PDC Documentation

Release 1.10.0-1.77.gaf79f73

PDC Devel Team

Apr 03, 2018

Contents

Contents:

CHAPTER 1

Using API

This page contains details about using PDC from the API user view-point.

1.1 PDC API Stability Promise

Product Definition Center promises API stability and forwards-compatibility of APIv1 from version 0.9.0. In a nutshell, this means that code you develop against PDC will continue to work with future releases. You might be required to make changes to your usage of the API when changing to a different version of the API or when you want to make use of new features.

1.1.1 What *stable* means?

In this context, stable means:

- The documented APIs will not be removed or renamed
- Arguments for APIs will not be removed or renamed
- Keys in returned JSON dictionaries will not be removed or renamed
- If new features are added to these APIs – which is quite possible – they will not break or change the meaning of existing methods. In other words, "stable" does not (necessarily) mean *complete*
- Default values of optional arguments will not change
- **Order of returned results will not change for following results:**
  - releases in releases resource API
  - composes inside compose_set in releases resource API
  - composes in composes resource API
- If, for some reason, an API declared stable must be removed or replaced, it will be declared deprecated in given version of the API and removed/replaced in future version of API

- We'll only break backwards compatibility of these APIs if a bug or security hole makes it completely unavoidable.

To make use of this stability your client code has to accept unknown keys and values in responses and ignore them if they are not recognized.

1.1.2 Exceptions

There are a few exceptions to the above stability promise. Specifically:

- APIs marked as *experimental* are not part of promise. This enables us to add new APIs and test them properly before marking them as stable
- If a security or other high-impact bug is encountered we might break stability promise. This would be used as last resort.

1.2 Authentication

By default, all the API calls, except GET requests, require authentication, though this depends on PDC server settings. If you're logged in, you can list permissions and group you're in at `/rest_api/v1/auth/current-user/`. To see who has which permissions see table at `/rest_api/perms/`.

While the web UI is authenticated using an external system such as Kerberos or SAML2, the API uses a custom authentication for performance reasons.

The expected workflow is as follows:

1. Obtain an authorization token from the API `/rest_api/v1/auth/token/obtain/`. This is one of endpoints that actually use the same authentication system as the web UI.
2. Perform requested actions using the token. It needs to be sent with the request in an HTTP header `Authorization`. With `curl`, this can be done with the `-H` flag (for example `-H "Authorization: Token XXX"`).

The token you receive from the API is tied to your user account. Currently, the token is valid indefinitely. However, if you leak it somewhere, you can manually request a new token, which will invalidate the old one. To do this, use the `/rest_api/v1/auth/token/refresh/` API.

If you access the API through one of PDC client, the authentication can be handled transparently for you.

1.3 Paging

The lists returned from the API can be quite long. They are paginated by default with pages containing 20 items.

The structure of paginated reply is JSON object with following keys:

`count` Total number of items. Essentially, this tells you how many items you would get if you got all the pages and concatenated them.

`next` URL where you can get the next page. Contains `null` on the last page.

`previous` URL where the previous page is. On the first page it contains `null`.

`results` The actual data as a JSON array.

You can control the details of the paginating by a couple query parameters. The `page` parameter specifies which page you want. With `page_size` you can set up different size of a page. The maximum allowed number of results per page is 100. Specifying anything higher has the same result as specifying 100. There is a special value of `-1` for the page size, which would turn pagination off and give all the results at once. In this case, the response is just the result array without any count or URLS.

Please be careful when turning the pagination off. If your query could return hundreds or thousands of results, consider getting the data page by page instead.

1.4 Change monitoring

Whenever a change is performed through the API, a log is created so that it is possible to find out what, when, why and by who was changed.

The changes can be viewed from the API under the `/rest_api/v1/changesets/` end-point. There is also a view on the web pages. A logged-in user can access it from the menu in top right corner.

To store the reason for the change, add HTTP header `PDC-Change-Comment`, whose value is an arbitrary string that will be stored with the change.

1.5 Override Ordering

The client can override the ordering of the results with query parameter. By default, the query parameter is named `ordering`. For example, to order releases by release_id:

http://example.com/rest_api/v1/releases/?ordering=release_id

The client may also specify reverse orderings by prefixing the field name with '-'.

