor checksum type.

get_export_entity()
Provide the export entity.

get_export_entity_type()
Provide the export entity type.

get_repomd_iso_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path within an exported ISO.

get_repomd_publish_path(*export_dir*, *distributor*)
Provide the repomd.xml publish path.

classmethod setUpClass()
Create required entities for publishing a repository group.

Do the following:

1. Create a repository group and add the repository created by super call.
2. Creates some distributors. Each distributor is configured with a valid checksum type.

pulp_smash.tests.pulp2.rpm.api_v2.test_export.**setUpModule**()
Possibly skip the tests in this module.

Skip tests if Pulp #2277 affects us.

6.75 pulp_smash.tests.pulp2.rpm.api_v2.test_force_full

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_force_full* Tests for publishing with the `force_full` parameter set.

When a repository is published, Pulp checks to see if certain steps can be skipped. For example, if one publishes a repository twice in a row, the second publish is a no-op, as nothing needs to be done. As of Pulp 2.9, clients can tell Pulp to perform a "full" publish. When this is done, Pulp executes all publication steps, even steps that normally would be skipped.

This module tests Pulp's handling of "full" publishes.

class pulp_smash.tests.pulp2.rpm.api_v2.test_force_full.**ForceFullTestCase**(*methodName='runTest'*)
Test the `force_full` option.

Repeatedly publish a repository. Set the `force_full` option to various values, and try omitting it too.

get_step(*steps*, *step_type*)
Return the task step with the given `step_type`.

Parameters

- **steps** – A list of dicts, as can be accessed by `call_report_task['result']['details']`.
- **step_type** – The `step_type` of a particular step, such as `save_tar`.

Returns The matching step.

Raises An assertion error if exactly one match is not found.

classmethod setUpClass()
Create and sync a repository.

test_01_force_full_false()
Publish the repository and set `force_full` to false.

A full publish should occur.

test_02_force_full_omit()
Publish the repository and omit `force_full`.

A fast-forward publish should occur. This test targets Pulp #1966.

test_03_force_full_true()
Publish the repository and set `force_full` to true.

A full publish should occur. The "force" publish feature was introduced in Pulp 2.9, and as such, this test will skip when run against an older version of Pulp. See Pulp #1938.

6.76 *pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud* Test CRUD for ISO RPM repositories.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud.**AddImporterDistributorTestCase**(*methodNam*
Add an importer and a distributor to an existing untyped repository.

See:

- Associate an Importer to a Repository
- Associate a Distributor with a Repository

classmethod setUpClass()
Create a repository and add an importer and distributor to it.

Do the following:

1. Create a repository.
2. Read the repository's importers and distributors.
3. Add an importer and distributor to the repo.
4. Re-read the repository's importers and distributors.

test_add_distributor()
Check the HTTP status code for adding a distributor.

test_add_importer()
Check the HTTP status code for adding an importer.

test_after()
Verify the repository ends up with one importer and distributor.

test_before()
Verify the repository has no importer or distributors initially.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud.**CreateTestCase**(*methodName='runTest'*)
Create an ISO RPM repo with an importer and distributor.

classmethod setUpClass()
Create an ISO RPM repo with an importer and distributor.

test_attributes()
Assert the created repository has the requested attributes.

test_headers_location()
Assert the response's `Location` header is correct.

The `Location` may be either an absolute or relative URL. See *pulp_smash.tests.pulp2.platform.api_v2.test_repository.CreateSuccessTestCase.test_location_header()*.

test_status_code()
Assert the response has an HTTP 201 status code.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud.**PulpManifestTestCase**(*methodName='runTest'*)
Ensure ISO repo properly handles PULP_MANIFEST information.

static parse_pulp_manifest(*feed_url*)
Parse PULP_MANIFEST information from `feed_url`/PULP_MANIFEST.

Parameters **feed_url** – The URL for the file feed. It will be joined with /PULP_MANIFEST in order to find the PULP_MANIFEST file.

Returns A list of dicts mapping each PULP_MANIFEST row. The dict contains the keys name, checksum and size which represent the PULP_MANIFEST data.

test_invalid_file_feed()
Create and sync a ISO repo from an invalid file feed.

Assert that the sync fails with the information that some units were not available.

test_valid_file_feed()
Create and sync a ISO repo from a file feed.

Assert that the number of units synced is the same as PULP_MANIFEST lists.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_crud.**ReadUpdateDeleteTestCase**(*methodName='runT*
Establish that we can interact with typed repositories as expected.

classmethod setUpClass()
Create three repositories and read, update and delete them.

test_read()
Assert that the "read" call returns correct attributes.

test_read_distributors()
Assert each read w/distributors contains info about distributors.

test_read_importers()
Assert each read with importers contains info about importers.

test_status_code()
Assert each response has a correct HTTP status code.

6.77 pulp_smash.tests.pulp2.rpm.api_v2.test_iso_sync_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_iso_sync_publish* Tests that sync and publish ISO repositories.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_sync_publish.**ServeHttpsFalseTestCase**(*methodNa*
Publish w/an rsync distributor when serve_https is false.

More precisely, do the following:

1. Create an ISO RPM repository. Ensure the repository has distributors of type iso_distributor and iso_rsync_distributor, and ensure the former distributor's publish_https attribute is false.
2. Populate the repository with some content.
3. Publish the repository with both distributors. Assert that the ISO rsync distributor successfully places files on the target system.

This test targets Pulp #2657. According to this issue, the ISO rsync distributor will fail to publish files if the the ISO distributor has not published files via HTTPS.

setUp()
Set the pulp_manage_rsync boolean.

tearDown()
Reset the pulp_manage_rsync boolean.

test_all()
Publish w/an rsync distributor when serve_https is false.

class pulp_smash.tests.pulp2.rpm.api_v2.test_iso_sync_publish.**UploadIsoTestCase**(*methodName='run*
Upload an ISO file into an ISO repository.

test_all()
Upload an ISO file into an ISO repository.

Specifically, do the following:

1. Create an ISO repository.
2. Upload *pulp_smash.constants.FILE_URL* to the repository.
3. Publish the repository.
4. Download the published ISO, and assert it's equal to the uploaded ISO.

6.78 pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist* Tests that exercise Pulp's support for mirrorlist feeds.

The tests in this module target:

- Pulp #175: "As a user, I can specify mirrorlists for rpm repository feeds."
- Pulp #2224: "Cannot sync from mirrorlists."

The test cases in this module reference "good," "mixed" and "bad" mirrorlist files. A "good" file contains only valid references, a "mixed" file contains both valid and invalid references, and a "bad" file contains only invalid references.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**BadMirrorlistTestCase**(*methodName='runTes*
Create a repository that references a "bad" mirrorlist file.

Do the following:

1. Create a repository. Make its importer `feed` option reference a "bad" mirrorlist file.
2. Sync the repository. Expect a failure.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**BadRelativeUrlTestCase**(*methodName='runT*
Like *BadMirrorlistTestCase*, but pass `relative_url` too.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**GoodMirrorlistTestCase**(*methodName='runT*
Create a repository that references a "good" mirrorlist file.

Do the following:

1. Create a repository. Make its importer `feed` option reference a "good" mirrorlist file.
2. Sync and publish the repository.
3. Download and verify a file from the published repository.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**GoodRelativeUrlTestCase**(*methodName='run*
Like *GoodMirrorlistTestCase*, but pass `relative_url` too.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**MixedMirrorlistTestCase**(*methodName='run*
Create a repository that references a "mixed" mirrorlist file.

1. Create a repository. Make its importer `feed` option reference a "mixed" mirrorlist file.
2. Sync and publish the repository.
3. Download and verify a file from the published repository.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**MixedRelativeUrlTestCase**(*methodName='ru*
Like *MixedMirrorlistTestCase*, but pass `relative_url` too.

test_all()
Execute the test case business logic.

class pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.**UtilsMixin**
A mixin providing methods to the test cases in this module.

Any class inheriting from this mixin must also inherit from `unittest.TestCase` or a compatible clone.

check_issue_2277(*cfg*)
Skip the current test method if Pulp issue #2277 affects us.

check_issue_2321(*cfg*)
Skip the current test method if Pulp issue #2321 affects us.

check_issue_2326(*cfg*)
Skip the current test method if Pulp issue #2326 affects us.

check_issue_2363(*cfg*)
Skip the current test method if Pulp issue #2363 affects us.

create_repo(*cfg*, *feed*, *relative_url=None*)
Create an RPM repository with a yum importer and distributor.

In addition, schedule the repository for deletion with `addCleanup`.

Parameters

- **cfg** (`pulp_smash.config.PulpSmashConfig`) – The Pulp deployment on which to create a repository.
- **feed** – A value for the yum importer's `feed` option.
- **relative_url** – A value for the yum distributor's `relative_url` option. If `None`, this option is not passed to Pulp.

Returns A detailed dict of information about the repository.

`pulp_smash.tests.pulp2.rpm.api_v2.test_mirrorlist.`**tearDownModule**()
Delete orphan content units.

6.79 pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish* Test Pulp's handling of no-op publishes.

As of version 2.9, Pulp has the ability to perform no-op publishes, instead of the more typical full publish. A no-op publish is one in which no files are changed, including metadata files. No-op publishes occur when all of the components — including the repository, distributor and override config — are unchanged.

Tests for this feature include the following:

- Publish a repository twice. (See *PubTwiceTestCase*.)
- Publish a repository, change it, and publish it again. (See *ChangeRepoTestCase*.)
- Publish a repository twice, with an override config the second time. (See *pulp_smash.tests.pulp2.rpm.api_v2.test_export.ExportDistributorTestCase*.)
- Publish a repository twice, with an override config both times. (See *PubTwiceWithOverrideTestCase*.)

For more information, see:

- Pulp #1724
- Pulp #1928
- Pulp Smash #127
- Pulp Smash #232

class `pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.`**BaseTestCase**(*methodName='runTest'*)
Provide common behaviour for the test cases in this module.

classmethod setUpClass()
Create a repository with a distributor, and populate it.

In addition, create several variables for use by the test methods.

test_first_publish()
Assert the first publish is a full publish.

class pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**ChangeRepoTestCase**(*methodName='runTes*
Publish a repository, change it, and publish it again.

classmethod setUpClass()
Publish a repository, change it, and publish it again.

test_second_publish_repomd()
Assert the two publishes produce identical `repomd.xml` files.

test_second_publish_tasks()
Assert the second publish's last task reports a no-op publish.

class pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**NoOpPublishMixin**
Provide tests for the no-op publish test cases in this module.

test_second_publish_repomd()
Assert the two publishes produce identical `repomd.xml` files.

test_second_publish_tasks()
Assert the second publish's last task reports a no-op publish.

class pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**PubTwiceTestCase**(*methodName='runTest'*)
Publish a repository twice.

classmethod setUpClass()
Publish a repository twice.

class pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**PubTwiceWithOverrideTestCase**(*metho*
Publish a repository twice, with an override config both times.

classmethod setUpClass()
Publish a repository twice, with an override config both times.

pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**get_repomd_xml_path**(*distributor_rel_url*)
Construct the path to a repository's `repomd.xml` file.

Parameters **distributor_rel_url** – A distributor's `relative_url` option.

Returns An string path to a `repomd.xml` file.

pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**setUpModule**()
Possibly skip the tests in this module. Create and sync an RPM repo.

Skip this module of tests if Pulp is older than version 2.9. (See Pulp #1724.) Then create an RPM repository with a feed and sync it. Test cases may copy data from this repository but should **not** change it.

pulp_smash.tests.pulp2.rpm.api_v2.test_no_op_publish.**tearDownModule**()
Delete the repository created by *setUpModule()*.

6.80 pulp_smash.tests.pulp2.rpm.api_v2.test_orphan_remove

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_orphan_remove* Test Pulp's handling of orphaned content units.

This module integrates tightly with Pulp Fixtures. Pulp Smash #134 and *Pulp Smash #459* describe specific tests that should be in this module.

class pulp_smash.tests.pulp2.rpm.api_v2.test_orphan_remove.**OrphansTestCase**(*methodName='runTest'*)
Establish that API calls related to orphans function correctly.

At a high level, this test case does the following:

1. Create an RPM repository and populate it with content. Delete the repository, thus leaving behind orphans.
2. Make several orphan-related API calls, and assert that each call has the desired effect.

check_one_orphan_deleted(*orphans_pre*, *orphans_post*, *orphan*)
Ensure that a specific orphan is well and truly deleted.

Parameters

- **orphans_pre** – The response to GET *pulp_smash.tests.pulp2.constants.ORPHANS_PATH* before the orphan was deleted.
- **orphans_post** – The response to GET *pulp_smash.tests.pulp2.constants.ORPHANS_PATH* after the orphan was deleted.
- **orphan** – A dict describing the orphan that was deleted.

Returns Nothing.

classmethod setUpClass()
Create orphans.

Create, sync and delete an RPM repository. Doing this creates orphans that the test methods can make use of.

test_00_orphans_available()
Assert that an expected number of orphans is present.

If the expected number of orphans is present, set a class variable indicating such. The following test methods can conditionally run or skip based on this variable. If this method fails, it may indicate that other repositories exist and have references to the same content units.

test_01_get_by_href()
Get an orphan by its href.

test_01_get_by_invalid_type()
Get orphans by content type. Specify a non-existent content type.

test_02_delete_by_href()
Delete an orphan by its href.

test_02_delete_by_type_and_id()
Delete an orphan by its ID and type.

This test exercises Pulp #1923.

test_03_delete_by_content_type()
Delete orphans by their content type.

test_04_delete_all()
Delete all orphans.

test_05_no_orphans_exist()
Assert no orphans exist.

6.81 *pulp_smash.tests.pulp2.rpm.api_v2.test_package_paths*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_package_paths* Tests that sync repositories whose packages are in varying locations.

The packages in a repository may be in the root of a repository, like so:

```
repository
- bear-4.1-1.noarch.rpm
- camel-0.1-1.noarch.rpm
- ...
- repodata
    - b83f17e552fa7a86f75811147251b4cc4f411eacfde5a187375d-primary.xml.gz
    - 5ec9512bc0461c579aebd3fe6d89133db5255cc1e15d529a20-other.sqlite.bz2
    - ...
```

However, the files in the *repodata/* directory specify the paths to each file included in a repository. Consequently, other layouts are possible, like so:

```
repository
- packages
|   - keep-going
|       - bear-4.1-1.noarch.rpm
|       - camel-0.1-1.noarch.rpm
|       - ...
- repodata
    - b83f17e552fa7a86f75811147251b4cc4f411eacfde5a187375d-primary.xml.gz
    - 5ec9512bc0461c579aebd3fe6d89133db5255cc1e15d529a20-other.sqlite.bz2
    - ...
```

The tests in this module verify that Pulp can correctly handle these situations.

class pulp_smash.tests.pulp2.rpm.api_v2.test_package_paths.**ReuseContentTestCase**(*methodName='run*

Sync two repositories with identical content but differing layouts.

If two repositories have some identical content, then Pulp should be able to re-use that content. This is true even if the content is placed in differing locations in the two repositories. Do the following:

1. Create a pair of repositories. Give the two repositories a download policy of either on demand or background. (One of these download policies must be used to ensure that the Pulp streamer is used.) Give the two repositories differing feeds, where the feeds reference a pair of repositories with some identical content and differing layouts.
2. Sync each of the repositories.
3. Publish each of the repositories.
4. Fetch an identical content unit from each of the two repositories.

