woohoo pDNS

Release 2019.1.3

Andreas Scherrer

Aug 21, 2019

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Timing

A complete installation of woohoo pDNS will require about 45 minutes to complete for an experienced admin when a relational database server is already available.

This does not include making the source data available (e.g. copying log files from one machine to another or similar tasks).

1.2 Requirements

woohoo pDNS is a Python 3 project, therefore you need Python 3 to run it.

Also, a relational database is required. I use PostgreSQL but anything that SQLAlchemy can handle should do. For testing, sqlite is just fine; do not use it in production though because of limited support for timezones in datetime fields.

The RESTful API is served by Gunicorn. It is strongly suggested to have a reverse proxy (like Nginx, lighttpd, Apache, . . .) in front of it.

1.3 Overview

The installation will consist of the following steps:

1) create a virtual environment (Python 3)
2) install woohoo pDNS and dependencies
3) configure access to the relational database
4) set up the configuration in the reverse proxy
5) configure Gunicorn to serve the RESTful API
6) [OPTIONAL] configure automatic loading of new pDNS data

1.4 Installing

1.4.1 The virtual environment

Any way of virtualising the Python environment can be used to run woohoo pDNS. For this guide we use Python's integrated `venv` method.

In case you are wondering: in production, I use Miniconda.

Caution: woohoo pDNS has pinned its dependencies! This means that the exact version is specified in `requirements.txt` for all dependencies. This might have undesired side effects when installing in a non-empty environment where one of the packages woohoo pDNS depends on is already installed.

So, go ahead and choose a suitable home for your installation of woohoo pDNS. For Linux/*BSD systems, something under `/usr/local` might make sense (e.g. `/usr/local/opt/woohoo-pdns`).

Once you have decided on the location and created a folder for woohoo pDNS, create a new virtual environment like this:

```
$ python -m venv .pdns
```

This will create a folder named `.pdns` in the current directory and this folder will hold your virtual environment of the same name.

Note: on a Mac of mine, creating the virtual environment like this failed with an error like:

```
Error: Command '['/Users/<username>/tmp/.pdns/bin/python', '-Im', 'ensurepip', '--
↪upgrade', '--default-pip']' returned non-zero exit status 1.
```

which can be fixed by following advice found on Stackoverflow:

```
$ python -m venv --without-pip .pdns
$ source .pdns/bin/activate
$ curl https://bootstrap.pypa.io/get-pip.py | python
$ deactivate
$ source .pdns/bin/activate
```

1.4.2 Install woohoo pDNS and dependencies

Go ahead and activate the new environment if not already done (your shell prompt should change):

```
$ source .pdns/bin/activate
```

You should now populate this new virtual environment with woohoo pDNS and the required dependencies:

```
(.pdns)$ pip install woohoo-pdns
```

or install it from source:

```
(.pdns)$ git clone https://gitlab.com/scherand/woohoo-pdns
(.pdns)$ cd woohoo-pdns
(.pdns)$ python setup.py install
(.pdns)$ pip install -r requirements.txt
```

You should now be able to run the `pdns` command:

```
(.pdns)$ pnds -h
```

Create the configuration file

To properly run the pdns command, you will have to provide a config file with the following information/format (`-f` or `--config-file` CLI switch):

```
[DB]
conn_str = "sqlite:///demo.db"

[LOAD]
loader_class = "woohoo_pdns.load.SilkFileImporter"
data_timezone = "UTC"
```

The values shown here are the default values that will be used if you do *not* provide a config file.

1.4.3 Configure access to the relational database

This step depends on the database you want to use and the administrative processes you have in place for managing (relational) databases and access to them.

woohoo pDNS needs access to a database with permissions to create tables as well as read and write data.

For PostgreSQL the process is as follows:

```
[root@database:~]# su - postgres
$ createuser --interactive
Enter name of role to add: pdns
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
$ createdb pdns
$ psql
postgres=# ALTER USER pdns WITH ENCRYPTED PASSWORD '...';
postgres=# GRANT ALL PRIVILEGES ON DATABASE pdns to pdns;
```

1.4.4 Set up the configuration in the reverse proxy

Again, the exact steps depend on the reverse proxy software you use and the administrative processes around it. Assuming you have all the required permissions and want to use lighttpd, the configuration should look about as follows:

```
$HTTP["host"] =~ "^pdns.example.com$" {
    $HTTP["url"] =~ "^/api/" {
        proxy.server = ( "" => ( (
            "host" => "localhost",
            "port" => 5001
        ) ) )
    }
}
```

1.4.5 Configure Gunicorn to serve the RESTful API

The API is served by a Flask application (WSGI application) that lives in `woohoo_pdns.api` and is served by Gunicorn. To fire it up, you can use many different ways. For example, a startup script.