For example, to reverse orderings of releases by release_id:

http://example.com/rest_api/v1/releases/?ordering=-release_id

CHAPTER 2

Bulk operations

The REST API provides a way to perform many operations in a single request. However, all these operations in a single request must work with the same collection (e.g. *release components* or *products*).

All these *bulk* operations are available on the resource list URL. Unless stated otherwise in the documentation for a particular resource, the bulk operation is implemented in terms of the standard call operating on a single item.

The bulk call is atomic – all operations will be performed or none will.

2.1 Create

The input to this call should be a list of JSON objects. The rules for the items in the list are same as when creating a single item in the collection.

The items will be created in the order in which they were specified in the request.

The response from this call will include a list of whatever would be returned by creating the resources one by one. The status code on success is `201 CREATED`.

Note that the only way in which the bulk create differs from regular create is the request data.

2.2 Delete

To delete multiple items in a collection, send a `DELETE` request to the list URL. The request should include a body which should be a list of identifiers of items to be deleted.

The items will be deleted in the order in which they were specified in the request.

The exact format of the identifier is collection specific. Generally it should be the identifier you would append to the URL to get a detail view of the item.

On success the response will be `204 NO CONTENT`.

2.3 Update

Updating multiple items is possible via the `PUT` or `PATCH` method directed to the list URL of a collection. In both cases, the request body should be a JSON object, where keys are identifiers of objects (same as for bulk delete) and their values describe desired changes.

Exact format of the changes description is resource dependant. It should have the same structure as when updating a single item.

Because JSON objects are not ordered, the order in which the items will be updated is not specified and can be different to what is specified in the request.

On success the response will have the `200 OK` status code. The response body will include a JSON object with the same structure as in the original request, only change descriptions will be replaced with whatever gets returned by the update method for single item. Note that if the requested change results in a change of the identifier, the response will still contain the old identifier with new value for the item.

2.4 Errors

If an error occurs during processing a bulk operation, all changes from the request will be aborted and no change log will be recorded. The status code of an error response depends on what went wrong.

The structure of the response body should (in case of client errors) consist of a JSON object with the following structure.:

```
{
    "detail":              <string|object>,
    "id_of_invalid_data": <string|int>,
    "invalid_data":        object
}
```

The `detail` key denotes a more precise description of the error. Its value is supplied by the single item manipulating function.

The `id_of_invalid_data` describes which part of the request caused the error. For create, it is an integer index from the request array (starting from zero), for update or delete it is the identifier.

The `invalid_data` contains the actual part of request that was invalid. It is only present when creating or updating.

Please note that the processing always stops when encountering the first error. It may be very well possible that even when the reported error is fixed, the request will fail with another error.

CHAPTER 3

Setup development environment

3.1 Source Code

```
$ git clone https://github.com/release-engineering/product-definition-center.git
```

3.2 Installation

3.2.1 Option 1: Start it on RPM

For development purposes, install following dependencies:

- python = 2.7
- python-django = 1.11
- python-ldap
- python-requests
- python-requests-kerberos
- python-mock
- kobo >= 0.4.3
- kobo-django
- djangorestframework >= 3.5.4
- django-mptt >= 0.8.6
- Markdown
- django-cors-headers >= 2.0.0
- [productmd](https://github.com/release-engineering/productmd.git)

- [patternfly1](https://copr.fedoraproject.org/coprs/patternfly/patternfly1/)
- django-filter >= 1.0.2
- python-qpid-proton

3.2.2 Option 2: Start it on virtualenv

- Install virtualenvwrapper

```
$ pip install virtualenvwrapper
```

and setup according to 'Setup' steps in `/usr/bin/virtualenvwrapper.sh`. Then create virtual environment

```
$ mkvirtualenv pdc
```

- Run the following

```
$ workon pdc
$ pip install -r requirements/devel.txt
```

to activate *pdc* virtualenv and install all the dependencies.

3.2.3 Option 3: Start it on Docker

- Install Docker: see the official installation guide for details. Generally, it might be enough to run install it with `yum` and then run it.

```
$ sudo yum install docker-engine
$ sudo service docker start
```

- Use this command to build a new image

```
$ sudo docker build -t <YOUR_NAME>/pdc <the directory your Dockerfile is located>
```

- Run the container (`:Z` flag is required to mount volumes with SELinux)

```
$ docker run -it -P -v $PWD:$PWD:Z <YOUR_NAME>/pdc python $PWD/manage.py␣
↪runserver 0.0.0.0:8000
```

- Check the address
  1. Find the address of the docker machine (127.0.0.1 –> DOCKER_HOST).
  2. Find the mapped port of your running container

```
$ sudo docker ps -l --> PORT
```

- Access it by visiting `DOCKER_HOST:PORT` in your web browser.