This test targets Pulp #2354.

create_repo(*cfg*, *feed*, *download_policy*)

Create an RPM repository with the given feed and download policy.

Also, schedule the repository for deletion at the end of the current test. Return a detailed dict of information about the just-created repository.

test_all()

Sync two repositories w/identical content but differing layouts.

6.82 pulp_smash.tests.pulp2.rpm.api_v2.test_remove_unit

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_remove_unit* Tests for the remove_missing repository option.

class pulp_smash.tests.pulp2.rpm.api_v2.test_remove_unit.**RemoveCountTestCase**(*methodName='runTest*
Re-sync a child repository with the remove_missing enabled.

After removing an arbitrary number of units from the parent repository.

Do the following:

1. Create 1st repository with, feed, sync and publish it.
2. Create 2nd repository with feed pointed to 1st repository, and remove_missing enabled, sync.
3. Remove an arbitrary number of units from 1st repository, re-publish it.
4. Re-sync the 2nd repository. Assert that removed_count is equal to number of removed units.

Pulp issue #2616 caused after sync with remove_missing option the next publish is no-op.

test_all()
Re-sync a child repository with the remove_missing enabled.

class pulp_smash.tests.pulp2.rpm.api_v2.test_remove_unit.**RemoveMissingTestCase**(*methodName='runT*
Test the remove_missing repository option.

The test procedure is as follows:

1. Create a repository, populate it with content, and publish it.
2. Create several more repositories whose "feed" attribute references the repository created in the previous step. Sync each of these child repositories.
3. Remove a content unit from the parent repository and re-publish it.
4. Re-sync each of the child repositories.

The child repositories are designed to test the following issues:

- Pulp #1621
- Pulp #2503

classmethod setUpClass()
Initialize class-wide variables.

classmethod tearDownClass()
Delete all resources named by resources.

test_01_create_root_repo()
Create, sync and publish a repository.

The repositories created in later steps sync from this one.

test_02_create_immediate_child()
Create a child repository with the "immediate" download policy.

Sync the child repository, and verify it has the same contents as the root repository.

test_02_create_on_demand_child()
Create a child repository with the "on_demand" download policy.

Also, let the repository's "remove_missing" option be true. Then, sync the child repository, and verify it has the same contents as the root repository.

test_03_update_root_repo()
Remove a content unit from the root repository and republish it.

test_04_sync_immediate_child()
Sync the "immediate" repository.

Verify it has the same contents as the root repository.

test_04_sync_on_demand_child()
Sync the "on demand" repository.

Verify it has the same contents as the root repository.

6.83 pulp_smash.tests.pulp2.rpm.api_v2.test_repomd

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_repomd* Verify the *repomd.xml* file generated by a YUM distributor.

class pulp_smash.tests.pulp2.rpm.api_v2.test_repomd.**FastForwardIntegrityTestCase**(*methodName='ru*
Ensure fast-forward publishes use files referenced by `repomd.xml`.

When Pulp performs an incremental fast-forward publish, it should copy the original repository's `repodata/[...]-primary.xml` file to the new repository, and then modify it as needed.

According to Pulp #1088, Pulp does something different: it searches for all files named `repodata/[0-9a-zA-Z]*-primary.xml.*`, sorts them by mtime, copies the newest one to the new repository and modifies it. This behaviour typically works, because Pulp only creates one `[...]-primary.xml` file in a given repository. However, this behaviour is fragile, and it's especially likely to fail when third-party tools are used to supplement Pulp's functionality. What Pulp *should* do is to consult the old repository's `repomd.xml` file to find the `[...]-primary.xml` file.

Do the following:

1. Create a repository with a yum distributor, sync in some content, and publish it. Verify that `[...]-primary.xml` contains a certain phrase.
2. Create a second `[...]-primary.xml` file in the published repository, and replace the known phrase with a new phrase. Trigger a full publish, and verify that the known phrase is present, not the new phrase.
3. Create a second `[...]-primary.xml` file in the published repository, and replace the known phrase with a new phrase. Trigger an incremental publish, and verify that the known phrase is present, not the new phrase.

test_all()
Ensure fast-forward publishes use files referenced by repomd.xml.

class pulp_smash.tests.pulp2.rpm.api_v2.test_repomd.**RepoMDTestCase**(*methodName='runTest'*)
Tests to ensure `repomd.xml` can be created and is valid.

classmethod setUpClass()
Generate, fetch and parse a `repomd.xml` file.

Do the following:

1. Create an RPM repository with a YUM distributor and publish it.
2. Fetch the `repomd.xml` file from the distributor, and parse it.

test_data()
Assert the tree's "data" elements have correct "type" attributes.

test_tag()
Assert the XML tree's root element has the correct tag.

6.84 pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout* Tests for the published repository layout.

Starting with Pulp 2.12, YUM distributor must publish repositories using the layout where the packages are published to into `Packages/<first_letter>` where `<first_letter>` is the first letter of a given RPM package. For example, the `bear.rpm` package will be published into `Packages/b/bear.rpm`. For more information, see Pulp issue #1976.

Old versions of Pulp uses the old layout where all repository's packages were published on the root of the repository directory not inside the `Packages/<first_letter>` subdirectory.

class pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout.**RepositoryLayoutTestCase**(*method*
Test the YUM distributor publishing repository layout.

test_all()
Do not use the `packages_directory` option.

Create a distributor with default options, and use it to publish the repository. Verify packages end up in the current directory, relative to the published repository's root. (This same directory contains the `repodata` directory, and it may be changed by setting the distributor's `relative_url`.)

pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout.**get_file_hrefs**(*repomd_xml*)
Return the href of each file in a `repomd.xml` file.

Parameters **repomd_xml** – An `xml.etree.ElementTree` object representing the root of a `repomd.xml` document.

Returns An iterable of hrefs, with each href as a string.

pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout.**get_package_hrefs**(*primary_xml*)
Return the href of each package in a `primary.xml` file.

Parameters **primary_xml** – An `xml.etree.ElementTree` object representing the root of a `primary.xml` document.

Returns An iterable of hrefs, with each href as a string.

pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout.**get_parse_repodata_primary_xml**(*cfg*, *distributor*)
Fetch, decompress, parse and return a `repodata/...primary.xml.gz` file.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.
- **distributor** – Information about a distributor. It should be a dict containing at least `{'config': {'relative_url': ...}}`.

Returns An `xml.etree.ElementTree` object.

pulp_smash.tests.pulp2.rpm.api_v2.test_repository_layout.**get_parse_repodata_xml**(*server_config*, *distributor*, *file_path*)

Fetch, parse and return an XML file from a `repodata` directory.

Parameters

- **server_config** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp deployment being targeted.
- **distributor** – Information about a distributor. It should be a dict containing at least `{'config': {'relative_url': ...}}`.
- **file_path** – The path to an XML file, relative to the distributor's relative URL. For example: `repodata/repomd.xml`.

Returns The XML document, parsed as an `xml.etree.ElementTree` object.

6.85 pulp_smash.tests.pulp2.rpm.api_v2.test_repoview

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_repoview* Tests that exercise Pulp's repoview feature.

For more information, see:

- Pulp #189: "Repoview-like functionality for browsing repositories via the web interface"
- Yum Plugins → Yum Distributor → Optional Configuration Parameters

class pulp_smash.tests.pulp2.rpm.api_v2.test_repoview.**RepoviewTestCase**(*methodName='runTest'*)

Publish a repository with the repoview feature on and off.

Do the following:

1. Create an RPM repository, and add some content to it.
2. Publish the repository. Get `/pulp/repos/{rel_url}/`, and verify that no redirects occur.
3. Publish the repository with the `repoview` and `generate_sqlite` options set to true. Get `/pulp/repos/{rel_url}/`, and verify that a redirect to `/pulp/repos/{rel_url}/repoview/index.html` occurs.
4. Repeat step 2.

test_all()

Publish a repository with the repoview feature on and off.

6.86 pulp_smash.tests.pulp2.rpm.api_v2.test_republish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_republish* Test re-publish repository after unassociating content.

Following steps are executed in order to test correct functionality of repository created with valid feed.

1. Create repository foo with valid feed, run sync, add distributor to it and publish over http and https.
2. Pick a unit X and and assert it is accessible.

3. Remove unit X from repository foo and re-publish.
4. Assert unit X is not accessible.

class pulp_smash.tests.pulp2.rpm.api_v2.test_republish.**RepublishTestCase**(*methodName='runTest'*)
Test re-publish repository after unassociating content.

test_all()
Create one repository with feed, unassociate unit and re-publish.

Following steps are executed:

1. Create, sync and publish a repository.
2. Pick a content unit from the repository and verify it can be downloaded.
3. Remove the content unit from the repository, re-publish, and verify it can't be downloaded.

6.87 pulp_smash.tests.pulp2.rpm.api_v2.test_retain_old_count

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_retain_old_count* Test the retain_old_count feature.

When more than one version of an RPM is present in a repository and Pulp syncs from that repository, it must choose how many versions of that RPM to sync. By default, it syncs all versions of that RPM. The retain_old_count option lets one sync a limited number of outdated RPMs.

class pulp_smash.tests.pulp2.rpm.api_v2.test_retain_old_count.**RetainOldCountTestCase**(*methodNam*
Test the `retain_old_count` feature.

create_sync_repo(*retain_old_count*)
Create and sync a repository. Return detailed information about it.

Implement the logic described by the `test_retain_*` methods.

classmethod setUpClass()
Create, populate and publish a repository.

Ensure at least two versions of an RPM are present in the repository.

test_retain_one()
Give `retain_old_count` a value of one.

Create a repository whose feed references the repository created in *setUpClass()*, and whose `retain_old_count` option is one. Sync the repository, and assert that one old version of any duplicate RPMs were copied over.

test_retain_zero()
Give `retain_old_count` a value of zero.

Create a repository whose feed references the repository created by *setUpClass()*, and whose `retain_old_count` option is zero. Sync the repository, and assert that zero old versions of any duplicate RPMs were copied over.

6.88 pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor* Test the functionality of the RPM rsync distributor.

The RPM rsync distributor lets one publish content units to a directory via rsync+ssh. A typical usage of the RPM rsync distributor is as follows:

1. Create RPM repository with yum and RPM rsync distributors.
2. Upload some content units to the repository.
3. Publish the repository with the yum distributor.
4. Publish the repository with the RPM rsync distributor.

The RPM rsync distributor may not be used by itself. One cannot create an RPM repository with just an RPM rsync distributor; and one cannot publish a repository with the RPM rsync distributor without first publishing with a yum distributor.

For more information on the RPM rsync distributor, see Pulp #1759. As a quick reference, consider a repository with the following abbreviated distributor definitions:

```
[
    {
        'distributor_type_id': 'rpm_rsync_distributor',
        'config': {
            'remote': {'host': '192.168.100.32', 'root': '/home/myuser'},
            'remote_units_path': 'foo/bar/biz'  # default: 'content/units'
        }
    },
    {
        'distributor_type_id': 'yum_distributor',
        'config': {'relative_url': 'rel-url/'}
    }
]
```

Following a publish with the yum and rpm rsync distributors, respectively, files will be laid out as follows:

```
/home/myuser
- foo
|   - bar
|       - biz
|           - rpm
|               - 06
|               |   - ...
|               |       - dog-4.23-1.noarch.rpm
|               - 09
|               |   - ...
|                       - crow-0.8-1.noarch.rpm
- rel-url
    - bear-4.1-1.noarch.rpm -> ../foo/bar/biz/rpm/a9/.../bear-4.1-1.noarch...
    - camel-0.1-1.noarch.rpm -> ../foo/bar/biz/rpm/92/.../camel-0.1-1.noar...
```

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**AddUnitTestCase**(*methodName='runT*

Add a content unit to a repo in the middle of several publishes.

When executed, this test case does the following:

1. Create a yum repository with a yum and rsync distributor.
2. Add some content to the repository.
3. Publish the repository with its yum distributor.
4. Add additional content to the repository.

5. Publish the repository with its rsync distributor. This publish shouldn't distribute the new content unit, as the new content unit wasn't included in the most recent publish with the yum distributor.
6. Publish the repository with its yum distributor.
7. Publish the repository with its rsync distributor. This publish should distribute the new content unit, as the new content unit was included in the most recent publish with the yum distributor.

This test case targets:

- Pulp #2532
- Pulp Smash #526

test_all()
Add a content unit to a repo in the middle of several publishes.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**DeleteTestCase**(*methodName='runTes*
Use the `delete` RPM rsync distributor option.

Do the following:

1. Create a repository with a yum distributor and RPM rsync distributor. Add content units to the repository.
2. Publish with the yum distributor.
3. Publish with the RPM rsync distributor. Verify that the correct files are in the target directory.
4. Remove all files from the repository, and publish with the yum distributor.
5. Publish with the RPM rsync distributor, with `delete` set to true. Verify that all files are removed from the target directory.

This test targets Pulp #2221.

test_all()
Use the `delete` RPM rsync distributor option.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**ForceFullTestCase**(*methodName='ru*
Use the `force_full` RPM rsync distributor option.

Do the following:

1. Create a repository with a yum distributor and RPM rsync distributor. Add content units to the repository.
2. Publish with the yum distributor.
3. Publish with the RPM rsync distributor. Verify that the correct files are in the target directory.
4. Remove all files from the target directory. Publish again, and verify that:
   - The task for publishing has a result of "skipped."
   - No files are placed in the target directory. (This tests Pulp's fast-forward logic.)
5. Publish with the RPM rsync distributor, with `force_full` set to true. Verify that files are placed in the target directory. Skip this step if Pulp #2202 is not yet fixed.

Additionally, SELinux is temporarily set to "permissive" mode on the target system if Pulp #2199 is not yet fixed.

test_all()
Use the `force_full` RPM rsync distributor option.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**PublishBeforeYumDistTestCase**(*n*
Publish a repo with the rsync distributor before the yum distributor.

Do the following:

1. Create a repository with a yum distributor and rsync distributor.
2. Publish with the rpm rsync distributor. Verify that:
   - The publish has a result of "skipped."
   - No files are placed on the remote system.
3. Publish with the rpm rsync distributor again. Perform the same verification steps.

This test targets:

- Pulp #2187
- Pulp #2722

test_all()
Publish the rpm rsync distributor before the yum distributor.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**PublishTwiceTestCase**(*methodName*
Publish with a yum and rsync distributor twice.

Do the following when executed:

1. Create a yum repository with a yum and rsync distributor.
2. Add some content to the repository.
3. Publish with the yum and rsync distributor.
4. Publish with the yum and rsync distributor again.

The second publish with the rsync distributor should be a fast-forward publish, as no new units were added to the repository between the first and second publishes. Verify that the first rsync publish is not a fast-foward publish and that the second one is.

This test case targets Pulp #2666.

static get_num_processed(*publish_task*)
Return num_processed for unit query step in publish_task.

test_all()
Publish with a yum and rsync distributor twice.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**RemoteUnitsPathTestCase**(*methodN*
Exercise the remote_units_path option.

Do the following:

1. Create a repository with a yum distributor and RPM rsync distributor, and ensure that the latter distributor's remote_units_path option is set to a non-default value (not content/units). Add content units to the repository.
2. Publish with the yum distributor.
3. Publish with the RPM rsync distributor. Verify that files are placed in the correct directory.
4. Publish with the RPM rsync distributor, with remote_units_path passed as publish option. Verify that files are placed in this directory.

test_all()
Exercise the remote_units_path option.

class pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**VerifyOptionsTestCase**(*methodNam*
Test Pulp's verification of RPM rsync distributor configuration options.