Consider using a dedicated user for Gunicorn.

You **must** provide the name of a config file via an environment variable called `WOOHOO_PDNS_API_SETTINGS`. That file should contain the following options. (In the example below, the file is called `pdns_api_conf.py`.) If only a filename is specified, the file is expected to be in a folder called `instance` in the directory you are starting flask from.

```
SECRET_KEY = "snakeoil"
DATABASE = "sqlite:///demo.db"
API_KEYS = [
    "IXsA7uRnxR4xek4JDEG5vk2oGjTYDSqaoKLRQLVjV2s3kw0bbv49qrgAT7Bk3g2K",
    "jLHKK0AIk1l6r3W8SAJj4Lh0v2a27JGbSSd406mr0u5FNrJn6RLWQ5m6qPYXT0d5",
]
```

The options shown above are the default values that are used if the file referenced in the `WOOHOO_PDNS_API_SETTINGS` environment variable does not set them.

You can use whatever you like for the `SECRET_KEY`; it is a Flask thing, see *`woohoo_pdns.api.config.DefaultSettings.SECRET_KEY`*.

The `DATABASE` option specifies the connection string to the relational database (this is forwarded 'as is' to SQLAlchemy).

The list of `API_KEYS` specifies all strings that will be accepted as keys for API access.

Note: The API keys can be any string, but it is suggested to create a random character sequence using something like the following command (inspired by a gist by earthgecko):

```
$ cat /dev/urandom | base64 | tr -dc 'a-zA-Z0-9' | fold -w 64 | head -1
```

The following outlines the FreeBSD rc.d script (`/usr/local/etc/rc.d/pdns-api-gunicorn`) I use for this purpose (inspired by a thread in the FreeBSD forums):

```
#! /bin/sh

# PROVIDE: pdns_api_gunicorn
# REQUIRE: DAEMON
# KEYWORD: shutdown

#
# Add the following lines to /etc/rc.conf to enable the woohoo pDNS API:
#
#pdns_api_gunicorn_enable="YES"

. /etc/rc.subr

name="pdns_api_gunicorn"
rcvar="${name}_enable"
start_cmd="${name}_start"
stop_cmd="${name}_stop"
pidfile="/var/run/${name}.pid"
procname="daemon:"
gip="localhost"
gport="5001"
```

(continues on next page)

(continued from previous page)

```
pdns_api_gunicorn_start(){
    chdir /usr/local/opt/woohoo-pdns
    . /root/.virtualenvs/pdns/bin/activate
    LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8 FLASK_ENV=production WOOHOO_PDNS_API_SETTINGS=
↪"pdns_api_conf.py" daemon -r -S -P ${pidfile} -T pdns-api-gunicorn -u root /root/.
↪virtualenvs/pdns/bin/gunicorn --workers 3 --bind ${gip}:${gport} "woohoo_pdns.
↪api:create_app()"
}

pdns_api_gunicorn_stop(){
    if [ -f ${pidfile} ]; then
        echo -n "Stopping services: ${name}"
        # MUST send TERM signal (not e.g. INT) to work properly with '-P' switch
        # check daemon(8) for details
        kill -s TERM $(cat ${pidfile})
        if [ -f ${gsocket} ]; then
            rm -f ${gsocket}
        fi
        echo "."
    else
        echo "It appears ${name} is not running."
    fi
}

load_rc_config ${name}
# this sets the default 'enable' (to no)
: ${pdns_api_gunicorn_enable:="no"}
run_rc_command "$1"
```

1.4.6 Automatic loading of additional data

I run the following script every three minutes via a cron job:

```
*/3 * * * *  /usr/local/bin/woohoo-pdns-load.sh 2>&1 | /usr/bin/logger -t woohoo-pdns
```

`/usr/local/bin/woohoo-pdns-load.sh`:

```
#!/usr/local/bin/bash

. /root/.virtualenvs/pdns/bin/activate
pdns -f /usr/local/etc/woohoo-pdns/pdns.conf load -p "dns.*.txt" /var/spool/silk/dns

exit 0
```

New files matching the glob pattern `dns.*.txt` in `/var/spool/silk/dns/` will be read into the database like this. After they are processed, they are renamed by appending `.1` to the filename so they are not read again.

