PDC Client Documentation

Release pdc-client-1.8.0-4-30-g7236fd8-7236fd8

PDC Devel Team

Mar 29, 2018

Contents

PDC Client is Python API and scripts which simplify access to PDC server.

Contents:

CHAPTER 1

Installation

There are multiple ways to install PDC Client API and scripts on your system.

1. Install from packages provided by your distribution.

```
# install scripts
sudo yum install pdc-client

# install API for Python 3
sudo yum install python3-pdc-client

# install API for Python 2
sudo yum install python2-pdc-client
```

2. Install from PyPI.

```
pip install pdc-client
```

CHAPTER 2

Configuration

The client can read server connection details from a configuration file. The configuration file should be located in `/etc/pdc.d/` directory which contains `fedora.json`, or in `~/.config/pdc/client_config.json`. If both files are present, the system one is loaded first and the user configuration is applied on top of it (to add other options or overwrite existing ones).

The configuration file should contain a JSON object, which maps server name to JSON object with details. The name is an arbitrary string used at client run time to identify which server you want to connect to.

The details of a single server must contain at least one key: `host` which specifies the URL to the API root (e.g. `http://localhost:8000/rest_api/v1/` for local instance).

Other possible keys are:

- `token`

  If specified, this token will be used for authentication. The client will not try to obtain any token from the server.

- `ssl-verify`

  If set to `false`, server certificate will not be validated. See [Python requests documentation](http://docs.python-requests.org/en/master/user/advanced/#ssl-cert-verification) for other possible values.

- `develop`

  When set to `true`, the client will not use any authentication at all, not requesting a token nor sending any token with the requests. This is only useful for working with servers which don't require authentication.

- `plugins`

  Plugins are configurable which depends on the user's needs. If no plugins are configured, the default plugins will be used. If plugins are configured, they will be merged to the default ones.

2.1 Example

This config defines connection to development server running on localhost and a production server:

```
{
    "local": {
        "host": "http://localhost:8000/rest_api/v1/",
        "develop": true,
        "ssl-verify": false
    },
    "prod": {
        "host": "https://pdc.example.com/rest_api/v1/",
        "plugins": ["permission.py", "release.py"]
    }
}
```

CHAPTER 3

Python API

Usually you just need to import main class from the module.

```
from pdc_client import PDCClient
```

3.1 Examples

- Creating global components based on imported source RPMs
- Find components with multiple contacts of same role

3.2 Module pdc_client

exception pdc_client.**NoResultsError**(*response*)

Exception for getting all pages of data Raise this NoResultsError if there is an unexpected data returned when get all pages of data by results() function.

Data members:

- response – response object

__init__(*response*)

Create a NoResultsError

class pdc_client.**PDCClient**(*server*, *token=None*, *develop=None*, *ssl_verify=None*, *page_size=None*)

BeanBag wrapper specialized for PDC access.

This class wraps general BeanBag.v1 objects, but provides easy-to-use interface that can use configuration files for specifying server connections. The authentication token is automatically retrieved (if needed).

```
# Example: Get per page of release
client = PDCClient(<server>)
client.releases._()
client["releases"]._()
...

# Example: create one release data
client["releases"]._(<dict data>)
...

# Example: Iterate all pages of releases
# Will raise NoResultsError with response when return unexpected result.
try:
    for r in client["releases"].results():
        ...
except NoResultsError as e:
    # handle e.response ...
```

__getattr__(*name*)

If the first attribute/endpoint with "-", just replace with "_" in name.

```
# Example: get endpoint
client = PDCClient(<server>)
# Get the endpoint base-products/
client.base_products._
# Get the endpoint base-products/test_123/
client.base_products.test_123._
# Get the endpoint products/
client.products._
```

__init__(*server*, *token=None*, *develop=None*, *ssl_verify=None*, *page_size=None*)

Create new pdc client instance.

Once the class is instantiated, use it as you would use a regular BeanBag object. Please look at its documentation for how to use this class to perform requests.

Parameters

- **server** – Server API url or server name from configuration
- **token** – An authentication token string of visiting pdc server
- **develop** – This is use for dev mode
- **ssl_verify** – True for validating SSL certificates with system CA store; False for no validation; path to CA file or directory to use for validation otherwise
- **page_size** – This is a number of data which is returned per page. A -1 means that pdc server will return all the data in one request.

Raises *pdc_client.config.ServerConfigError* – on an configuration error

get_paged(*res*, ***kwargs*)

This call is equivalent to `res(**kwargs)`, only it retrieves all pages and returns the results joined into a single iterable. The advantage over retrieving everything at once is that the result can be consumed immediately.

Parameters

- **res** – what resource to connect to
- **kwargs** – filters to be used

```
# Example: Iterate over all active releases
for release in client.get_paged(client['releases']._, active=True):
    ...
```

This function is obsolete and not recommended.

obtain_token()
Try to obtain token from all end-points that were ever used to serve the token. If the request returns 404 NOT FOUND, retry with older version of the URL.

set_comment(*comment*)
Set PDC Change comment to be stored on the server.

Once you set the comment, it will be sent in all subsequent requests.

Parameters **comment** (*string*) – what comment to send to the server

class pdc_client.**PDCClientWithPage**(*server*, *token=None*, *develop=None*, *ssl_verify=None*, *page_size=None*, *page=None*)
PDCClient wrapper specialized for setting page in get_paged function.

__init__(*server*, *token=None*, *develop=None*, *ssl_verify=None*, *page_size=None*, *page=None*)
Create new client instance with page prarameter. Other params are all used for base class. :param page: the page number of the data.

get_paged(*res*, ***kwargs*)
Re-write the ge_paged here, and add the self.page check. This call is equivalent to `res(**kwargs)`, if there is no self.page parameter,only it retrieves all pages and returns the results joined into a single iterable. The advantage over retrieving everything at once is that the result can be consumed immediately.