Do the following:

- Repeatedly attempt to create an RPM repository with an importer and pair of distributors. Each time, pass an invalid option, or omit a required option.
- Create an RPM repository with an importer and pair of distributors. Pass valid configuration options. This demonstrates that creation failures are due to Pulp's validation logic, not some other factor.

classmethod **setUpClass**()
Create a value for the rsync distrib's `remote` config section.

Using the same config for each of the test methods allows the test methods to behave more similarly.

tearDown()
Verify that the `remote` config section hasn't changed.

test_predistributor_id()
Pass a bogus ID as the `predistributor_id` config option.

test_remote_units_path()
Pass an absolute path to the `remote_units_path` config option.

test_required_options()
Omit each of the required RPM rsync distributor config options.

test_root()
Pass a relative path to the `root` configuration option.

test_success()
Successfully create an RPM repo with importers and distributors.

pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**setUpModule**()
Conditionally skip the tests in this module.

Skip the tests in this module if:

- The RPM plugin is not installed on the target Pulp server.
- Pulp #1759 is not implemented on the target Pulp server.

pulp_smash.tests.pulp2.rpm.api_v2.test_rsync_distributor.**tearDownModule**()
Delete orphan content units.

6.89 pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish* Test the API's schedule functionality for repository publication.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish.**CreateFailureTestCase**(*methodName*
Establish that schedules are not created in documented scenarios.

classmethod **setUpClass**()
Create several schedules.

Each schedule is created to test a different failure scenario.

test_status_code()
Assert that each response has the expected HTTP status code.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish.**CreateSuccessTestCase**(*methodName*
Establish that we can create a schedule to publish the repository.

classmethod **setUpClass**()
Create a schedule to publish the repository.

Do the following:

1. Create a repository with a valid feed
2. Sync it
3. Schedule publish to run every 30 seconds

test_is_enabled()
Check if sync is enabled.

test_status_code()
Assert the response has an HTTP 201 status code.

test_total_run_count()
Check that `total_run_count` is sane.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish.**ReadUpdateDeleteTestCase**(*methodN*
Establish that we can read, update and delete schedules.

This test case assumes the assertions in *CreateSuccessTestCase* hold true.

classmethod setUpClass()
Create three schedules and read, update and delete them.

test_read_many()
Assert the "read_many" response body contains all schedules.

test_read_one()
Assert the "read_one" response contains the correct attributes.

test_status_code()
Assert each response has a correct HTTP status code.

test_update()
Assert the "update" response body has the correct attributes.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_publish.**ScheduledPublishTestCase**(*methodN*
Establish that publish runs according to the specified schedule.

This test case assumes the assertions in *CreateSuccessTestCase* and *ReadUpdateDeleteTestCase* hold true.

classmethod setUpClass()
Create a schedule to publish a repo, verify the `total_run_count`.

Do the following:

1. Create a repository with a valid feed
2. Sync it
3. Schedule publish to run every 2 minutes
4. Wait for 130 seconds and read the schedule to get the number of "publish" runs

test_no_failure()
Make sure any failure ever happened.

test_total_run_count()
Check for the expected total run count.

6.90 pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync* Test the API's schedule functionality for repository syncronization.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync.**CreateFailureTestCase**(*methodName='ru*
Establish that schedules are not created in documented scenarios.

classmethod setUpClass()
Intentionally fail at creating several sync schedules for a repo.

Each schedule tests a different failure scenario.

test_status_code()
Assert that each response has the expected HTTP status code.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync.**CreateRepoMixin**
Provide a method for creating a repository.

classmethod create_repo()
Create a semi-random RPM repository with a valid RPM feed URL.

Add this repository's href to `cls.resources`. Return a two-tuple of `(href, importer_type_id)`. This method requires a server config, `cls.cfg`, and a set, `cls.resources`.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync.**CreateSuccessTestCase**(*methodName='ru*
Establish that we can create a schedule to sync the repository.

classmethod setUpClass()
Create an RPM repo with a valid feed, create a schedule to sync it.

Do the following:

1. Create a repository with a valid feed
2. Schedule sync to run every 30 seconds

test_enabled()
Verify the `enabled` attribute in the response.

test_status_code()
Assert the response has an HTTP 201 status code.

test_total_run_count()
Verify the `total_run_count` attribute in the response.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync.**ReadUpdateDeleteTestCase**(*methodName*
Establish that we can read, update and delete schedules.

This test case assumes the assertions in *`CreateSuccessTestCase`* hold true.

classmethod setUpClass()
Create three schedules and read, update and delete them.

test_delete()
Assert the "delete" response body is null.

test_read_many()
Assert the "read_many" response body contains all schedule hrefs.

test_read_one()
Assert the "read_one" response contains the correct attributes.

test_status_code()
Assert each response has a correct HTTP status code.

test_update()
Assert the "update" response body has the correct attributes.

class pulp_smash.tests.pulp2.rpm.api_v2.test_schedule_sync.**ScheduledSyncTestCase**(*methodName='ru*
Establish that sync runs according to the specified schedule.

This test case assumes the assertions in *CreateSuccessTestCase* and *ReadUpdateDeleteTestCase* hold true.

classmethod setUpClass()
Create a schedule to sync the repo, verify the total_run_count.

Do the following:

1. Create a repository with a valid feed
2. Schedule sync to run every 30 seconds
3. Wait for 40 seconds and read the schedule to get the number of "sync" runs.

test_consecutive_failures()
Assert the sync encountered no consecutive failures.

6.91 *pulp_smash.tests.pulp2.rpm.api_v2.test_search*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_search* Test Pulp's Searching facilities.

The test cases in this module searches for content units by their type. See:

- Pulp Smash #133 asks for tests showing that it's possible to search for content units by their type.
- The Search for Units API documentation is relevant.

class pulp_smash.tests.pulp2.rpm.api_v2.test_search.**BaseSearchTestCase**(*methodName='runTest'*)
Provide common functionality for the other test cases in this module.

static get_feed_url()
Return a repository feed URL. Used by *setUpClass()*.

All child classes should override this method.

get_unit_by_filename(*units*, *filename*)
Return the unit with the given filename.

Test methods in child classes may find this method to be of use.

Parameters

- **units** – An iterable of SRPM content units.
- **filename** – The filename of one of the content units.

Returns The content unit with the given filename.

Raises An assertion error if more or less than one matching unit is found.

classmethod setUpClass()
Create and sync a repository.

class pulp_smash.tests.pulp2.rpm.api_v2.test_search.**SearchForRpmTestCase**(*methodName='runTest'*)
Search for rpm content units in several different ways.

static get_feed_url()
Return an RPM repository feed URL.

test_include_repos()
Search for "rpm" units, and include repos in the results.

Perform the following searches. Assert a unit with filename *pulp_smash.constants.RPM* is in the search results, and assert it belongs to the repository created in *BaseSearchTestCase.setUpClass()*.

GET	`{'include_repos': True}` (urlencoded)
POST	`{'criteria': {}, 'include_repos': True}`

test_search_for_all()
Search for all "rpm" units.

Perform the following searches. Assert a unit with filename *pulp_smash.constants.RPM* is in the search results.

GET	n/a
POST	`{'criteria': {}}`

class pulp_smash.tests.pulp2.rpm.api_v2.test_search.**SearchForSrpmTestCase**(*methodName='runTest'*)
Search for srpm content units in several different ways.

static get_feed_url()
Return an RPM repository feed URL.

test_include_repos()
Search for all "srpm" units, and include repos in the results.

Perform the following searches. Assert a unit with filename *pulp_smash.constants.SRPM* is in the search results, and assert it belongs to the repository created in *BaseSearchTestCase.setUpClass()*.

GET	`{'include_repos': True}` (urlencoded)
POST	`{'criteria': {}, 'include_repos': True}`

test_search_for_all()
Search for all "srpm" units.

Perform the following searches. Assert a unit with filename *pulp_smash.constants.SRPM* is in the search results.

GET	n/a
POST	`{'criteria': {}}`

6.92 pulp_smash.tests.pulp2.rpm.api_v2.test_service_resiliency

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_service_resiliency* Test the resiliency of Pulp's services.

Pulp is designed to be resilient. It should be possible for a service to disappear and reappear, with the only consequence being slower processing of tasks. This module has tests for Pulp's resilience in the face of such issues.

class pulp_smash.tests.pulp2.rpm.api_v2.test_service_resiliency.**MissingWorkersTestCase**(*methodN*
Test that Pulp deals well with missing workers.

When executed, this test case will do the following:

1. Ensure there is only one Pulp worker.
2. Create a repository. Let its feed reference a large repository. (For example, EPEL.)
3. Start a sync. Immediately restart the `pulp_workers` service. (It's important that `pulp_workers` be restarted, not started and stopped. For details, see Pulp #2835.) This should cause the first sync to abort.
4. Update the repository. Let its feed reference a small repository. (For example, *pulp_smash.constants.RPM_UNSIGNED_FEED_URL*.)
5. Start a sync. Verify that it completes. If Pulp #2835 still affects Pulp, then the worker will be broken, and the sync will never start.

setUp()
Ensure there is only one Pulp worker.

tearDown()
Reset the number of Pul pworkers, and reset Pulp.

Reset Pulp because *test_all()* may break Pulp.

test_all()
Test that Pulp deals well with missing workers.

class pulp_smash.tests.pulp2.rpm.api_v2.test_service_resiliency.**TaskDispatchTestCase**(*methodNam*
Test whether `httpd` dispatches a task while the broker is down.

test_all()
Test whether `httpd` dispatches a task while the broker is down.

This test targets the following issues:

- Pulp Smash #650
- Pulp #2770

This test does the following:

1. Create a repository.
2. Stop the AMQP broker. (Also, schedule it to be re-started later!)
3. Sync the repository, ignore any errors that are returned when doing so, and assert that no tasks are left in the `waiting` state.

6.93 pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies*
Tests for repository signature checks when copying packages.

This module mimics *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*, except that packages are primarily copied into repositories, instead of being uploaded.

Note: Pulp's signature checking logic is subtle. Please read *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**AllowAnyKeyTestCas**
Use an importer that allows unsigned packages and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": []}
```

test_signed_packages()
Copy signed packages into a repository.

Assert packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert packages are copied in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**AllowInvalidKeyTes**
Use an importer that allows unsigned packages and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["invalid key id"]}
```

test_signed_packages()
Copy signed packages into a repository.

Assert no packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert packages are copied in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**AllowValidKeyTestC**
Use an importer that allows unsigned packages and has a valid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["valid key id"]}
```

Note: Pulp's signature checking logic is subtle. Please read *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*.

---

test_signed_packages()
Copy signed packages into a repository.

Assert packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert packages are copied in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**RequireAnyKeyTestC**
Use an importer that requires signatures and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": []}
```

test_signed_packages()
Copy signed packages into a repository.

Assert packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert no packages are copied in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**RequireInvalidKeyT**
Use an importer that requires signatures and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": ["invalid key id"]}
```

test_signed_packages()
Copy signed packages into a repository.

Assert no packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert no packages are copied in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**RequireValidKeyTes**
Use an importer that requires signatures and has a valid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": ["valid key id"]}
```

test_signed_packages()
Copy signed packages into a repository.

Assert packages are copied in.

test_unsigned_packages()
Copy unsigned packages into a repository.

Assert no packages are copied in.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**setUpModule**()
Conditionally skip tests. Create repositories with fixture data.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_copies.**tearDownModule**()
Delete repositories with fixture data.

6.94 pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs*
Tests for repository signature checks when syncing packages.

This module mimics *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*, except that packages are synced in to Pulp instead of being uploaded.

Note: Pulp's signature checking logic is subtle. Please read *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**AllowAnyKeyTestCase**
Use an importer that allows unsigned packages and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": []}
```

test_signed_drpm()
Sync signed DRPMs into the repository.

Assert packages are synced in.

test_signed_rpm()
Sync signed RPMs into the repository.

Assert packages are synced in.

test_signed_srpm()
Sync signed SRPMs into the repository.

Assert packages are synced in.

test_unsigned_drpms()
Import unsigned DRPMs into a repository.

Verify packages are synced.

test_unsigned_rpms()
Import unsigned RPMs into a repository.

Verify packages are synced.

test_unsigned_srpms()
Import unsigned SRPMs into a repository.

Verify packages are synced.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**AllowInvalidKeyTest**
Use an importer that allows unsigned packages and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["invalid key id"]}
```

test_signed_packages()
Import signed RPMs, DRPMs and SRPMs into repositories.

Assert no packages are synced.

test_unsigned_drpms()
Import unsigned DRPMs into a repository.

Assert packages are synced.

test_unsigned_rpms()
Import unsigned RPMs into a repository.

Assert packages are synced.

test_unsigned_srpms()
Import unsigned SRPMs into a repository.

Assert packages are synced.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**AllowValidKeyTestCa**
Use an importer that allows unsigned packages and has a valid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["valid key id"]}
```

Note: Pulp's signature checking logic is subtle. Please read *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*.

test_signed_drpm()
Sync signed DRPMs into the repository.

Assert packages are synced in.

test_signed_rpm()
Sync signed RPMs into the repository.

Assert packages are synced in.

test_signed_srpm()
Sync signed SRPMs into the repository.

Assert packages are synced in.

test_unsigned_drpms()
Import unsigned DRPMs into a repository.

Assert packages are synced.

test_unsigned_rpms()
Import unsigned RPMs into a repository.

Assert packages are synced.

test_unsigned_srpms()
Import unsigned SRPMs into a repository.

Assert packages are synced.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**RequireAnyKeyTestCa**
Use an importer that requires signatures and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": []}
```

test_signed_drpm()
Sync signed DRPMs into the repository.

Assert packages are synced in.

test_signed_rpm()
Sync signed RPMs into the repository.

Assert packages are synced in.

test_signed_srpm()
Sync signed SRPMs into the repository.

Assert packages are synced in.

test_unsigned_packages()
Sync unsigned RPMs, DRPMs and SRPMS into repositories.

Assert no packages are synced in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**RequireInvalidKeyTe**
Use an importer that requires signatures and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": ["invalid key id"]}
```

test_packages()
Sync signed and unsigned RPMs, DRPMs and SRPMs into repositories.

Assert no packages are synced in.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**RequireValidKeyTest**
Use an importer that requires signatures and has a valid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": ["valid key id"]}
```

test_signed_drpm()
Sync signed DRPMs into the repository.

Assert packages are synced in.

test_signed_rpm()
Sync signed RPMs into the repository.

Assert packages are synced in.

test_signed_srpm()
Sync signed SRPMs into the repository.

Assert packages are synced in.

test_unsigned_packages()
Sync unsigned RPMs, DRPMs and SRPMS into repositories.

Assert no packages are synced in.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**setUpModule**()
Conditionally skip tests.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_syncs.**tearDownModule**()
Delete orphan content units.

6.95 *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads*
Tests for repository importer signature checks.

As of Pulp 2.10, it's possible to configure an RPM repository importer to perform checks on all synced-in and uploaded packages. Two new importer options are available:

require_signature A boolean. If true, imported packages must be signed with a key listed in `allowed_keys`.

allowed_keys A list of 32-bit key IDs, as hex characters. (e.g. `["deadbeef"]`) An empty list is treated as the list of all possible key IDs.