I have another ‘cron job’ (it is actually a job for FreeBSD’s `periodic`) that cleans out old files from `/var/spool/silk/dns/` – well – periodically.

It lives in `/usr/local/etc/periodic/daily/405.woohoo-pdns-cleanup` and looks as follows:

```
#!/bin/sh

```

(continues on next page)

(continued from previous page)

```
cleanup_1_files() {
    local rc

    /usr/bin/find /var/spool/silk/dns/ -name "*.1" -type f -maxdepth 1 -mmin +60 -
↪delete

    rc=$?
    return $rc
}

cleanup_1_files

exit $rc
```

CHAPTER

TWO

USING WOOHOO PDNS

There are three ways you can use your installation of woohoo pDNS.

- via the command line interface (CLI)
- via the REST interface
- via the Python API

We well show each of the usage types in turn here.

2.1 Command Line Interface (CLI)

The CLI is accessible via the `pdns` command. It has a help switch (`-h/--help`):

```
$ pdns -h
usage: pdns [-h] [-f CONFIG_FILE] [-q | -v | -d] {load,export,query} ...

CLI for woohoo pDNS.

optional arguments:
  -h, --help            show this help message and exit
  -f CONFIG_FILE, --config-file CONFIG_FILE
                        The config file to use, must contain the connection
                        string for the database, e.g. 'conn_str =
                        sqlite:///test.db' in a section called '[DB]'
  -q, --quiet           be quiet (only output critical messages)
  -v, --verbose         be verbose
  -d, --debug           be as verbose as you can

subcommands:
  available subcommands

  {load,export,query}   see help for respective sub-command
    load                load data into the pDNS database
    export              export data to a JSON file
    query               query the database (returns JSON)
```

So you could for example load files created by `supermediator` using the following command:

```
$ pdns -f pdns.conf load -p "dns.*.txt" /var/spool/silk/dns
```

The contents of `pdns.conf` is described in the Installation guide under *Create the configuration file*.

Or you could query the database as follows:

```
$ pdns -f pdns.conf query "*example.com"
{
    "hitcount": 2,
    "rdata": "example.com",
    "rrname": "example.com",
    "rrtype": 6,
    "time_first": 1479937407.0,
    "time_last": 1479937407.0
}
```

2.2 RESTful

Note: You cannot *add* data to the database via the RESTful API.

Once `gunicorn` sering the RESTful API (and the reverse proxy protecting it) is up and running, you can query it using `curl` as follows:

```
curl -D - -H "Authorization: <your_api_token>" <server_ip>/api/count
curl -D - -H "Authorization: <your_api_token>" <server_ip>/api/recent
curl -D - -H "Authorization: <your_api_token>" <server_ip>/api/q/www.example.com
```

2.3 Python API

Here is a sample Python API session:

```
import logging
import configparser
logging.basicConfig(level=logging.DEBUG)

from woohoo_pdns.pdns import Database

config_f = configparser.ConfigParser()
config_f.read("my.conf")
conn_str = config_f["DB"]["conn_str"]
d = Database(conn_str)

r = d.add_record(1, "foo", "bar.")
d.find_record(1, "foo")
d.add_record(1, "foo", "bar", num_hits=40)
d.find_record(1, "foo")

d.query("gmail-imap.l.google.com")
d.query("2a00:1450:4001:080b::200a")
d.query("127.0.0.1")
d.query("meteoswiss-app.ch")
d.query("*example.com")
```

CHAPTER

THREE

WOOHOO PDNS' TODO LIST OR WHISHLIST

3.1 Some things I am considering to add

- log some statistics showing the activity in the DB
- implement a Kafka source (`woohoo_pdns.load.Source`)

3.2 Some things I am looking for from the community

- Init scripts (especially for Linux)
- Reverse proxy configurations for other webservers than lighttpd
- General tips and tricks to run Python web applications (e.g. logging)

CHAPTER FOUR

CONTRIBUTING

To build woohoo pDNS (and the documentation), three additional dependencies exist:

```
pip-tools
nose
sphinx-rtd-theme
```

Caution: woohoo pDNS has pinned its dependencies! This means that the exact version is specified in `dev-requirements.txt` for all dependencies. This might have undesired side effects when installing in a non-empty environment where one of the packages woohoo pDNS depends on is already installed.