3.3 Customize settings

You can use the dist settings template by copying it to *settings_local.py*:

```
$ cp settings_local.py.dist settings_local.py
```

Feel free to customize your *settings_local.py*. Changes will be populated automatically. In local development environment, you may need to set `DEBUG = True` to get better error messages and comment out `ALLOWED_HOSTS` setting.

When you run PDC locally, you may not want to enable the permission checks, just uncomment `DISABLE_RESOURCE_PERMISSION_CHECK` line.

3.4 Init database

To initialize database, run:

```
$ python manage.py migrate --noinput
```

3.5 Run devel server

To run development server, run:

```
$ make run
```

For development you may find it useful to enable Django Debug Toolbar.

Related settings is documented in comment at the top of `settings_local.py.dist`.

CHAPTER 4

API Development

Each resource available on the REST API is implemented in terms of a couple objects. The main one is a `ViewSet`, which may optionally use a `Serializer` and a `FilterSet`.

This is a guide for adding new resources.

4.1 Checklist

1. Identify where to implement it: it can be part of existing application or you can create a new application. If you want a new application, use

```
$ mkdir pdc/apps/your_app
$ python manage.py startapp your_app pdc/apps/your_app
```

2. Create your models. Make sure to implement `export` method for each model that will be editable through the API.
3. Generate migrations

```
$ python manage.py makemigrations your_app
```

4. Make sure the `ViewSet` inherits from `StrictQueryParamMixin` to properly handle unknown query parameters.
5. If the resource objects can be created, updated or deleted, use `ChangeSet*` mixins (or `PDCModelViewSet` as single parent).
6. The docstring of the methods will be visible in browsable documentation. Use Markdown for formatting. See below for other helpers you can use to simplify documentation.
7. Serializer should inherit from `StrictSerializerMixin` or implement the same logic itself (report error when unknown field is specified).
8. Write test cases for both success and error paths.

4.2 Writing documentation

The browsable documentation is exported from docstrings of view set methods. It uses Markdown as a markup language. There are a couple helpers that make some things easier.

First of all, string `%(HOST_NAME)s`, `%(API_ROOT)s` expand to host name of the current server and path to the API root, respectively.

To include a link to another resource, rather than using the macros above, there is a better way:

- `$URL:resourcename:param1$` will expand to URL to that resource. Examples:
  - `$URL:release-list$` → `http://pdc.example.com/rest_api/v1/releases/`
  - `$URL:product-detail:dp$` → `http://pdc.example.com/rest_api/v1/products/dp/`
- `$LINK:resourcename:param1:param2$` will expand to clickable link to that resource. The link label will be the URL of the resource (without the host name).

To describe available query filters, use `%(FILTERS)s` macro. This expands to an unordered list with filter names and types of the value. The serializer can be described with `%(SERIALIZER)s`, which expands to a code block with JSON describing the data. For create/update actions you may need to use `%(WRITABLE_SERIALIZER)s` which excludes all read-only fields.

CHAPTER 5

PDC Model Graphs

Current PDC Model Graphs:

5.1 overview

5.2 auth

5.3 bindings

5.4 changeset

5.5 common

5.6 component

5.7 compose

5.8 contact

5.9 osbs

5.10 package

5.11 partners

5.12 release

5.13 repository

5.14 usage

CHAPTER 6

Release

6.1 Versioning

PDC 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

Thanks to fedorapeople.org and Fedora Copr, we could use *tito* to release *PDC* as a *yum* or *dnf* repo. So that user could install *PDC* packages after enable the repo.[1]

Note: Before doing any release, make sure that you have account on both sites and also make sure that you could access to your fedorapeople space[2] and have enough permissions[3] to build *PDC* in *Copr*.

You need to create a directory called *pdc_srpms/* under your fedorapeople space *public_html/* to hold all the uploaded srpms.

copr-cli will be used, installed by *sudo yum/dnf install copr-cli* and configure it.[4]

Currently there are two projects in *Copr*: *pdc* for all tag based releases and *pdc-test* for test builds. We have two release targets in *tito*, *copr-pdc* is for *pdc* in *Copr* and *copr-pdc-test* is for *pdc-test* respectively.