Beware that if a package has a signature, its signature *must* be listed in `allowed_keys`, even when `require_signature` is false. The only importer configuration that allows all packages is `{'require_signature': False, 'allowed_keys': []}`.

To test this feature, importers with at least the following options should be created:

```
{'require_signature': False, 'allowed_keys': ['invalid key id']}
{'require_signature': False, 'allowed_keys': ['valid key id']}
{'require_signature': False, 'allowed_keys': []}
{'require_signature': True, 'allowed_keys': ['invalid key id']}
{'require_signature': True, 'allowed_keys': ['valid key id']}
{'require_signature': True, 'allowed_keys': []}
```

In addition, at least the following types of packages should be imported:

```
* Signed DRPMs
* Signed RPMs
* Signed SRPMs
* Unsigned DRPMs
* Unsigned RPMs
* Unsigned SRPMs
```

Finally, importer options may be changed in some circumstances, and Pulp should gracefully handle those changes.

For more information, see Pulp #1991 and Pulp Smash #347.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**AllowAnyKeyTestCa**
Use an importer that allows unsigned packages and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": []}
```

classmethod setUpClass()
Create a repository with an importer.

test_all_packages()
Import signed and unsigned DRPM, RPM & SRPM packages into the repo.

Verify that each import succeeds.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**AllowInvalidKeyTe**
Use an importer that allows unsigned packages and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["invalid key id"]}
```

classmethod setUpClass()
Create a repository with an importer.

test_signed_packages()
Import signed DRPM, RPM and SRPM packages into the repository.

Verify that each import fails.

test_unsigned_packages()
Import unsigned DRPM, RPM and SRPM packages into the repository.

Verify that each import succeeds.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**AllowValidKeyTest**
Use an importer that allows unsigned packages and has a valid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": false, "allowed_keys": ["valid key id"]}
```

classmethod setUpClass()
Create a repository with an importer.

test_all_packages()
Import signed and unsigned DRPM, RPM & SRPM packages into the repo.

Verify that each import succeeds.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**KeyLengthTestCase**
Verify pulp rejects key IDs that are not 32-bits long.

An OpenPGP-compatible key ID (key fingerprint) is traditionally a 32-bit value. Newer OpenPGP key handling software allows for longer key IDs, and this is recommended, as it's extremely easy to find colliding key IDs.[1] However, Pulp allows only the short key IDs.

test_key_ids()
Create importers with key IDs shorter and longer than 32 bits.

Pulp should prevent the importers from being created.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**RequireAnyKeyTest**
Use an importer that requires signatures and has no key IDs.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": []}
```

classmethod setUpClass()
Create a repository with an importer.

test_signed_packages()
Import signed DRPM, RPM and SRPM packages into the repo.

Verify that each import succeeds.

test_unsigned_packages()
Import unsigned DRPM, RPM and SRPM packages into the repo.

Verify that each import fails.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**RequireInvalidKey**
Use an importer that requires signatures and has an invalid key ID.

The importer should have the following pseudocode configuration:

```
{"require_signature": true, "allowed_keys": ["invalid key id"]}
```

classmethod setUpClass()
Create a repository with an importer.

test_all_packages()
Import signed and unsigned DRPM, RPM & SRPM packages into the repo.

Verify that each import fails.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**RequireValidKeyTe**
Use an importer that requires signatures and has a valid key ID.

The importer should have the following pseudocode configuration:

[1] https://evil32.com/

```
{"require_signature": true, "allowed_keys": ["valid key id"]}
```

classmethod **setUpClass**()
Create a repository with an importer.

test_signed_packages()
Import signed DRPM, RPM and SRPM packages into the repository.

Verify that each import succeeds.

test_unsigned_packages()
Import unsigned DRPM, RPM and SRPM packages into the repository.

Verify that each import fails.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**setUpModule**()
Conditionally skip tests. Cache packages to be uploaded to repos.

Skip the tests in this module if:

- The RPM plugin is unsupported.
- Pulp #1991 is untestable for the version of Pulp under test.

pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_checked_for_uploads.**tearDownModule**()
Delete the cached set of packages to be uploaded to repos.

6.96 pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_saved_for_packages

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_saved_for_packages*
Tests for how well Pulp can deal with package signatures.

When a package is added to an RPM repository, its signature should be stored by Pulp. This is true regardless of whether the package is an RPM, DRPM or SRPM.

Tests for this feature include the following:

- Upload signed and unsigned packages to RPM repositories. (See *UploadPackageTestCase*.)
- Synchronize signed and unsigned packages to RPM repositories. (See *SyncPackageTestCase*.)

For more information, see:

- Pulp #1156

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_saved_for_packages.**SyncPackageTestCas**
Test if Pulp saves signatures from synced-in packages.

test_signed_drpm()
Assert signature is stored for signed drpm during sync.

test_signed_rpm()
Assert signature is stored for signed rpm during sync.

test_signed_srpm()
Assert signature is stored for signed srpm during sync.

test_unsigned_drpm()
Assert no signature is stored for unsigned drpm during sync.

test_unsigned_rpm()
Assert no signature is stored for unsigned rpm during sync.

test_unsigned_srpm()
Assert no signature is stored for unsigned srpm during sync.

class pulp_smash.tests.pulp2.rpm.api_v2.test_signatures_saved_for_packages.**UploadPackageTestC**
Test if Pulp saves signatures from uploaded packages.

test_signed_drpm()
Import a signed DRPM into Pulp. Verify its signature.

test_signed_rpm()
Import a signed RPM into Pulp. Verify its signature.

test_signed_srpm()
Import a signed SRPM into Pulp. Verify its signature.

test_unsigned_drpm()
Import an unsigned DRPM into Pulp. Verify it has no signature.

test_unsigned_rpm()
Import an unsigned RPM into Pulp. Veriy it has no signature.

test_unsigned_srpm()
Import an unsigned SRPM into Pulp. Verify it has no signature.

6.97 pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish* Tests that sync and publish RPM repositories.

For information on repository sync and publish operations, see Synchronization and Publication.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**ChangeFeedTestCase**(*methodName='runTest*
Sync a repository, change its feed, and sync it again.

Specifically, the test case procedure is as follows:

1. Create three repositories — call them A, B and C.
2. Populate repository A and B with identical content, and publish them.
3. Set C's feed to repository A. Sync and publish repository C.
4. Set C's feed to repository B. Sync and publish repository C.
5. Download an RPM from repository C.

The entire procedure should succeed. This test case targets Pulp #1922.

create_sync_publish_repo(*body*)
Create, sync and publish a repository.

Also, schedule the repository for deletion.

Parameters **body** – A dict of information to use when creating the repository.

Returns A detailed dict of information about the repository.

get_feed(*repo*)
Build the feed to an RPM repository's distributor.

test_all()
Sync a repository, change its feed, and sync it again.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**ErrorReportTestCase**(*methodName='runTes*
Test whether an error report contains sufficient information.

test_all()
Test whether an error report contains sufficient information.

Do the following:

1. Create and sync a repository using an invalid feed URL.
2. Get a reference to the task containing error information.
3. Assert that:
   - The error description is sufficiently verbose. See Pulp #1376 and Pulp Smash #525.
   - The traceback is non-null. See Pulp #1455.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**NonExistentRepoTestCase**(*methodName='*
Perform actions on non-existent repositories.

This test targets Pulp Smash #157.

setUp()
Set variables used by each test case.

test_publish()
Publish a non-existent repository.

test_sync()
Sync a non-existent repository.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncDrpmRepoTestCase**(*methodName='runT*
Test one can create and sync an RPM repository with an DRPM feed.

static get_feed_url()
Return an DRPM repository feed URL.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncInParallelTestCase**(*methodName='ru*
Sync several repositories in parallel.

test_all()
Sync several repositories in parallel.

Specifically, do the following:

1. Create several repositories. Ensure each repository has an importer whose feed references a repository containing one or more errata.
2. Sync each repository. Assert each sync completed successfully.
3. Get a summary of information about each repository, and assert the repo has an appropriate number of errata.

Pulp #2721 describes how a race condition can occur when multiple repos with identical errata are synced at the same time. This test case attempts to trigger that race condition.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncInvalidMetadataTestCase**(*methodNa*
Sync various repositories with invalid metadata.

When a repository with invalid metadata is encountered, Pulp should gracefully fail. This test case targets Pulp #1287.

do_test(*feed_url*)
Implement the logic described by each of the `test*` methods.

classmethod tearDownClass()
Delete orphan content units.

test_incomplete_filelists()
Sync a repository with an incomplete `filelists.xml` file.

test_incomplete_other()
Sync a repository with an incomplete `other.xml` file.

test_missing_filelists()
Sync a repository that's missing its `filelists.xml` file.

test_missing_other()
Sync a repository that's missing its `other.xml` file.

test_missing_primary()
Sync a repository that's missing its `primary.xml` file.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncRepoBaseTestCase**(*methodName='runT*
A parent class for repository syncronization test cases.

get_feed_url() should be overridden by concrete child classes. This method's response is used when setting the repository's importer feed URL.

static get_feed_url()
Return an RPM repository feed URL. Should be overridden.

Raises `NotImplementedError` if not overridden by a child class.

classmethod setUpClass()
Create an RPM repository with a valid feed and sync it.

test_start_sync_code()
Assert the call to sync a repository returns an HTTP 202.

test_task_progress_report()
Assert no task's progress report contains error details.

Other assertions about the final state of each task are handled by the client's response handler. (For more information, see the source of *pulp_smash.api.safe_handler()*.)

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncRpmRepoTestCase**(*methodName='runTes*
Test one can create and sync an RPM repository with an RPM feed.

static get_feed_url()
Return an RPM repository feed URL.

test_no_change_in_second_sync()
Verify that syncing a second time has no changes.

If the repository have not changed then Pulp must state that anything was changed when doing a second sync.

test_unit_count_on_repo()
Verify that the sync added the correct number of units to the repo.

Read the repository and examine its `content_unit_counts` attribute. Compare these attributes to metadata from the remote repository. Expected values are currently hard-coded into this test.

class pulp_smash.tests.pulp2.rpm.api_v2.test_sync_publish.**SyncSrpmRepoTestCase**(*methodName='runT*
Test one can create and sync an RPM repository with an SRPM feed.

static get_feed_url()
Return an SRPM repository feed URL.

6.98 *pulp_smash.tests.pulp2.rpm.api_v2.test_tasks*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_tasks* Verify that operations can be performed over Tasks.

For information on tasks operations, see REST API Task Management and Admin Client Tasks.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_tasks.`**`TasksOperationsTestCase`**(*methodName='runTest'*)

Perform different operations over tasks.

This test targets the following issues:

- Pulp Smash #108
- Pulp Smash #142
- Pulp issue #1418
- Pulp issue #1483
- Pulp issue #1664

`test_all`()

Perform different operation over tasks.

Do the following:

1. Create, sync and publish a repository.
2. List current tasks.
3. Search tasks.
4. Polling a specific task progress.
5. Delete finished tasks.
6. Purge tasks.

6.99 *pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate* Test Unassociating Content Units from a Repository for RPM.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate.`**`RemoveUnitsTestCase`**(*methodName='runTest*

Remove units of various types from a synced repository.

`do_remove_unit`(*type_id*, *repo*)

Remove a content unit from the repository.

Do the following:

1. Note the repository's `last_unit_removed` field.
2. Sleep for at least one second.
3. Remove a unit of type `type_id` from the repository.
4. Note the repository's `last_unit_removed` field.

When the first unit is removed, assert that `last_unit_removed` changes from null to a non-null value. When each subsequent unit is removed, assert that `last_unit_removed` increments.

do_test(*feed*, *type_ids*)
Remove units from a repo and make assertions about it.

Do the following:

1. Create and sync a repository with the given `feed`.
2. For each type ID in `type_ids`, remove a content unit of that type from the repository. See *do_remove_unit()*.
3. Assert the correct units are still in the repository. The repository should have all the units that were originally synced into the repository, minus those that have been removed.
4. Remove a non-existent unit from the repository. Assert that the `last_unit_removed` timestamp was not updated.

get_repo_last_unit_removed(*repo*)
Get the repository's `last_unit_removed` attribute.

setUp()
Set variables used by each test case.

test_drpm()
Un-associate units from a DRPM repo. See *do_test()*.

test_rpm()
Un-associate units from a RPM repo. See *do_test()*.

test_srpm()
Un-associate units from a SRPM repo. See *do_test()*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate.**RepublishTestCase**(*methodName='runTest'*)
Repeatedly publish a repository, with different content each time.

Specifically, do the following:

1. Create a repository.
2. Add a content unit to the repository. Publish the repository.
3. Unassociate the content unit and repository. Publish the repository.

Verify that:

- The `last_unit_added`, `last_unit_removed` and `last_publish` timestamps are correct.
- The content unit in question is only available when associated with the repository.

get_repo()
Get detailed information about the repository.

classmethod setUpClass()
Create a repository.

classmethod tearDownClass()
Remove the created repository and any orphans.

test_01_add_unit()
Add a content unit to the repository. Publish the repository.

test_02_find_unit()
Search for the content unit. Assert it is available.

test_03_unassociate_unit()
Unassociate the unit from the repository. Publish the repository.

test_04_find_unit()
Search for the content unit. Assert it isn't available.

class pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate.**SelectiveAssociateTestCase**(*methodName*
Ensure Pulp only associate needed content.

Test steps:

1. Create and sync an RPM repository.
2. Unassociate some RPMs from the repository created on the previous step.
3. Sync the repository again and check if only the missing units were associated.

See Pulp #2457

test_all()
Check if Pulp only associate missing repo content.

pulp_smash.tests.pulp2.rpm.api_v2.test_unassociate.**setUpModule**()
Maybe skip this module of tests.

6.100 *pulp_smash.tests.pulp2.rpm.api_v2.test_unavailable_checksum*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_unavailable_checksum*
Tests that publish repositories with unavailable checksum types.

class pulp_smash.tests.pulp2.rpm.api_v2.test_unavailable_checksum.**UnavailableChecksumTestCase**
Publish a lazily-synced repository with unavailable checksum types.

do_test(*feed*, *type_id*)
Publish a repository with an invalid checksum.

Do the following:

1. Create and sync repository of type `type_id` and with the given feed. Ensure the repository's importer has a deferred/lazy download policy.
2. Determine which checksum type the units in the repository use. (As of this writing, Pulp can handle md5, sha1 and sha256 checksum types.)
3. Publish the repository with checksum types different from what the units in the repository use. Assert the publish fails.

This test targets Pulp Smash #287.

test_drpm()
Publish a DRPM repo. See *do_test()*.

test_rpm()
Publish a RPM repo. See *do_test()*.

test_srpm()
Publish a SRPM repo. See *do_test()*.

class pulp_smash.tests.pulp2.rpm.api_v2.test_unavailable_checksum.**UpdatedChecksumTestCase**(*met*
Ensure that the updated `checksum_type` is present on the repo metadata.

Do the following:

1. Create, sync and publish a repository with checksum type "sha256".
2. After that several XML files - part of repodata will be inspected.

primary.xml Assert that the given checksum type is the only one present.

filelists.xml and other.xml Assert that the length of `pkgid` is according to the given checksum type.

prestodelta.xml Assert that the given checksum type is the only one present.

For `sha256` the length of `pkgid` should be 64 hex digits, and for `sha1` the length should be 40 hex digits. For DRPM repositories the `prestodelta.xml` will be inspected as well.