You can easily install them using the following pip command in your development environment:

```
$ pip install -r dev-requirements.txt
```

To build the documentation after cloning the repository, run the following command in the `woohoo_pdns/docs` directory:

```
$ make html
```

Note: do *not* run:

```
$ sphinx-quickstart
```

To run the tests, issue the following command:

```
$ python setup.py test
```

And to see test coverage:

```
$ pytest [--cov-report html] --cov=woohoo_pdns woohoo_pdns/tests/
```

4.1 Managing dependencies

Following the advice of people with (much) more experience in that field (namely Vincent Driessen and Hynek Schlawack) woohoo pDNS pins its dependencies.

The tool used is pip-tools, for the runtime dependencies in Hash-Checking Mode, and here's how.

4.1.1 Runtime dependencies

Dependencies required to *run* woohoo pDNS are listed in the `install_requires` variable in `setup.py`:

```
setup(
    <snip>
    install_requires = [
        "alembic",
        "flask",
        <snap>
    ]
)
```

If you want to add a new (run time) dependency for woohoo pDNS, this is the place to do so.

4.1.2 Build dependencies

Dependencies required to *develop* woohoo pDNS are listed in the `dev-requirements.in` file:

```
pip-tools
...
```

4.1.3 Using `pip-tools` for woohoo pDNS

To *generate a requirements.txt file* (i.e. a `requirements.txt` file that listing the runtime dependencies), run the following command (you have `pip-tools` installed, right?):

```
$ pip-compile --generate-hashes
```

This will *overwrite* the current requirements.txt file with the most recent version available on PiPI for every package and will *add new dependencies* also.

To check if there are newer versions of dependencies available in PyPI, use the following command:

```
$ pip-compile --upgrade --generate-hashes
```

This will *overwrite* the current requirements.txt file with the most recent version available on PiPI for every package. It will *not* add new dependencies though.

Note: `pip-compile` has a `dry-run` command line switch.

To *generate the ‘‘dev-requirements.txt‘‘ file* (i.e. a file listing the build dependencies), run the following command:

```
$ pip-compile --allow-unsafe --output-file=dev-requirements.txt dev-requirements.in
```

This will *overwrite* the current `dev-requirements.txt` file with the most recent version available on PiPI for every package and will *add new dependencies* also.

To check if there are newer versions of build dependencies available in PyPI, use the following command:

```
$ pip-compile --upgrade --allow-unsafe --output-file=dev-requirements.txt dev-
↪requirements.in
```

This will *overwrite* the current `dev-requirements.txt` file with the most recent version available on PiPI for every package. It will *not* add new dependencies though.

References:

- Pin Your Packages
- Better Package Management
- Python Application Dependency Management in 2018
- pip-tools (GitHub)

4.2 Implementing an Importer

I consider the need for a custom `woohoo_pdns.load.Importer` the most likely scenario of extending woohoo pDNS. Therefore this process is extensively documented here.

4.2.1 Overview

While the Importer is the workhorse of the data loading, it relies on another component called `woohoo_pdns.load.Source` to provide one record that should be loaded at a time.

There are currently two types of sources implemented: both read from files, but one just reads one line after the other (skipping empty lines) while the other expects to read YAML documents and therefore keeps reading until the YAML document separator (`---`) is encountered (or the file ends).

The former is `woohoo_pdns.load.SingleLineFileSource` while the latter is `woohoo_pdns.load.YamlFileSource`. Because both sources do read data from files on disk, they are both subclasses of `woohoo_pdns.load.FileSource`.

4.2.2 Source

A custom/new source is only required if the existing sources do not cover your needs. Otherwise, just writing an Importer is enough.

Requirements

If a new Source is implemented, it should subclass `woohoo_pdns.load.Source`.