Request as *Builder* for projects *pdc/pdc-test* and *pdc/pdc*, wait until admin approves.

After all setup, release with *tito*:

```
$ tito release copr-pdc
# or
$ tito release copr-pdc-test
```

Go and grab a cup of tea or coffee, the release build will be come out soon

```
# test builds: `https://copr.fedoraproject.org/coprs/pdc/pdc-test/builds/`
# tag based builds: `https://copr.fedoraproject.org/coprs/pdc/pdc/builds/`
```

[1] https://fedorahosted.org/copr/wiki/HowToEnableRepo
[2] http://fedoraproject.org/wiki/Infrastructure/fedorapeople.org#Accessing_Your_fedorapeople.org_Space
[3] https://fedorahosted.org/copr/wiki/UserDocs#CanIgiveaccesstomyrepotomyteammate
[4] https://copr.fedoraproject.org/api/

CHAPTER 7

Deployment

7.1 Install via yum

```
$ yum install pdc-server
```

The RPM includes a cron job to perform daily synchronization of users with LDAP. It is installed to `/etc/cron.daily` and does not need any configuration.

7.2 Configure Django settings

```
# mv settings_local.py.dist to settings_local.py
# change database settings in /usr/lib/pythonX.Y/site-packages/pdc/settings_local.py
```

7.3 Initialize database

```
# create database
$ su - postgres
$ psql
postgres=# create database "db_name" owner "user_name";
postgres=# \q

# migrate database
$ django-admin migrate --settings=pdc.settings --noinput
```

7.4 Collect static

```
$ django-admin collectstatic --settings=pdc.settings
```

7.5 Config apache

replace `PDC_HOSTNAME` with server's hostname in `/etc/httpd/conf.d/pdc.conf`

7.6 Running PDC behind reverse proxy

To make sure documentation links work correctly when PDC is running behind proxy, add `USE_X_FORWARDED_HOST = True` in *setting_local.py* file.

The link to Django documentation: https://docs.djangoproject.com/en/1.9/ref/settings/#use-x-forwarded-host .