3. Update the checksum type to "sha1", and use option `force_full` set as True. `force_full` will force a complete re-publish to happen. Re-publish the repository.
4. All aforementioned assertions performed on the step 2 will be executed again to assure that new checksum type was updating properly.

This test targets Pulp Smash #286.

do_test(*feed*)
Verify `checksum_type` is updated on the repo metadata.

test_drpm()
Verify updated checksum in a DRPM repo. See *do_test()*.

test_rpm()
Verify update checksum in a RPM repo. See *do_test()*.

test_srpm()
Verify updated checksum in a SRPM repo. See *do_test()*.

verify_filelists_xml(*cfg*, *distributor*, *checksum_type*)
Verify a published repo's filelists.xml uses the given checksum.

verify_other_xml(*cfg*, *distributor*, *checksum_type*)
Verify a published repo's other.xml uses the given checksum.

verify_pkgid_len(*pkgid_len*, *checksum_type*)
Perform assertion based on the given checksum type.

Based on the checksum type provided different assertions will be performed over the length of `pkgid`.

verify_presto_delta_xml(*cfg*, *distributor*, *checksum_type*)
Verify a published repo's prestodelta.xml uses given checksum.

verify_primary_xml(*cfg*, *distributor*, *checksum_type*)
Verify a published repo's primary.xml uses the given checksum.

6.101 *pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo* Test the `updateinfo.xml` files published by Pulp's yum distributor.

The purpose of an `updateinfo.xml` file is to document the errata (i.e. maintenance patches) provided by a repository. For an overview of the structure and contents of such a file, see openSUSE:Standards Rpm Metadata UpdateInfo. Beware that yum and dnf do not adhere to the openSUSE standards. We link to their standard anyway because it's easier to understand than yum or dnf's source code. (!)

One discrepancy is in the schema for the `<pkglist>` element. According to the yum source code, it has the following structure:

```
<!ELEMENT pkglist (collection+)>
<!ELEMENT collection (name?, package+)>
    <!ATTLIST collection short CDATA #IMPLIED>
    <!ATTLIST collection name CDATA #IMPLIED>
<!ELEMENT name (#PCDATA)>
```

Here's a concrete example of what that might look like:

```
<pkglist>
    <collection name="..." short="...">
        <name>...</name>
        <package>...</package>
        <package>...</package>
    </collection>
</pkglist>
```

yum (and, therefore, dnf) allows a `<collection>` element to have "name" and "short" attributes. openSUSE does not.

class pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo.**CleanUpTestCase**(*methodName='runTest'*)

Test whether old `updateinfo.xml` files are cleaned up.

Do the following:

1. Create, populate and publish a repository. Verify that an `updateinfo.xml` file is present and can be downloaded.
2. Add an additional content unit to the repository, and publish it again. Verify that the `updateinfo.xml` file created by the first publish is no longer available, and that a new `updateinfo.xml` file is available.

This procedure targets Pulp #2096. Note that the second publish must be an incremental publish.

get_updateinfo_xml_href()

Return the path to the `updateinfo.xml` file.

classmethod setUpClass()

Create and sync a repository.

classmethod tearDownClass()

Remove the created repository and any orphans.

test_01_first_publish()

Populate and publish the repository.

test_02_second_publish()

Add an additional content unit and publish the repository again.

class pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo.**PkglistsTestCase**(*methodName='runTest'*)

Sync a repository whose updateinfo file has multiple pkglist sections.

This test case targets Pulp #2227.

test_all()

Sync a repo whose updateinfo file has multiple pkglist sections.

Specifically, do the following:

1. Create, sync and publish an RPM repository whose feed is set to *pulp_smash.constants.RPM_PKGLISTS_UPDATEINFO_FEED_URL*.
2. Fetch and parse the published repository's `updateinfo.xml` file.

Verify that the `updateinfo.xml` file has three packages whose `<filename>` elements have the following text:

- penguin-0.9.1-1.noarch.rpm
- shark-0.1-1.noarch.rpm
- walrus-5.21-1.noarch.rpm

Note that Pulp is free to change the structure of a source repository at will. For example, the source repository has three `<collection>` elements, the published repository can have one, two or three `<collection>` elements. Assertions are not made about these details.

class `pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo.`**`UpdateInfoTestCase`**(*methodName='runTest'*)

Tests to ensure `updateinfo.xml` can be created and is valid.

classmethod `setUpClass`()

Create, populate and publish a repository.

More specifically, do the following:

1. Create an RPM repository with a distributor.
2. Populate the repository with an RPM and two errata, where one erratum references the RPM, and the other does not.
3. Publish the repository Fetch and parse its `updateinfo.xml` file.

`test_erratum`()

Assert the erratum generated by Pulp is correct.

Walk through each top-level element of the erratum which was generated during this test case's set-up and uploaded to Pulp. For each element, verify that the `updateinfo.xml` file generated by Pulp has a corresponding entry. Each of the `verify_*` methods on this test case implement the test logic for a single element.

`test_one_task_per_import`()

Assert only one task is spawned per erratum upload.

`test_reboot_not_suggested`()

Assert the update info tree does not suggest a spurious reboot.

The errata uploaded by this test case do not suggest that a reboot be applied. As a result, the relevant `<update>` element in the `updateinfo.xml` file should not have a `<reboot_suggested>` tag. Verify that this is so. See Pulp #2032.

Note: In previous versions of Pulp, if no reboot should be applied, a `<reboot_suggested>False</reboot_suggested>` element would be present. See Pulp #1782.

`test_tasks_result`()

Assert each task's result success flag (if present) is true.

This test assumes *`test_one_task_per_import()`* passes.

`test_tasks_state`()

Assert each task's state is "finished".

This test assumes *`test_one_task_per_import()`* passes.

`test_update_elements`()

Assert there is one "update" element in `updateinfo.xml`.

test_updates_element()
Assert `updateinfo.xml` has a root element named `updates`.

verify_description(*erratum*, *update_element*)
Verify `erratum` and `update_element` have same description.

verify_id(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same id.

verify_issued(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same issued.

verify_pkglist(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same pkglist.

verify_references(*erratum*, *update_element*)
Verify `erratum` and `update_element` have same references.

verify_solution(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same solution.

verify_status(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same status.

verify_title(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same title.

verify_type(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same type.

verify_version(*erratum*, *update_element*)
Verify `erratum` and `update_element` have the same version.

class pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo.**UpdateRepoTestCase**(*methodName='runTest'*)
Verify `updateinfo.xml` changes as its repo changes.

classmethod setUpClass()
Create an RPM repository with a feed and distributor.

test_01_sync_publish()
Sync and publish the repository.

When executed, this test method will fetch `updateinfo.xml` and verify that:

- An `<update>` with an `<id>` of *pulp_smash.constants.RPM_ERRATUM_ID* is present, and
- one of its child `<package>` elements has a "name" attribute equal to *pulp_smash.constants.RPM_ERRATUM_RPM_NAME*.

test_02_unassociate_publish()
Unassociate a content unit and publish the repository.

Fetch `updateinfo.xml`. Verify that an `<update>` with an `<id>` of *pulp_smash.constants.RPM_ERRATUM_ID* is not present.

pulp_smash.tests.pulp2.rpm.api_v2.test_updateinfo.**setUpModule**()
Possibly skip the tests in this module.

Skip tests if Pulp #2277 affects us.

6.102 pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish* Tests that upload to and publish RPM repositories.

For information on repository upload and publish operations, see Uploading Content and Publication.

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**UploadDrpmTestCase**(*methodName='runT*
Test whether one can upload a DRPM into a repository.

Specifically, this method does the following:

1. Create a yum repository.
2. Upload a DRPM into the repository.
3. Search for all content units in the repository.

This test case targets Pulp Smash #336

test_all()
Import a DRPM into a repository and search it for content units.

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**UploadDrpmTestCaseWithCheckSumTyp**
Test whether one can upload a DRPM into a repository.

Pulp issue #2627 caused uploading to fail when "checksumtype" was specified.

This test case targets Pulp Smash #585

classmethod setUpClass()
Import a DRPM into a repository and search it for content units.

Specifically, this method does the following:

1. Create a yum repository.
2. **Upload a DRPM into the repository with "checksumtype" set to** "sha256"
3. Search for all content units in the repository.

test_all()
Test that uploading DRPM with checksumtype specified works.

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**UploadRpmTestCase**(*methodName='runTes*
Test whether one can upload, associate and publish RPMs.

The test procedure is as follows:

1. Create a pair of repositories.
2. Upload an RPM to the first repository, and publish it.
3. Copy the RPM to the second repository, and publish it.

classmethod setUpClass()
Create a pair of RPM repositories.

classmethod tearDownClass()
Clean up resources created during the test.

test_01_upload_publish()
Upload an RPM to the first repository, and publish it.

Execute *verify_repo_search()* and *verify_repo_download()*.

test_02_copy_publish()
Copy and RPM from the first repo to the second, and publish it.

Execute *verify_repo_search()* and *verify_repo_download()*.

test_03_compare_repos()
Verify the two repositories contain the same content unit.

verify_repo_download(*repo*)
Download *pulp_smash.constants.RPM* from the given `repo`.

Verify that it is exactly equal to the one uploaded earlier.

verify_repo_search(*repo*)
Search for units in the given `repo`.

Verify that only one content unit is in `repo`, and that several of its metadata attributes are correct. This test targets Pulp #2365 and Pulp #2754

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**UploadSrpmTestCase**(*methodName='runT*
Test whether one can upload a SRPM into a repository.

This test case targets Pulp Smash #402

classmethod setUpClass()
Import a SRPM into a repository and search it for content units.

Specifically, this method does the following:

1. Create a yum repository.
2. Upload a SRPM into the repository.
3. Search for all content units in the repository.

test_srpm_file_name_is_correct()
Test if SRPM extracted correct metadata for creating filename.

test_srpm_uploaded_successfully()
Test if SRPM has been uploaded successfully.

test_status_code_units()
Verify the HTTP status code for repo units response.

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**UploadedDrpmChecksumTypeTestCase**(
Verify uploaded DRPMs have checksums of the requested type.

test_all()
Verify uploaded DRPMs have checksums of the requested type.

Specifically, this method does the following:

1. Create a yum repository.
2. Upload a DRPM into the repository with "checksumtype" set to "md5".
3. Assert that "checksumtype" was set to "md5".
4. Assert that checksum value was calculated according to "md5".

This test targets:

- Pulp #2774
- Pulp Smash #663

class pulp_smash.tests.pulp2.rpm.api_v2.test_upload_publish.**VendorInfoTestCase**(*methodName='runT*
Test whether the vendor info is present in case of sync and upload.

This tests case targets the following issues:

- Pulp Smash #680
- Pulp #2781

do_test(*repo*)
Verify that the given `repo` has an RPM with vendor data.

Perform the following checks:

- Search `repo` for RPMs and filter with a valid unit license. Assert that one search result is returned, and that it has vendor metadata.
- Search `repo` for RPMs and filter with a valid unit vendor. Assert that one search result is returned, and that it has vendor metadata.
- Search `repo` for RPMs and filter with an invalid unit license. Assert that no search results are returned.
- Search `repo` for RPMs and filter with an invalid unit vendor. Assert that no search results are returned.

The license-based searches serve as a sanity check. They show that the search syntax is correct.

classmethod setUpClass()
Create class-wide variables.

test_sync()
Test whether Pulp recognizes a synced RPM's vendor information.

Create a repository, sync in an RPM with a non-null vendor, and perform several checks. See *do_test()*.

test_upload()
Test whether Pulp recognizes an uploaded RPM's vendor information.

Create a repository, upload an RPM with a non-null vendor, and perform several checks. See *do_test()*.

6.103 pulp_smash.tests.pulp2.rpm.api_v2.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.api_v2.utils* Utility functions for RPM API tests.

class pulp_smash.tests.pulp2.rpm.api_v2.utils.**DisableSELinuxMixin**
A mixin providing the ability to temporarily disable SELinux.

maybe_disable_selinux(*cfg*, *pulp_issue_id*)
Disable SELinux if appropriate.

If the given Pulp issue is unresolved, and if SELinux is installed and enforcing on the target Pulp system, then disable SELinux and schedule it to be re-enabled. (Method `addCleanup` is used for the schedule.)

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.
- **pulp_issue_id** – The (integer) ID of a Pulp issue. If the referenced issue is fixed in the Pulp system under test, this method immediately returns.

Returns Nothing.

class pulp_smash.tests.pulp2.rpm.api_v2.utils.**TemporaryUserMixin**

A mixin providing the ability to create a temporary user.

A typical usage of this mixin is as follows:

```
ssh_user, priv_key = self.make_user(cfg)
ssh_identity_file = self.write_private_key(cfg, priv_key)
```

This mixin requires that the `unittest.TestCase` class from the standard library be a parent class.

static delete_user(*cfg*, *username*)

Delete a user.

The Pulp rsync distributor has a habit of leaving (idle?) SSH sessions open even after publishing a repository. When executed, this function will:

1. Poll the process list until all processes belonging to `username` have died, or raise a `unittest.SkipTest` exception if the time limit is exceeded.
2. Delete `username`.

make_user(*cfg*)

Create a user account with a home directory and an SSH keypair.

In addition, schedule the user for deletion with `self.addCleanup`.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the host being targeted.

Returns A `(username, private_key)` tuple.

write_private_key(*cfg*, *private_key*)

Write the given private key to a file on disk.

Ensure that the file is owned by user "apache" and has permissions of `600`. In addition, schedule the key for deletion with `self.addCleanup`.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the host being targeted.

Returns The path to the private key on disk, as a string.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**gen_distributor**()

Return a semi-random dict for use in creating a YUM distributor.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**gen_repo**()

Return a semi-random dict for use in creating an RPM repository.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**gen_repo_group**()

Return a semi-random dict for use in creating a RPM repository group.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**get_dists_by_type_id**(*cfg*, *repo*)

Return the named repository's distributors, keyed by their type IDs.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **repo_href** – A dict of information about a repository.

Returns A dict in the form `{'type_id': {distributor_info}}`.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**get_repodata**(*cfg*, *distributor*, *type_*, *response_handler=None*, *repomd_xml=None*)

Download a file of the given `type_` from a `repodata/` directory.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **distributor** – A dict of information about a repository distributor.
- **type** – The type of file to fetch from a repository's `repodata/` directory. Valid values might be "updateinfo" or "group".
- **response_handler** – The callback function used by *pulp_smash.api.Client* after downloading the `repomd.xml` file. Defaults to *xml_handler()*. Use *pulp_smash.api.safe_handler()* if you want the raw response.
- **repomd_xml** – A `repomd.xml` file as an `ElementTree`. If not given, *get_repodata_repomd_xml()* is consulted.

Returns Whatever is dictated by `response_handler`.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**get_repodata_repomd_xml**(*cfg*, *distributor*, *response_handler=None*)

Download the given repository's `repodata/repomd.xml` file.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **distributor** – A dict of information about a repository distributor.
- **response_handler** – The callback function used by *pulp_smash.api.Client* after downloading the `repomd.xml` file. Defaults to *xml_handler()*. Use *pulp_smash.api.safe_handler()* if you want the raw response.

Returns Whatever is dictated by `response_handler`.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**get_rpm_names_versions**(*cfg*, *repo*)

Get a dict of a repository's RPMs and their versions.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp app.
- **repo** – A dict of information about a repository.