Sources must be context managers (i.e. be able to be used with `with`) and must have a method called `woohoo_pdns.load.Source.get_next_record()` that does not take any argument and returns a string. That string should be something the Importer can then work with.

In addition, they must implement the `woohoo_pdns.load.Source.state` property which allows the Importer to retrieve and restore the source's state between batches of data loading.

4.2.3 The Importer subclass

Importers must be subclasses of `woohoo_pdns.load.Importer`. There are two important methods that every Importer must provide:

- `woohoo_pdns.load.Importer._tokenise_record()`
- `woohoo_pdns.load.Importer._parse_tokenised_record()`

The first one is called for every 'raw' record (i.e. whatever is returned by the Source's `woohoo_pdns.load.Source.get_next_record()`) and must return a list of `woohoo_pdns.util.record_data` named tuples. This method *can* filter the record by returning an empty list.

The return value is a list because (depending on the source) a single 'raw' entry can lead to multiple records (e.g. when a query has multiple responses).

The second function is called for every entry in the list returned by `_tokenise_record`. It is mainly meant to 'polish' the entries, for example by parsing dates, etc.

Why this complexity?

The main reason for the differentiation between the two steps in loading data is that the second might depend on information that is only available after at least one record was read from the source (per batch).

Imagine for example that the exact format of the dates (timestamps) is unknown but consistent within one batch.

In a situation like this, `_tokenise_record` would probably not be concerned with the date format. But `_parse_tokenised_record` would have to (re-)determine the format for every single record, which would be inefficient.

That is why there are two more methods that *can* be implemented in an Importer:

- `woohoo_pdns.load.Importer._inspect_raw_record()`
- `woohoo_pdns.load.Importer._inspect_tokenised_record()`

These methods are called for every *first* record of a batch. In the `_inspect_tokenised_record` method, the Importer could establish the timestamp format which could then be used for all remaining records of the batch.

Similar, `_inspect_raw_record` could be used to do an operation on the first raw record of a batch, if required .

CHAPTER FIVE

SUPPORT

If you need to get help with woohoo pDNS feel free to open an issue on Gitlab and I will do my best to help out.

Please understand however that this currently is a private project run in my free time and that I can only spend as much time on it as I can.

Be assured: I will come back to you; just maybe not right now?

CHAPTER

SIX

WOOHOO_PDNS

6.1 woohoo_pdns package

6.1.1 Subpackages

woohoo_pdns.api package

Submodules

woohoo_pdns.api.api module

`woohoo_pdns.api.api.`**`count`**`()`

The method supporting the count API endpoint. It just returns the number of records in the database.

Returns The number of entries in the database (as string).

`woohoo_pdns.api.api.`**`most_recent`**`()`

The method supporting the recent API endpoint. It returns the most recent entry from the database.

For example like this:

```
{
    "hitcount": 56,
    "time_first": 1559244767.913,
    "time_last": 1559245313.506,
    "rrtype": 1,
    "rrname": "prod-cc-asn-20190411-1321-nlb-19436c10e4427871.elb.us-east-1.
↪amazonaws.com",
    "rdata": "3.208.62.22"
}
```

`woohoo_pdns.api.api.`**`query`**`(`*q*`)`

The method supporting the query API endpoint.

Parameters **q** (*str*) – The term to search for (use '*' as wildcard).

Returns

A JSON structure compatible with the Passive DNS - Common Output Format.

An example:

```
[
    {
        "hitcount": 7,
        "time_first": 1559245077.432,
        "time_last": 1559245077.432,
        "rrtype": 5,
        "rrname": "www.icloud.com.edgekey.net",
        "rdata": "e4478.a.akamaiedge.net."
    }
]
```

`woohoo_pdns.api.api.`**`verify_password`**(*username*, *password*)

Check if a valid API key was provided.

Called by `flask_httpauth` when authentication is required. As woohoo pDNS is 'misusing' `flask_httpauth` to avoid reinventing the wheel, `username` and `password` will always be empty (we do *not* use basic authentication).

The API key must be provided in a header called `Authorization` and have the following format:

```
"Authorization: <API key as configured in config file>"
```

Parameters

- **`username`** (*`str`*) – Ignored (would be the username for basic authentication).
- **`password`** (*`str`*) – Ignored (would be the password for basic authentication).

woohoo_pdns.api.config module

`class` `woohoo_pdns.api.config.`**`DefaultSettings`**

Bases: `object`

The default configuration of the API just demonstrates the available options.