CHAPTER 8

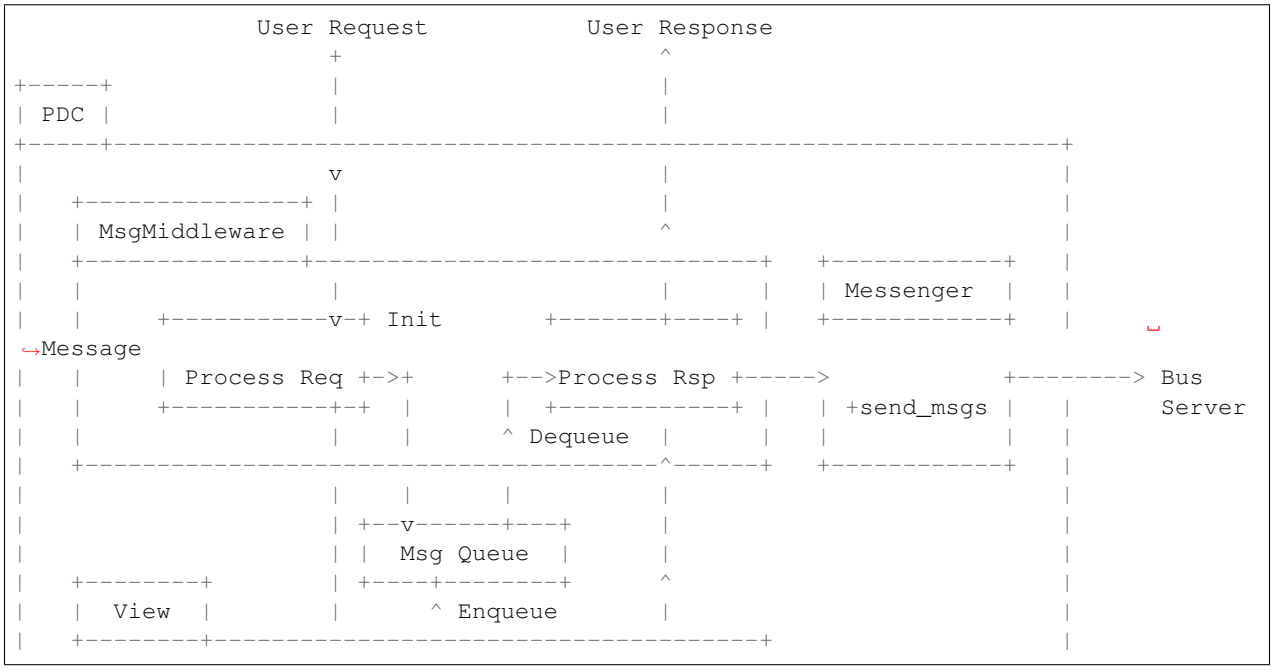
Messaging

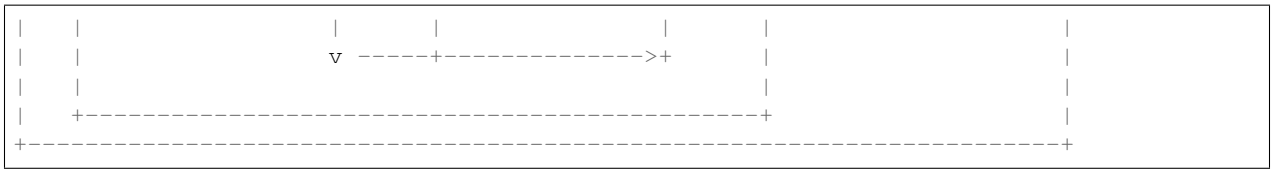
8.1 Overview

Messaging enables PDC to send out useful informations to message bus so that other systems can subscribe and deal with them.

Current design is based on Django Middleware system, by implementing our MsgMiddleware, we could initialize a message queue for every incoming request during the `process_request`, and push generated messages into it while processing the request in the view, if no error occurs, before response get returned to user, we pop out all the messages and invoke the configured backend to send them to the Message Bus.

PDC Messaging Overview and Dataflow:





8.2 Supported Messengers

There are two messengers that PDC provides, you should choose which one to use according to your messaging infrastructure. Also you could write your own messenger based on your own requirements.

Once you got the answer what left is to configure `MESSAGE_BUS` item in the settings file accordingly. The key determining which backend to use is `BACKEND`, whose value should be a dotted module path pointing to the messenger implementation.

Following is the brief introduction of each messenger along with their settings examples, it will help you to know which one to use and how to configure it as well.

1. `DummyMessenger` (Default)

   If you do not need to send messages out, you could just omit the configuration out entirely. The messages will be logged, but not sent anyhere.

2. `FedmsgMessenger`

   If you want to send PDC messages to Fedora Infrastructure Message Bus, you should choose `FedmsgMessenger`.

   `fedmsg` needs to be installed.

   ```
   MESSAGE_BUS = {
       'BACKEND': 'pdc.apps.messaging.backends.fedmsg.FedmsgMessenger',
   }
   ```

3. `RHMsgMessenger`

   This messenger is built on top of `python-rhmsg` library and is used to send notification to Red Hat's internal message bus.

   To use it, the configuration should specify a list of message brokers, a path to a certificate used for authentication and a topic prefix to use for all messages. See the example below for details.

   The file pointed to by `CERTIFICATE` should contain both the public certificate and the private key in PEM format.

   ```
   MESSAGE_BUS = {
       'BACKEND': 'pdc.apps.messaging.backends.rhmsg.RHMsgMessenger',
       # The brokers will be tried in random order. If connection can not
       # be made in a given timeout, another URL is tried.
       'URLS': [
           'amqps://broker01.example.com:5671',
           'amqps://broker02.example.com:5671',
       ],
       # How long to wait for each broker. The default is 60 seconds.
       'CONNECTION_TIMEOUT': 5,

       'CERTIFICATE': '/etc/pdc/certificate.pem',
       'CACERT': '/etc/pdc/authoritycert.crt',
   ```

```
    # This value is prepended to topic of all messages.
    'TOPIC_PREFIX': 'VirtualTopic.eng.pdc',
}
```

8.3 To Be Improved

- Better Error handling
- Message structure refine
- Transaction based Messaging
- Persistent messages that failed to send out
- Non-blocking

CHAPTER 9

Indices and tables

- genindex
- search