Returns The name and versions of each package in the repository, with the versions sorted in ascending order. For example: `{'walrus': ['0.71', '5.21']}`.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**get_unit**(*cfg*, *distributor*, *unit_name*, *primary_xml=None*)

Download a file from a published repository.

A typical invocation is as follows:

```
>>> foo_rpm = get_unit(cfg, repo['distributors'][0], 'foo.rpm')
```

If multiple units are being fetched, efficiency can be improved by passing in a parsed `primary.xml` file:

```
>>> distributor = repo['distributors'][0]
>>> primary_xml = get_repodata(cfg, distributor, 'primary')
```

```
>>> foo_rpm = get_unit(cfg, distributor, 'foo.rpm', primary_xml)
>>> bar_rpm = get_unit(cfg, distributor, 'bar.rpm', primary_xml)
```

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **distributor** – A dict of information about a repository distributor.
- **unit_name** – The name of a content unit to be fetched. For example: "bear-4.1-1.noarch.rpm".
- **primary_xml** – A `primary.xml` file as an `ElementTree`. If not given, *get_repodata()* is consulted.

Returns A raw response. The unit is available as `response.content`.

pulp_smash.tests.pulp2.rpm.api_v2.utils.**set_pulp_manage_rsync**(*cfg*, *boolean*)

Set the `pulp_manage_rsync` SELinux policy.

If the `semanage` executable is not available, return. (This is the case if SELinux isn't installed on the system under test.) Otherwise, set the `pulp_manage_rsync SELinux policy on or off, depending on the truthiness of ``boolean`.

For more information on the `pulp_manage_rsync` SELinux policy, see ISO rsync Distributor → Configuration.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **boolean** – Either `True` or `False`.

Returns Information about the executed command, or `None` if no command was executed.

Return type *pulp_smash.cli.CompletedProcess*

pulp_smash.tests.pulp2.rpm.api_v2.utils.**xml_handler**(*_*, *response*)

Decode a response as if it is XML.

This API response handler is useful for fetching XML files made available by an RPM repository. When it handles a response, it will check the status code of `response`, decompress the response if the request URL ended in `.gz`, and return an `xml.etree.Element` instance built from the response body.

Note:

- The entire response XML is loaded and parsed before returning, so this may be unsafe for use with large XML files.
- The `Content-Type` and `Content-Encoding` response headers are ignored due to https://pulp.plan.io/issues/1781.

6.104 *pulp_smash.tests.pulp2.rpm.cli*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli* Tests that communicate with the server via the CLI.

6.105 pulp_smash.tests.pulp2.rpm.cli.test_character_encoding

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_character_encoding* Tests for Pulp's character encoding handling.

RPM files have metadata embedded in their headers. This metadata should be encoded as utf-8, and Pulp should gracefully handle cases where invalid byte sequences are encountered.

class pulp_smash.tests.pulp2.rpm.cli.test_character_encoding.**UploadNonAsciiTestCase**(*methodName*
Test whether one can upload an RPM with non-ascii metadata.

Specifically, do the following:

1. Create an RPM repository.
2. Upload and import *pulp_smash.constants.RPM_WITH_NON_ASCII_URL* into the repository.

test_all()
Test whether one can upload an RPM with non-ascii metadata.

class pulp_smash.tests.pulp2.rpm.cli.test_character_encoding.**UploadNonUtf8TestCase**(*methodName=*
Test whether one can upload an RPM with non-utf-8 metadata.

Specifically, do the following:

1. Create an RPM repository.
2. Upload and import *pulp_smash.constants.RPM_WITH_NON_UTF_8_URL* into the repository.

This test case targets Pulp #1903.

test_all()
Test whether one can upload an RPM with non-utf-8 metadata.

pulp_smash.tests.pulp2.rpm.cli.test_character_encoding.**setUpModule**()
Execute `pulp-admin login`.

6.106 pulp_smash.tests.pulp2.rpm.cli.test_copy_units

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_copy_units* Tests that copy units from one repository to another.

class pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**CopyLangpacksTestCase**(*methodName='runTest'*)
Copy langpacks from one repository to another.

This test case verifies that it is possible to use the `pulp-admin rpm repo copy langpacks` command to copy langpacks from one repository to another. See Pulp Smash #255.

test_all()
Copy langpacks from one repository to another.

Assert that:

- `pulp-admin` does not produce any errors.
- A non-zero number of langpacks are present in the target repository.

class pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**CopyRecursiveTestCase**(*methodName='runTest'*)
Recursively copy a "chimpanzee" unit from one repository to another.

This test case verifies that it is possible to use the `pulp-admin rpm repo copy` command to recursively copy units from one repository to another. See Pulp Smash #107.

test_all()
Recursively copy a "chimpanzee" unit from one repository to another.

"chimpanzee" depends on "walrus," and there are multiple versions of "walrus" in the source repository. Verify that one "walrus" unit has been copied to the target repository, and that the newer one has been copied.

class pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**CopyTestCase**(*methodName='runTest'*)
Copy a "chimpanzee" unit from one repository to another.

This test case verifies that it is possible to use the `pulp-admin rpm repo copy` command to copy a single unit from one repository to another.

test_all()
Copy a "chimpanzee" unit from one repository to another.

Verify that only the "chimpanzee" unit is in the target repository.

class pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**UpdateRpmTestCase**(*methodName='runTest'*)
Update an RPM in a repository and on a host.

Do the following:

1. Create two repositories. Populate the first, and leave the second empty.
2. Pick an RPM with at least two versions.
3. Copy the older version of the RPM from the first repository to the second, and publish the second repository.
4. Pick a host system capable of installing RPMs. (By default, this is the system hosting Pulp.) Make it add the second repository as a source of packages, and make it install the RPM.
5. Copy the newer version of the RPM from the first repository to the second, and publish the second repository.
6. Make the host install the newer RPM with `yum update rpm_name`, or a similar command.

This test case targets Pulp Smash #311.

test_all()
Update an RPM in a repository and on a host.

class pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**UtilsMixin**
A mixin providing useful tools to unittest subclasses.

Any class inheriting from this mixin must also inherit from `unittest.TestCase` or a compatible clone.

create_repo(*cfg*)
Create a repository and schedule it for deletion.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system on which to create a repository.

Returns The repository's ID.

pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**setUpModule**()
Possibly skip tests. Create and sync an RPM repository.

Skip tests in this module if the RPM plugin is not installed on the target Pulp server. Then create an RPM repository with a feed and sync it. Test cases may copy data from this repository but should **not** change it.

pulp_smash.tests.pulp2.rpm.cli.test_copy_units.**tearDownModule**()
Delete the repository created by `setUpModule`.

6.107 pulp_smash.tests.pulp2.rpm.cli.test_environments

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_environments* Tests for RPM package environments.

class pulp_smash.tests.pulp2.rpm.cli.test_environments.**UploadPackageEnvTestCase**(*methodName='run*
Test whether Pulp can upload package environments into a repository.

This test case covers Pulp #1003 and the corresponding Pulp Smash issue Pulp Smash #319. The following test steps are based on official Pulp RPM Recipes.

1. Create and sync a repository.
2. Upload the environmrnt into this repo and check if there's any error.

classmethod setUpClass()
Create and sync a repository.

classmethod tearDownClass()
Destroy the repository.

test_upload_environment()
Test if package environments can be uploaded.

6.108 pulp_smash.tests.pulp2.rpm.cli.test_langpacks

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_langpacks* Tests for Pulp's langpack support.

class pulp_smash.tests.pulp2.rpm.cli.test_langpacks.**UploadAndRemoveLangpacksTestCase**(*methodNam*
Test whether one can upload to and remove langpacks from a repository.

This test targets Pulp Smash #270. The test steps are as follows:

1. Create a repository.
2. Upload langpacks to the repository. Verify the correct number of langpacks are present.
3. Remove langpacks from the repository. Verify that no langpacks are present.

classmethod setUpClass()
Create a repository.

classmethod tearDownClass()
Delete the repository created by *setUpClass()*.

test_01_upload_langpacks()
Upload a langpack to the repository.

test_02_remove_langpacks()
Remove all langpacks from the repository.

6.109 pulp_smash.tests.pulp2.rpm.cli.test_process_recycling

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_process_recycling* Test Pulp's ability to recycle processes.

class pulp_smash.tests.pulp2.rpm.cli.test_process_recycling.**MaxTasksPerChildTestCase**(*methodNam*
Test Pulp's handling of its PULP_MAX_TASKS_PER_CHILD setting.

The PULP_MAX_TASKS_PER_CHILD setting controls how many tasks a worker process executes before being destroyed. Setting this option to a low value, like 2, ensures that processes don't have a chance to consume large amounts of memory.

Test this feature by doing the following:

1. Use ps to see how Pulp worker processes were invoked. None should have the --maxtasksperchild option set.
2. Set PULP_MAX_TASKS_PER_CHILD and restart Pulp. Use ps to see how Pulp worker processes were invoked. Each should have the --maxtasksperchild option set as appropriate.
3. Execute a sync and publish. No errors should be reported.
4. Reset the PULP_MAX_TASKS_PER_CHILD option and restart Pulp. ps to see how Pulp worker processes were invoked. Each should have the --maxtasksperchild option set as appropriate.

For more information, see Pulp #2172.

test_all()
Test Pulp's handling of its PULP_MAX_TASKS_PER_CHILD setting.

pulp_smash.tests.pulp2.rpm.cli.test_process_recycling.**get_pulp_worker_procs**(*cfg*)
Use ps aux to get information about each Pulp worker process.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.

Returns An iterable of strings, one per line of matching output.

pulp_smash.tests.pulp2.rpm.cli.test_process_recycling.**restart_pulp**(*cfg*)
Restart all of Pulp's services.

Unlike *pulp_smash.utils.reset_pulp()*, do not reset the state of any of Pulp's services.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.

pulp_smash.tests.pulp2.rpm.cli.test_process_recycling.**setUpModule**()
Conditionally skip this module, and execute pulp-admin login.

6.110 *pulp_smash.tests.pulp2.rpm.cli.test_retain_old_count*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_retain_old_count* Test the retain_old_count feature.

When more than one version of an RPM is present in a repository and Pulp syncs from that repository, it must choose how many versions of that RPM to sync. By default, it syncs all versions of that RPM. The retain_old_count option lets one sync a limited number of outdated RPMs.

class pulp_smash.tests.pulp2.rpm.cli.test_retain_old_count.**RetainOldCountTestCase**(*methodName='*
Test the retain_old_count feature.

This test targets Pulp #2785.

create_sync_repo(*retain_old_count*)
Create and sync a repository. Return information about it.

Implement the logic described by the test_retain_* methods.

classmethod setUpClass()
Create, populate and publish a repository.

Ensure at least two versions of an RPM are present in the repository.

classmethod tearDownClass()
Destroy the repository created by *setUpClass()*.

test_retain_one()
Give retain_old_count a value of one.

Create a repository whose feed references the repository created in *setUpClass()*, and whose retain_old_count option is one. Sync the repository, and assert that one old version of any duplicate RPMs were copied over.

test_retain_zero()
Give retain_old_count a value of zero.

Create a repository whose feed references the repository created by *setUpClass()*, and whose retain_old_count option is zero. Sync the repository, and assert that zero old versions of any duplicate RPMs were copied over.

6.111 *pulp_smash.tests.pulp2.rpm.cli.test_search*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_search* Tests that perform searches.

class pulp_smash.tests.pulp2.rpm.cli.test_search.**SearchReposWithFiltersTestCase**(*methodName='run*
Search for repositories, and use filters to limit matches.

This test case targets Pulp #1784 and Pulp Smash #184. The repository search documentation describes the CLI search syntax.

static gen_commands(*repo_id*)
Generate the commands used by the test methods.

Commands with the following filters are returned:

```
--filters {'id':'...'}
--filters {'repo_id':'...'}
--str-eq id=...
--str-eq repo_id=...
```

classmethod setUpClass()
Create a repository.

classmethod tearDownClass()
Delete the repository created by *setUpClass()*.

test_negative_searches()
Search for the repository with a non-matching repository ID.

test_positive_searches()
Search for the repository with a matching repository ID.

6.112 *pulp_smash.tests.pulp2.rpm.cli.test_sync*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_sync* Tests that sync RPM repositories.

class pulp_smash.tests.pulp2.rpm.cli.test_sync.**ForceSyncTestCase**(*methodName='runTest'*)
Test whether one can force Pulp to perform a full sync.

This test case targets Pulp #1982 and Pulp Smash #353. The test procedure is as follows:

1. Create and sync a repository.
2. Remove some number of RPMs from `/var/lib/pulp/content/units/rpm/`. Verify they are missing.
3. Sync the repository. Verify the RPMs are still missing.
4. Sync the repository with `--force-full true`. Verify all RPMs are present.

test_all()
Test whether one can force Pulp to perform a full sync.

class pulp_smash.tests.pulp2.rpm.cli.test_sync.**RemovedContentTestCase**(*methodName='runTest'*)
Test whether Pulp can re-sync content into a repository.

This test case targets Pulp #1775 and the corresponding Pulp Smash issue, Pulp Smash #243.

1. Create and sync a repository. Select a content unit.
2. Delete the content unit from the repository, and verify it's absent.
3. Sync the repository, and verify that the content unit is present.

test_all()
Test whether Pulp can re-sync content into a repository.

pulp_smash.tests.pulp2.rpm.cli.test_sync.**get_rpm_names**(*cfg*, *repo_id*)
Get a list of names of all packages in a repository.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp deployment.
- **repo_id** – A RPM repository ID.

Returns The names of all modules in a repository, as an `list`.

pulp_smash.tests.pulp2.rpm.cli.test_sync.**setUpModule**()
Execute `pulp-admin login` and reset Pulp.

For *RemovedContentTestCase* to function correctly, we require that all of the content units on Pulp's filesystem belong to the repository created by that test case. Resetting Pulp guarantees that this is so. Ideally, all test cases would clean up after themselves so that no resets are necessary.

pulp_smash.tests.pulp2.rpm.cli.test_sync.**sync_repo**(*cfg*, *repo_id*, *force_sync=False*)
Sync an RPM repository.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp deployment.
- **repo_id** – A RPM repository ID.
- **repo_id** – A boolean flag to denote if is a force-full sync.

Returns A *pulp_smash.cli.CompletedProcess*.

6.113 pulp_smash.tests.pulp2.rpm.cli.test_upload

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.test_upload* Tests that upload content units into repositories.

class pulp_smash.tests.pulp2.rpm.cli.test_upload.**UploadDrpmTestCase**(*methodName='runTest'*)
Test whether one can upload a DRPM into a repository.

This test case targets Pulp Smash #336 and Pulp Smash #585

test_upload()
Create a repository and upload DRPMs into it.

Specifically, do the following:

1. Create a yum repository.
2. Download a DRPM file.
3. Upload the DRPM into it. Use `pulp-admin` to verify its presence in the repository.
4. Upload the same DRPM into the same repository, and use the `--skip-existing` flag during the upload. Verify that Pulp skips the upload.

test_upload_with_checksumtype()
Create a repository and upload DRPMs into it.

Specifically, do the following:

1. Create a yum repository.
2. Download a DRPM file.
3. Upload the DRPM into it specifying the checksumtype
4. Use `pulp-admin` to verify its presence in the repository.
5. Upload the same DRPM into the same repository, and use the `--skip-existing` flag during the upload. Verify that Pulp skips the upload.

pulp_smash.tests.pulp2.rpm.cli.test_upload.**setUpModule**()
Execute `pulp-admin login` on the target Pulp system.