`API_KEYS = ['IXsA7uRnxR4xek4JDEG5vk2oGjTYDSqaoKLRQLVjV2s3kw0bbv49qrgAT7Bk3g2K', 'jLHKK`

The complete list of available/valid API keys.

`DATABASE = 'sqlite:///demo.db'`

The connection string to be used by SQLAlchemy.

`SECRET_KEY = 'snakeoil'`

Flask uses a secret key to encrypt things that sould be tamper proof (for example the Session object).

`__dict__ = mappingproxy({'__module__': 'woohoo_pdns.api.config', '__doc__': 'The def`

`__module__ = 'woohoo_pdns.api.config'`

`__weakref__`

list of weak references to the object (if defined)

woohoo_pdns.api.db module

`woohoo_pdns.api.db.`**`close_db`**(*e=None*)

Close the *`woohoo_pdns.pdns.Database`* that is present in Flask's global state.

`woohoo_pdns.api.db.`**`get_db`**()

Provide access to a single *`woohoo_pdns.pdns.Database`* for all API endpoints.

`woohoo_pdns.api.db.`**`init_app`**(*app*)
Called from the Flask app's `create_app()` and used to register the teardown method (`close_db()`).

Module contents

`woohoo_pdns.api.`**`create_app`**(*test_config=None*)
A Flask specific method that is called automagically.

6.1.2 Submodules

6.1.3 woohoo_pdns.load module

`class` `woohoo_pdns.load.`**`DNSLogFileImporter`**(*source_name*, ***kwargs*)
Bases: *`woohoo_pdns.load.FileImporter`*

Importer capable of reading a different source file format (JSON based).

`__init__`(*source_name*, ***kwargs*)
To correctly initialise a file source a config dict must be supplied (see 'cfg' argument documentation).

Parameters

- **`source_name`** (*`str`*) – Either the name of a file to be read or the name of a directory to scan for files to load.
- **`cfg`** (*`dict`*) – A config dictionary that contains the following two keys: * file_pattern (str): The glob pattern to use when reading files from a directory. * rename (bool): Whether or not files should be renamed (by appending '.1') after they are read.

`__module__ = 'woohoo_pdns.load'`

`_parse_tokenised_record`(*tokenised_rec*)
Convert unix timestamps into aware datetime objects and convert string-type rrtype into their integer based pendants.

Parameters **`tokenised_rec`** (*`record_data`*) – The record_data as tokenised by *`_tokenise_record()`*

Returns A single element list of record_data named tuple.

`_tokenise_record`(*rec*)
Split a line into tokens:

```
{"rrclass": "IN", "ttl": 3600, "timestamp": "1562845812", "rrtype": "PTR",
↪"rrname": "24.227.156.213.in-addr.arpa.", "rdata": "mx2.mammut.ch.", "sensor
↪": 37690}
```

becomes:

```
tokens[0] = "1562845812"                        # first_seen
tokens[1] = "1562845812"                        # last_seen
tokens[2] = "PTR"                               # DNS type
tokens[3] = "24.227.156.213.in-addr.arpa."      # rrname
tokens[4] = "1"                                 # hitcount
tokens[5] = "mx2.mammut.ch."                    # rdata
```

respectively:

```
entry.first_seen
entry.last_seen
entry.rrtype
entry.rrname
entry.hitcount
entry.rdata
```

Parameters **rec** (*str*) – A record returned from the source object.

Returns A single entry list of record_data named tuple.

class woohoo_pdns.load.**DNSTapFileImporter**(*source_name*, ***kwargs*)

Bases: *woohoo_pdns.load.FileImporter*

An importer capable of reading YAML based dnstap log files.

__init__(*source_name*, ***kwargs*)

To correctly initialise a file source a config dict must be supplied (see 'cfg' argument documentation).

Parameters

- **source_name** (*str*) – Either the name of a file to be read or the name of a directory to scan for files to load.
- **cfg** (*dict*) – A config dictionary that contains the following two keys: * file_pattern (str): The glob pattern to use when reading files from a directory. * rename (bool): Whether or not files should be renamed (by appending '.1') after they are read.