Parameters

- **res** – what resource to connect to
- **kwargs** – filters to be used

```
# Example: Iterate over all active releases
for release in client.get_paged(client['releases']._, active=True):
    ...
```

If there is a self.page parameter here, just return that page's data with the self.page_size.

exception pdc_client.config.**ServerConfigConflictError**
Same server defined in multiple config files.

exception pdc_client.config.**ServerConfigError**
Base class for ServerConfiguration exceptions.

class pdc_client.config.**ServerConfigManager**(**paths*)
Provides configuration for given server name.

Configuration is read from multiple files or directories in order they're passed to constructor. Files and directories are read lazily when needed and at most once (cached for later access).

get(*server*)
Returns ServerConfig instance with configuration given server.

Raises

- ***ServerConfigConflictError*** – if configuration directory contains configuration for same server multiple times
- ***ServerConfigMissingUrlError*** – if URL is not specified in the configuration

- *ServerConfigNotFoundError* – if configuration for given server is not found

exception pdc_client.config.**ServerConfigMissingUrlError**
Server configuration is missing URL.

exception pdc_client.config.**ServerConfigNotFoundError**
Server configuration is missing.

CHAPTER 4

Script pdc_client

Note: Add argument `-h` or `--help` to get general help or help for a command.

This is a very simple client. Essentially this is just a little more convenient than using `curl` manually. Each invocation of this client obtains a token and then performs a single request.

This client is not meant for direct usage, but just as a helper for integrating with PDC from languages where it might be easier than performing the network requests manually.

CHAPTER 5

Script pdc

Note: Use argument `-h` or `--help` to get general help or help for a command.

This has much more user friendly user interface than `pdc_client`. A single invocation can perform multiple requests depending on what subcommand you used.

The `pdc` client supports Bash completion if argcomplete Python package is installed.

If you installed client from rpm package, the completion file `pdc.bash` has been installed to `/etc/bash_completion.d/`.

For developers or users who try to run `pdc` from source, to enable completion, run this in your terminal (assuming pdc is somewhere on path).

```
eval "$(register-python-argcomplete pdc)"
```

or put `pdc.bash` to `/etc/bash_completion.d/`.

CHAPTER 6

Release

6.1 Versioning

PDC Client versioning is based on Semantic Versioning.

And it's RPM compatible.

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner,
3. PATCH version when you make backwards-compatible bug fixing.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

1. A pre-release version MAY be denoted by appending a hyphen and an identifier immediately following the patch version.

   Identifier MUST be comprised and only with ASCII alphanumerics [0-9A-Za-z]. Identifier MUST NOT be empty. Numeric identifier MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-sprint5, 1.0.0-rc4.

2. Build metadata MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch or a dot and a series of dot separated identifiers immediately following the pre-release version.

   Identifiers MUST be comprise and only with ASCII alphanumerics [0-9A-Za-z]. Identifiers MUST NOT be empty. Build metadata SHOULD be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-12.g1234abc, 1.0.0-s5.4.g1234abc.

6.2 Release Instruction

In practice, we use *tito* to add git tag and do release including tag based on releases and current HEAD based on test releases.

Note: *tito* version >= 0.6.2, install guide refer to: *https://github.com/dgoodwin/tito*

A short instructions as:

1. Tag: `tito tag`
2. Test Build: `tito build --rpm ---offline`
3. Push: `git push origin && git push origin $TAG`
4. Release: `tito release copr-pdc/copr-pdc-test`

For each step, more detail are:

6.2.1 Tag

A new git tag need to be added before starting a new release:

```
$ tito tag
```

It will:

- bump version or release, based on which *tagger* is used, see *.tito/tito.props*;
- create an annotated git tag based on our version;
- update the spec file accordingly, generate changelog event.

For more options about *tito tag*, run *tito tag –help*.

6.2.2 Test Build

Once release tag is available, we can do some build tests including source tarball checking, and rpm building testing.

```
# generate local source tarball
$ tito build --tgz --offline

# generate local rpm build
$ tito build --rpm --offline
```

If everything goes well, you could push your commit and tag to remote, otherwise the tag need to be undo:

```
$ tito tag -u
```

Note: During developing, we could also generate test build any time, which will be based on current *HEAD* instead of latest tag.

```
# generate test builds
$ tito build --test --tgz/srpm/rpm
```

6.2.3 Push

When you're happy with your build, it's time to push commit and tag to remote.

```
$ git push origin && git push origin <your_tag>
```

6.2.4 Release To PyPI

The Python Package Index or PyPI is the official third-party software repository for the Python programming language. Release PDC Client to PyPI make it be able to pip install this for usage in other projects. pdc-client was already registered in PyPI.

If you haven't created an account in PyPI or configured PyPI in local environment, you may need:

- create your account on PyPI Live.
- contact PDC team to get PyPI pdc-client access.
- create ~/.pypirc configuration file with content:

```
[distutils]
index-servers=pypi

[pypi]
repository = https://pypi.python.org/pypi
username = your_username
password = your_password
```

Finally, you can upload your distributions to PyPI. There are two options:

1. Use twine. Twine uses only verified TLS to upload to PyPI in order to protect your credentials from theft:

   ```
   twine upload dist/*
   ```

2. **(Not recommended):** Use setuptools. This approach is covered here due to it being mentioned in other guides, but it is not recommended as it may use a plaintext HTTP or unverified HTTPS connection on some Python versions, allowing your username and password to be intercepted during transmission.

   The command could be:

   ```
   python setup.py sdist upload
   ```

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

p

Index

Symbols

G

N

O

P

S