6.114 pulp_smash.tests.pulp2.rpm.cli.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.cli.utils* Utility functions for RPM CLI tests.

pulp_smash.tests.pulp2.rpm.cli.utils.**count_langpacks**(*server_config*, *repo_id*)
Tell how many langpack content units are in the given repository.

Parameters

- **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.
- **repo_id** – A repository ID.

Returns The number of langpacks in the named repository, as an integer.

6.115 *pulp_smash.tests.pulp2.rpm.utils*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp2.rpm.utils* Utilities for RPM tests.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2277**(*cfg*)
Return true if Pulp #2277 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2354**(*cfg*)
Return true if Pulp #2354 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2387**(*cfg*)
Return true if Pulp #2387 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2620**(*cfg*)
Return true if Pulp #2620 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2798**(*cfg*)
Return true if Pulp #2798 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_2844**(*cfg*)
Return true if Pulp #2844 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**check_issue_3104**(*cfg*)
Return true if Pulp #3104 affects the targeted Pulp system.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – The Pulp system under test.

pulp_smash.tests.pulp2.rpm.utils.**gen_yum_config_file**(*cfg*, *repositoryid*, ***kwargs*)
Generate a yum configuration file and write it to `/etc/yum.repos.d/`.

Generate a yum configuration file containing a single repository section, and write it to `/etc/yum.repos.d/{repositoryid}.repo`.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – The system on which to create a yum configuration file.
- **repositoryid** – The section's `repositoryid`. Used when naming the configuration file and populating the brackets at the head of the file. For details, see yum.conf(5).
- **kwargs** – Section options. Each kwarg corresponds to one option. For details, see yum.conf(5).

Returns The path to the yum configuration file.

pulp_smash.tests.pulp2.rpm.utils.**os_is_rhel6**(*cfg*)
Return `True` if the server runs RHEL 6, or `False` otherwise.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the system being targeted.

Returns True or false.

pulp_smash.tests.pulp2.rpm.utils.**set_up_module**()
Skip tests if the RPM plugin is not installed.

See *pulp_smash.tests* for more information.

6.116 pulp_smash.tests.pulp3

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3* Tests for Pulp 3.

6.117 pulp_smash.tests.pulp3.constants

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.constants* Constants for Pulp 3 tests.

6.118 pulp_smash.tests.pulp3.pulpcore

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore* Tests for Pulp 3's core.

6.119 pulp_smash.tests.pulp3.pulpcore.api_v3

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore.api_v3* Tests that communicate with Pulp 3 via the v3 API.

6.120 pulp_smash.tests.pulp3.pulpcore.api_v3.test_auth

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore.api_v3.test_auth* Tests for Pulp 3's authentication API.

For more information, see the documentation on Authentication.

class pulp_smash.tests.pulp3.pulpcore.api_v3.test_auth.**AuthTestCase**(*methodName='runTest'*)
Test Pulp3 Authentication.

classmethod setUpClass()
Create class-wide variables.

test_base_auth_failure()
Perform HTTP basic authentication with invalid credentials.

Assert that a response indicating failure is returned.

test_base_auth_success()
Perform HTTP basic authentication with valid credentials.

Assert that a response indicating success is returned.

test_jwt_failure()
Perform JWT authentication with invalid credentials.

Assert that a response indicating failure is returned.

test_jwt_success()
Perform JWT authentication with valid credentials.

Assert that a response indicating success is returned.

class pulp_smash.tests.pulp3.pulpcore.api_v3.test_auth.**JWTResetTestCase**(*methodName='runTest'*)
Perform series of tests related to JWT reset.

setUp()
Create a user and several JWT tokens for that user.

Also, verify that the tokens are valid.

test_delete_user()
Delete the user, and verify their tokens are invalid.

test_reset_tokens()
Repeatedly reset the user's tokens, and verify they're invalid.

Repeatedly resetting tokens ensures that token resets work even when a user has no tokens.

6.121 *pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_repos*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_repos* Tests that CRUD repositories.

class pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_repos.**CRUDRepoTestCase**(*methodName='runTes*
CRUD repositories.

do_fully_update_attr(*attr*)
Update a repository attribute using HTTP PUT.

Parameters **attr** – The name of the attribute to update. For example, "description." The attribute to update must be a string.

do_partially_update_attr(*attr*)
Update a repository attribute using HTTP PATCH.

Parameters **attr** – The name of the attribute to update. For example, "description." The attribute to update must be a string.

setUp()
Create an API client.

classmethod setUpClass()
Create class-wide variables.

test_01_create_repo()
Create repository.

test_02_read_repo()
Read a repository by its href.

Assert that the response contains the correct repository name.

test_02_read_repos()
Search for the repository by its name.

Assert that just one search result is returned, and that the result has a correct name.

test_03_fully_update_desc()
Update a repository's description using HTTP PUT.

test_03_fully_update_name()
Update a repository's name using HTTP PUT.

test_03_partially_update_desc()
Update a repository's description using HTTP PATCH.

test_03_partially_update_name()
Update a repository's name using HTTP PATCH.

test_04_delete_repo()
Delete a repository.

6.122 pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_users

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_users* Tests that CRUD users.

class pulp_smash.tests.pulp3.pulpcore.api_v3.test_crud_users.**UsersCRUDTestCase**(*methodName='runT*
CRUD users.

setUp()
Create an API client.

classmethod setUpClass()
Create class-wide variables.

test_01_create_user()
Create a user.

test_02_read_user()
Read a user.

test_02_read_users()
Read all users. Verify that the created user is in the results.

test_03_fully_update_user()
Update a user info using HTTP PUT.

test_03_partially_update_user()
Update a user info using HTTP PATCH.

test_04_delete_user()
Delete an user.

6.123 pulp_smash.tests.pulp3.pulpcore.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.pulpcore.utils* Utilities for Pulpcore tests.

pulp_smash.tests.pulp3.pulpcore.utils.**gen_repo**()
Return a semi-random dict for use in creating a repository.

6.124 pulp_smash.tests.pulp3.utils

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.tests.pulp3.utils* Utility functions for Pulp 3 tests.

class pulp_smash.tests.pulp3.utils.**JWTAuth**(*token*, *header_format='Bearer'*)
A class that enables JWT authentication with the Requests library.

For more information, see the Requests documentation on custom authentication.

pulp_smash.tests.pulp3.utils.**get_auth**(*cfg=None*)
Return a random authentication object.

By default, *pulp_smash.api.Client* uses the same authentication method (HTTP BASIC) for every request. While this is a sane default, it doesn't let tests exercise other authentication procedures. This function returns a random authentication object. To demonstrate how this object can be used, here's an example showing how to create a user:

```
>>> from pulp_smash.api import Client
>>> from pulp_smash.config import get_config
>>> from pulp_smash.tests.pulp3.utils import get_auth
>>> from pulp_smash.tests.pulp3.constants import USER_PATH
>>> cfg = config.get_config()
>>> client = api.Client(cfg, api.json_handler)
>>> client.request_kwargs['auth'] = get_auth()
>>> client.post(USER_PATH, {
>>>     'username': 'superuser',
>>>     'password': 'admin',
>>>     'is_superuser': True
>>> })
```

The returned object can also be used directly with Requests. For more information, see the For more information, see the Requests Authentication documentation.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp app.

Returns A random authentication object.

pulp_smash.tests.pulp3.utils.**set_up_module**()
Skip tests if Pulp 3 isn't under test.

6.125 *pulp_smash.utils*

Location: *Pulp Smash* → *API Documentation* → *pulp_smash.utils* Utility functions for Pulp tests.

This module may make use of *pulp_smash.api* and *pulp_smash.cli*, but the reverse should not be done.

class pulp_smash.utils.**BaseAPICrudTestCase**(*methodName='runTest'*)
A parent class for API CRUD test cases.

create_body() and *update_body()* should be overridden by concrete child classes. The bodies of these two methods are encoded to JSON and used as the bodies of HTTP requests for creating and updating a repository, respectively. Be careful to return appropriate data when overriding these methods: the various `test*` methods assume the repository is fairly simple.

Relevant Pulp documentation:

Create http://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/repo/cud.html#create-a-repository

Read http://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/repo/retrieval.html#retrieve-a-single-repository

Update http://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/repo/cud.html#update-a-repository

Delete http://docs.pulpproject.org/en/latest/dev-guide/integration/rest-api/repo/cud.html#delete-a-repository

static **create_body**()
Return a dict for creating a repository. Should be overridden.

Raises NotImplementedError if not implemented by a child class.

classmethod **setUpClass**()
Create, update, read and delete a repository.

test_create()
Assert the created repository has all requested attributes.

Walk through each of the attributes returned by *create_body()* and verify the attribute is present in the repository.

NOTE: Any attribute whose name starts with importer or distributor is not verified.

test_importer_config()
Validate the config attribute of each importer.

test_importer_type_id()
Validate the repo importer's importer_type_id attribute.

test_number_importers()
Assert the repository has one importer.

test_read()
Assert the repo update response has the requested changes.

test_status_codes()
Assert each response has a correct status code.

test_update()
Assert the repo update response has the requested changes.

static **update_body**()
Return a dict for updating a repository. Should be overridden.

Raises NotImplementedError if not implemented by a child class.

class pulp_smash.utils.**BaseAPITestCase**(*methodName='runTest'*)
A class with behaviour that is of use in many API test cases.

This test case provides set-up and tear-down behaviour that is common to many API test cases. It is not necessary to use this class as the parent of all API test cases, but it serves well in many cases.

classmethod **setUpClass**()
Provide a server config and an iterable of resources to delete.

The following class attributes are created this method:

cfg A *pulp_smash.config.PulpSmashConfig* object.

resources A set object. If a child class creates some resources that should be deleted when the test is complete, the child class should add that resource's href to this set.

classmethod **tearDownClass**()
Delete all resources named by resources.

class pulp_smash.utils.**DuplicateUploadsMixin**
A mixin that adds tests for the "duplicate upload" test cases.

Consider the following procedure:

1. Create a new feed-less repository of any content unit type.

2. Upload a content unit into the repository.
3. Upload the same content unit into the same repository.

The second upload should silently fail for all Pulp releases in the 2.x series. See:

- https://pulp.plan.io/issues/1406
- https://github.com/PulpQE/pulp-smash/issues/81

This mixin adds tests for this case. Child classes should do the following:

- Create a repository. Content units will be uploaded into this repository.
- Create a class or instance attribute named `upload_import_unit_args`. It should be an iterable whose contents match the signature of *`upload_import_unit()`*.

test_01_first_upload()
Upload a content unit to a repository.

test_02_second_upload()
Upload the same content unit to the same repository.

pulp_smash.utils.**get_broker**(*server_config*)
Build an object for managing the target system's AMQP broker.

Talk to the host named by `server_config` and use simple heuristics to determine which AMQP broker is installed. If Qpid or RabbitMQ appear to be installed, return the name of that service. Otherwise, raise an exception.

Parameters **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the system on which an AMQP broker exists.

Returns A string such as 'qpidd' or 'rabbitmq'.

Raises ***pulp_smash.exceptions.NoKnownBrokerError*** – If unable to find any AMQP brokers on the target system.

pulp_smash.utils.**get_sha256_checksum**(*url*)
Return the sha256 checksum of the file at the given URL.

When a URL is encountered for the first time, do the following:

1. Download the file and calculate its sha256 checksum.
2. Cache the URL-checksum pair.
3. Return the checksum.

On subsequent calls, return a cached checksum.

pulp_smash.utils.**get_unit_type_ids**(*server_config*)
Tell which content unit types are supported by the target Pulp server.

Each Pulp plugin adds one (or more?) content unit types to Pulp, and each content unit type has a unique identifier. For example, the Python plugin[1] adds the Python content unit type[2], and Python content units have an ID of `python_package`. This function queries the server and returns those unit type IDs.

Parameters **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp deployment being targeted.

Returns A set of content unit type IDs. For example: `{'ostree', 'python_package'}`.

[1] http://docs.pulpproject.org/plugins/pulp_python/
[2] http://docs.pulpproject.org/plugins/pulp_python/reference/python-type.html

pulp_smash.utils.**http_get**(*url*, ***kwargs*)
Issue a HTTP request to the `url` and return the response content.

This is useful for downloading file contents over HTTP[S].

Parameters

- **url** – the URL where the content should be get.
- **kwargs** – additional kwargs to be passed to `requests.get`.

Returns the response content of a GET request to `url`.

pulp_smash.utils.**os_is_f26**(*cfg*, *pulp_system=None*)
Return `True` if the server runs Fedora 26, or `False` otherwise.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the system being targeted.
- **pulp_system** – A *pulp_smash.config.PulpSystem* to target, instead of the default chosen by *pulp_smash.cli.Client*.

Returns True or false.

pulp_smash.utils.**publish_repo**(*cfg*, *repo*, *json=None*)
Publish a repository.

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp host.
- **repo** – A dict of detailed information about the repository to be published.
- **json** – Data to be encoded as JSON and sent as the request body. Defaults to `{'id': repo['distributors'][0]['id']}`.

Raises `ValueError` when `json` is not passed, and `repo` does not have exactly one distributor.

Returns The server's reponse. Call `.json()` on the response to get a call report.

pulp_smash.utils.**pulp_admin_login**(*server_config*)
Execute `pulp-admin login`.

Parameters **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp server being targeted.

Returns The completed process.

Return type *pulp_smash.cli.CompletedProcess*

pulp_smash.utils.**reset_pulp**(*server_config*)
Stop Pulp, reset its database, remove certain files, and start it.

Parameters **server_config** (pulp_smash.config.PulpSmashConfig) – Information about the Pulp server being targeted.

Returns Nothing.

pulp_smash.utils.**reset_squid**(*cfg*)
Stop Squid, reset its cache directory, and restart it.

Parameters **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.

Returns Nothing.

pulp_smash.utils.**search_units**(*cfg*, *repo*, *criteria=None*, *response_handler=None*)
Find content units in a `repo`.

Parameters

- **cfg** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp host.
- **repo** – A dict of detailed information about the repository.
- **criteria** – A dict of criteria to pass in the search body. Defaults to an empty dict.
- **response_handler** – The callback function used by *pulp_smash.api.Client* after searching. Defaults to *pulp_smash.api.json_handler()*.

Returns Whatever is dictated by `response_handler`.

pulp_smash.utils.**set_up_module**()
Skip tests if Pulp 2 isn't under test.

pulp_smash.utils.**skip_if_type_is_unsupported**(*unit_type_id*, *server_config=None*)
Raise `SkipTest` if support for the named type is not availalble.

Parameters

- **unit_type_id** – A content unit type ID, such as "ostree".
- **server_config** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp server being targeted. If none is provided, the config returned by *pulp_smash.config.get_config()* is used.

Raises `unittest.SkipTest` if support is unavailable.

Returns Nothing.

pulp_smash.utils.**sync_repo**(*cfg*, *repo*)
Sync a repository.

Parameters

- **cfg** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp host.
- **repo** – A dict of detailed information about the repository to be published.

Returns The server's reponse. Call `.json()` on the response to get a call report.

pulp_smash.utils.**upload_import_erratum**(*server_config*, *erratum*, *repo_href*)
Upload an erratum to a Pulp server and import it into a repository.

For most content types, use *upload_import_unit()*.

Parameters

- **server_config** (`pulp_smash.config.PulpSmashConfig`) – Information about the Pulp server being targeted.
- **erratum** – A dict, with keys such as "id," "status," "issued," and "references."
- **repo_href** – The path to the repository into which `erratum` will be imported.