__module__ = 'woohoo_pdns.load'

_parse_tokenised_record(*tokenised_rec*)

Loop through all answers in the record and turn the datetimes into aware objects (using the default time-zone).

Parameters **tokenised_rec** (*record_data*) – The record_data as tokenised by *_tokenise_record()*

Returns A list of record_data named tuple.

_tokenise_record(*rec*)

Extract from YAML document:

```
type: MESSAGE
identity: dns.host.example.com
version: BIND 9.11.3-RedHat-9.11.3-6.el7.centos
message:
  type: RESOLVER_RESPONSE
  message_size: 89b
  socket_family: INET6
  socket_protocol: UDP
  query_address: 203.0.113.56
  response_address: 203.0.113.53
  query_port: 49824
  response_port: 53
  response_message_data:
    opcode: QUERY
    status: NOERROR
    id:  44174
    flags: qr aa
    QUESTION: 1
```

(continues on next page)

(continued from previous page)

```
    ANSWER: 2
    AUTHORITY: 0
    ADDITIONAL: 0
    QUESTION_SECTION:
      - clients6.google.com. IN AAAA
    ANSWER_SECTION:
      - clients6.google.com. 300 IN CNAME clients.l.google.com.
      - clients.l.google.com. 300 IN AAAA 2a00:1450:4002:807::200e
  response_message: |
    ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id:  44174
    ;; flags: qr aa    ; QUESTION: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0
    ;; QUESTION SECTION:
    ;clients6.google.com.       IN  AAAA

    ;; ANSWER SECTION:
    clients6.google.com.    300 IN  CNAME   clients.l.google.com.
    clients.l.google.com.   300 IN  AAAA    2a00:1450:4002:807::200e
```

becomes:

```
tokens[0] = "2018-06-18T19:22:56Z"    # first_seen
tokens[1] = "2018-06-18T19:22:56Z"    # last_seen
tokens[2] = "CNAME"                   # DNS type
tokens[3] = "clients6.google.com."    # rrname
tokens[4] = "1"                       # hitcount
tokens[5] = "clients.l.google.com."   # rdata
```

respectively:

```
entry.first_seen
entry.last_seen
entry.rrtype
entry.rrname
entry.hitcount
entry.rdata
```

Parameters **rec** (*str*) – A record (YAML document as string) returned from the source object.

Returns A list of record_data named tuple.

class woohoo_pdns.load.**FileImporter**(*source_name*, *cfg=None*, ***kwargs*)

Bases: *woohoo_pdns.load.Importer*

An abstract class to handle loading data from files.

The ‘source_name’ can be a filename or a directory name on disk. If it is a file name, that file will be read. If it is a directory, all files matching the glob pattern in cfg[“file_pattern”] will be read. An exception will be thrown if the file (or directory) does not exist).

Note: Errors will be written to a file called like the source file, but with ‘_err’ in the name (if ‘source_name’ is a file) or to a file in the parent directory of the directory to load files from, also with an ‘_err’ in the name, if ‘source_name’ is a directory.

Throws: FileNotFoundException if ‘source_name’ does not exist.

`__init__`(*source_name*, *cfg=None*, ***kwargs*)
To correctly initialise a file source a config dict must be supplied (see ‘cfg’ argument documentation).

Parameters

- **`source_name`** (*`str`*) – Either the name of a file to be read or the name of a directory to scan for files to load.
- **`cfg`** (*`dict`*) – A config dictionary that contains the following two keys: * file_pattern (str): The glob pattern to use when reading files from a directory. * rename (bool): Whether or not files should be renamed (by appending ‘.1’) after they are read.

`__module__ = 'woohoo_pdns.load'`

`_parse_tokenised_record`(*tokenised_rec*)
After a record was tokenised, it is passed to this method for parsing (e.g. turn a unix timestamp into a datetime, or similar).

Note: This method must be implemened by the concrete subclasses of *`Importer`*.

Parameters **`tokenised_rec`** (*`record_data`*) – The record as tokenised by *`_tokenise_record()`*.

Returns A record_data named tuple representing the final record to load. The importer also works if this method or returns None (i.e. nothing is loaded in to the database and the loading process continues) but the record is still considered ‘loaded’ by the statistics.