Returns The call report returned when importing the erratum.

pulp_smash.utils.**upload_import_unit**(*cfg*, *unit*, *import_params*, *repo*)
Upload a content unit to a Pulp server and import it into a repository.

This procedure only works for some unit types, such as `rpm` or `python_package`. Others, like `package_group`, require an alternate procedure. The procedure encapsulated by this function is as follows:

1. Create an upload request.

2. Upload the content unit to Pulp, in small chunks.
3. Import the uploaded content unit into a repository.
4. Delete the upload request.

The default set of parameters sent to Pulp during step 3 are:

```
{'unit_key': {}, 'upload_id': '...'}
```

The actual parameters required by Pulp depending on the circumstances, and the parameters sent to Pulp may be customized via the `import_params` argument. For example, if uploading a Python content unit, `import_params` should be the following:

```
{'unit_key': {'filename': '...'}, 'unit_type_id': 'python_package'}
```

This would result in the following upload parameters being used:

```
{
    'unit_key': {'filename': '...'},
    'unit_type_id': 'python_package',
    'upload_id': '...',
}
```

Parameters

- **cfg** (pulp_smash.config.PulpSmashConfig) – Information about a Pulp host.
- **unit** – The unit to be uploaded and imported, as a binary blob.
- **import_params** – A dict of parameters to be merged into the default set of import parameters during step 3.
- **repo** – A dict of information about the target repository.

Returns The call report returned when importing the unit.

pulp_smash.utils.**uuid4**()
Return a random UUID4 as a string.

6.126 *tests*

Location: *Pulp Smash* → *API Documentation* → *tests* Unit tests for Pulp Smash.

This package and its sub-packages contain tests for Pulp Smash. These tests verify that Pulp Smash's internal functions and libraries function correctly.

These tests are entirely different from the tests in *pulp_smash.tests*.

6.127 *tests.test_api*

Location: *Pulp Smash* → *API Documentation* → *tests.test_api* Unit tests for *pulp_smash.api*.

class tests.test_api.**ClientTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.api.Client*.

classmethod setUpClass()
Assert methods delegate to *pulp_smash.api.Client.request()*.

All methods on *pulp_smash.api.Client*, such as *pulp_smash.api.Client.delete()*, should delegate to *pulp_smash.api.Client.request()*. Mock out `request` and call the other methods.

test_called_once()
Assert each method calls `request` exactly once.

test_http_action()
Assert each method calls `request` with the right HTTP action.

class tests.test_api.**ClientTestCase2**(*methodName='runTest'*)
More tests for *pulp_smash.api.Client*.

test_json_arg()
Assert methods with a `json` argument pass on that argument.

test_response_handler()
Assert `__init__` saves the `response_handler` argument.

The argument should be saved as an instance attribute.

class tests.test_api.**CodeHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.api.code_handler()*.

test_202_check_skipped()
Assert HTTP 202 responses are not treated specially.

test_raise_for_status()
Assert `response.raise_for_status()` is called.

test_return()
Assert the passed-in `response` is returned.

class tests.test_api.**EchoHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.api.echo_handler()*.

test_202_check_skipped()
Assert HTTP 202 responses are not treated specially.

test_raise_for_status()
Assert `response.raise_for_status()` is not called.

test_return()
Assert the passed-in `response` is returned.

class tests.test_api.**JsonHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.api.json_handler()*.

test_202_check_run()
Assert HTTP 202 responses are not treated specially.

test_raise_for_status()
Assert `response.raise_for_status()` is called.

test_return()
Assert the JSON-decoded body of `response` is returned.

class tests.test_api.**SafeHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.api.safe_handler()*.

test_202_check_run()
Assert HTTP 202 responses are not treated specially.

test_raise_for_status()
Assert `response.raise_for_status()` is called.

test_return()
Assert the passed-in `response` is returned.

6.128 *tests.test_cli*

Location: *Pulp Smash* → *API Documentation* → *tests.test_cli* Unit tests for *pulp_smash.cli*.

class tests.test_cli.**ClientTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.cli.Client*.

test_default_response_handler()
Assert the default response handler checks return codes.

test_explicit_local_transport()
Assert it is possible to explicitly ask for a "local" transport.

test_explicit_pulp_system()
Assert it is possible to explicitly target a pulp cli PulpSystem.

test_explicit_response_handler()
Assert it is possible to explicitly set a response handler.

test_implicit_local_transport()
Assert it is possible to implicitly ask for a "local" transport.

test_implicit_pulp_system()
Assert it is possible to implicitly target a pulp cli PulpSystem.

class tests.test_cli.**CodeHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.cli.code_handler()*.

classmethod setUpClass()
Call the function under test, and record inputs and outputs.

test_check_returncode()
Assert `completed_proc.check_returncode()` is not called.

test_input_returned()
Assert the passed-in `completed_proc` is returned.

class tests.test_cli.**CompletedProcessTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.cli.CompletedProcess*.

setUp()
Generate kwargs that can be used to instantiate a completed proc.

test_can_eval()
Assert `__repr__()` can be parsed by `eval()`.

test_check_returncode_nonzero()
Call `check_returncode` when `returncode` is not zero.

test_check_returncode_zero()
Call `check_returncode` when `returncode` is zero.

test_init()
Assert all constructor arguments are saved as instance attrs.

class tests.test_cli.**EchoHandlerTestCase**(*methodName='runTest'*)
Tests for *pulp_smash.cli.echo_handler()*.

classmethod setUpClass()
Call the function under test, and record inputs and outputs.

test_check_returncode()
Assert `completed_proc.check_returncode()` is not called.

test_input_returned()
Assert the passed-in `completed_proc` is returned.

6.129 *tests.test_config*

Location: *Pulp Smash* → *API Documentation* → *tests.test_config* Unit tests for *pulp_smash.config*.

class tests.test_config.**ConvertOldConfigTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.convert_old_config()*.

test_convert_old_config()
Assert the conversion works.

class tests.test_config.**GetConfigFilePathTestCase**(*methodName='runTest'*)
Test `pulp_smash.config._get_config_file_path`.

test_failures()
Assert the method raises an exception when no config is found.

test_success()
Assert the method returns a path when a config file is found.

class tests.test_config.**GetConfigTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.get_config()*.

test_cache_empty()
A config is read from disk if the cache is empty.

test_cache_full()
No config is read from disk if the cache is populated.

class tests.test_config.**GetRequestsKwargsTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.PulpSmashConfig.get_requests_kwargs()*.

classmethod setUpClass()
Create a mock server config and call the method under test.

test_cfg_auth()
Assert that the method does not alter the config's `auth`.

test_kwargs()
Assert that the method returns correct values.

test_kwargs_auth()
Assert that the method converts `auth` to a tuple.

class tests.test_config.**HelperMethodsTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.PulpSmashConfig()* helper methods.

setUp()
Generate contents for a configuration file.

test_get_systems()
`get_systems` returns proper result.

test_services_for_roles()
`services_for_roles` returns proper result.

class tests.test_config.**InitTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.PulpSmashConfig* instantiation.

classmethod setUpClass()
Generate some attributes and use them to instantiate a config.

test_private_attrs()
Assert that private attributes have been set.

test_public_attrs()
Assert that public attributes have correct values.

class tests.test_config.**PulpSmashConfigFileTestCase**(*methodName='runTest'*)
Verify the `PULP_SMASH_CONFIG_FILE` environment var is respected.

test_var_set()
Set the environment variable.

test_var_unset()
Do not set the environment variable.

class tests.test_config.**ReadTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.PulpSmashConfig.read()*.

test_read_config_file()
Ensure Pulp Smash can read the config file.

test_read_old_config_file()
Ensure Pulp Smash can read old config file format.

class tests.test_config.**ReprTestCase**(*methodName='runTest'*)
Test calling `repr` on a *pulp_smash.config.PulpSmashConfig*.

classmethod setUpClass()
Generate attributes and call the method under test.

test_can_eval()
Assert that the result can be parsed by `eval`.

test_is_sane()
Assert that the result is in an expected set of results.

class tests.test_config.**ValidateConfigTestCase**(*methodName='runTest'*)
Test *pulp_smash.config.validate_config()*.

test_config_missing_roles()
Missing required roles in config raises an exception.

test_invalid_config()
An invalid config raises an exception.

test_valid_config()
A valid config does not raise an exception.

6.130 *tests.test_pulp_smash_cli*

Location: *Pulp Smash* → *API Documentation* → *tests.test_pulp_smash_cli* Unit tests for `pulp_smash.pulp_smash_cli`.

class `tests.test_pulp_smash_cli.`**`BasePulpSmashCliTestCase`**(*methodName='runTest'*)
Base class for all pulp_smash_cli tests.

`setUp`()
Configure a CliRunner.

class `tests.test_pulp_smash_cli.`**`MissingSettingsFileMixin`**
Test missing settings file.

Classes that inherit from this mixin should provide the `settings_subcommand` attribute set to the settings subcommand to run.

`test_missing_settings_file`()
Ensure show outputs proper settings file.

class `tests.test_pulp_smash_cli.`**`SettingsCreateTestCase`**(*methodName='runTest'*)
Test `pulp_smash.pulp_smash_cli.settings_create` command.

`setUp`()
Generate a default expected config dict.

`test_create_defaults_and_verify`()
Create settings file with defaults and custom SSL certificate.

`test_create_other_values`()
Create settings file with custom values.

`test_create_with_defaults`()
Create settings file with default values values.

`test_settings_already_exists`()
Create settings file by overriding existing one.

class `tests.test_pulp_smash_cli.`**`SettingsPathTestCase`**(*methodName='runTest'*)
Test `pulp_smash.pulp_smash_cli.settings_path` command.

`test_settings_path`()
Ensure path outputs proper settings file path.

class `tests.test_pulp_smash_cli.`**`SettingsShowTestCase`**(*methodName='runTest'*)
Test `pulp_smash.pulp_smash_cli.settings_show` command.

`test_settings_show`()
Ensure show outputs proper settings file.

class `tests.test_pulp_smash_cli.`**`SettingsValidateTestCase`**(*methodName='runTest'*)
Test `pulp_smash.pulp_smash_cli.settings_validate` command.

`test_invalid_config`()
Ensure validate fails on invalid config file schema.

`test_old_config_alert`()
Ensure validate notifies about the old config format.

`test_valid_config`()
Ensure validate does not complain about valid settings.

6.131 tests.test_selectors

Location: *Pulp Smash* → *API Documentation* → *tests.test_selectors* Unit tests for `pulp_smash.selectors`.

class `tests.test_selectors.`**`BugIsTestableTestCase`**(*methodName='runTest'*)
Test `pulp_smash.selectors.bug_is_testable()` and its partner.

`test_connection_error`()
Make the dependent function raise a connection error.

`test_testable_status`()
Assert the method correctly handles "testable" bug statuses.

`test_unknown_status`()
Assert the method correctly handles an unknown bug status.

`test_untestable_status`()
Assert the method correctly handles "untestable" bug statuses.

class `tests.test_selectors.`**`ConvertTPRTestCase`**(*methodName='runTest'*)
Test method `_convert_tpr`.

`test_empty_version_string`()
Assert `version_string` is converted if it is an empty string.

`test_invalid_version_string`()
Assert an exception is raised if `version_string` is invalid.

`test_valid_version_string`()
Assert `version_string` is converted if it is valid.

class `tests.test_selectors.`**`GetBugTestCase`**(*methodName='runTest'*)
Test method `_get_bug`.

`test_invalid_bug_id`()
Assert an exception is raised if `bug_id` isn't an integer.

class `tests.test_selectors.`**`GetTPRTestCase`**(*methodName='runTest'*)
Test method `_get_tpr`.

`test_failure`()
Assert the method raises the correct exception no TPR is present.

`test_success`()
Assert the method returns the target platform release if present.

6.132 tests.test_utils

Location: *Pulp Smash* → *API Documentation* → *tests.test_utils* Unit tests for `pulp_smash.utils`.

class `tests.test_utils.`**`BaseAPITestCase`**(*methodName='runTest'*)
Test `pulp_smash.utils.BaseAPITestCase`.

classmethod `setUpClass`()
Define a child class. Call setup and teardown methods on it.

We define a child class in order to avoid altering `pulp_smash.utils.BaseAPITestCase`. Calling class methods on it would do so.

test_set_up_class()
Assert method `setUpClass` creates correct class attributes.

Verify that the method creates attributes named `cfg` and `resources`.

test_tear_down_class()
Call method `tearDownClass`, and assert it deletes each resource.

pulp_smash.api.Client.delete() should be called once for each resource listed in `resources`, and once for *pulp_smash.tests.pulp2.constants.ORPHANS_PATH*.

class tests.test_utils.**GetBrokerTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.get_broker()*.

test_failure()
Fail to generate a broker service management object.

Assert that *pulp_smash.exceptions.NoKnownBrokerError* is raised if the function cannot find a broker.

test_success()
Successfully generate a broker service management object.

Assert that:

- `get_broker(...)` returns a string.
- The `server_config` argument is passed to the service object.
- The "qpidd" broker is the preferred broker.

class tests.test_utils.**GetSha256ChecksumTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.get_sha256_checksum()*.

test_all()
Call the function three times, with two URLs.

Call the function with the first URL, the second URL and the first URL again. Verify that:

- No download is attempted during the third call.
- The first and second calls return different checksums.
- The first and third calls return identical checksums.

class tests.test_utils.**GetUnitTypeIdsTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.skip_if_type_is_unsupported()*.

test_ids_are_returned()
Assert each unit type ID in the server response is returned.

class tests.test_utils.**IsRootTestCase**(*methodName='runTest'*)
Test `pulp_smash.utils.is_root`.

test_false()
Assert the method returns `False` when non-root.

test_true()
Assert the method returns `True` when root.

class tests.test_utils.**OsIsF26TestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.os_is_f26()*.

test_returncode_nonzero()
Assert false is returned if the CLI command returns non-zero.

test_returncode_zero()
Assert true is returned if the CLI command returns zero.

class tests.test_utils.**PulpAdminLoginTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.pulp_admin_login()*.

test_run()
Assert the function executes `cli.Client.run`.

class tests.test_utils.**SearchUnitsTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.search_units()*.

test_defaults()
Verify that default parameters are correctly set.

class tests.test_utils.**SkipIfTypeIsUnsupportedTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.skip_if_type_is_unsupported()*.

setUp()
Generate a random unit type ID.

test_type_is_supported()
Assert nothing happens if the given unit type is supported.

Also assert *pulp_smash.config.get_config()* is not called, as we provide a *pulp_smash.config.PulpSmashConfig* argument.

test_type_is_unsupported()
Assert `SkipTest` is raised if the given unit type is unsupported.

Also assert *pulp_smash.config.get_config()* is called, as we do not provide a *pulp_smash.config.PulpSmashConfig* argument.

class tests.test_utils.**SyncRepoTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.sync_repo()*.

test_post()
Assert the function makes an HTTP POST request.

class tests.test_utils.**UUID4TestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.uuid4()*.

test_type()
Assert the method returns a unicode string.

class tests.test_utils.**UploadImportErratumTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.upload_import_unit()*.

test_post()
Assert the function makes an HTTP POST request.

class tests.test_utils.**UploadImportUnitTestCase**(*methodName='runTest'*)
Test *pulp_smash.utils.upload_import_unit()*.

test_post()
Assert the function makes an HTTP POST request.

Python Module Index

p

t

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X