`_tokenise_record`(*rec*)
After a raw record is read from the source object, this method is called and passed the raw record to split it into the parts required for a pDNS record.

Note: This method must be implemened by the concrete subclasses of *`Importer`*.

Parameters

- **`rec`** (*`str`*) – The record as it was returned by the source object. This string must now be splitted into the
- **`parts of a record_data named tuple.`** (*`different`*) –

Returns A record_data named tuple that represents the record to load or None if the record could not be parsed (or should be ignored).

`class` `woohoo_pdns.load.`**`FileSource`**(*config*, ***kwargs*)
Bases: *`woohoo_pdns.load.Source`*

A source that reads data from files on disk.

This source can either read a single file or scan a directory for files that match a glob pattern and process all matching files from the given directory. If the `filename` passed in is a file, this file will be processed. If `filename` is a directory, the glob pattern in `file_pattern` will be used to find files to process in that directory.

If the optional configuration option `rename` is set to true (the default), *`RENAME_APPENDIX`* will be appended to the current file name after processing.

Note: The config dictionary (`config`) must contain the following keys:

- filename

And the following keys are optional in the `config` dictionary:

- file_pattern
- rename

`RENAME_APPENDIX = '1'`
If files should be renamed after processing, this is what is appended to the current filename.

`__enter__`()

`__exit__`(*exc_type*, *exc_val*, *exc_tb*)

`__init__`(*config*, ***kwargs*)

Parameters

- **`config`** (*`dict`*) – A dictionary that can hold data the source requires to configure itself.
- **`kwargs`** (*`kwargs`*) – These are mainly to make this a "cooperative class" according to super() considered super.

`__module__ = 'woohoo_pdns.load'`

`__str__`()
Return str(self).

`_open_next_file`()
Try to open the next file to process.

First, the currently open file will be closed and renamed, if requested. After this, the next file in the list is opened (if any).

Raises **`IndexError`** –

Returns Nothing.

`get_next_record`()
This method is called by the importer whenever it is ready to load the next record. What is returned will be passed into *`Importer._tokenise_record()`*.

Note: Subclasses must implement this method as it is not implemented here.

Returns A raw record from the source as string.

`property state`
Return a dictionary that describes the current state of the source.

The setter of this property expects a dictionary that was created by this getter and then restores the state of the source to what it was when `state` was retrieved.

Returns A dictionary containing the current file list `file_list` (list of all files pending processing, excluding the current file), the name of the currently being processed file `file_name` and the offset (index) into the currently being processed file (as retrieved by `tell()`).

class `woohoo_pdns.load.`**`Importer`**(*source_name*, *data_timezone='UTC'*, *strict=False*, ***kwargs*)

Bases: `object`

Importers are used to import new data into the pDNS database.

This is the super class for all importers. Different importers can import data from different sources. If no importer for a specific source is available, woohoo pDNS tries to make it simple to write a new importer for that particular source (format).

The main method of an importer is `load_batch()`. This method reads up to 'batch_size' records from the source, processes them into a list of record_data named tuples, adds some statistics and returns it.

To access the source data it uses a `Source` object. This object's job is to provide a single source record at a time to the importer. This can mean reading one or several lines from a file or a record from a Kafka topic or whatever produces a source record. The importer then processes this record (possibly into multiple entries, for example if the source record contained a single query that produced multiple answers).

This base class handles the fetching of records from the source (up to a maximum of batch_size), calling the respective hooks (`_inspect_raw_record()`, `_inspect_tokenised_record()`, `_tokenise_record()` and `_parse_tokenised_record()`) which implement the actual logic for the importer (i.e. these are the methods that must be overridden in the child classes), minimal cleansing of the data and handling errors (including writing an error logfile).

`IGNORE_TYPES = [0]`

DNS types that we want to ignore completely (0 for example does not exist)

`ILLEGAL_CHARS = ['/', '\\', '&', ':']`

If any of these characters is present in `rname` the record will not be loaded as these characters are not expected in `rrname` (they can, however, be present in `rdata`, for example in TXT records).

**`__dict__ = mappingproxy({'__module__': 'woohoo_pdns.load', '__doc__': "\n Importers`

`__init__`(*source_name*, *data_timezone='UTC'*, *strict=False*, ***kwargs*)

Constructor for an importer.

Parameters

- **`source_name`** (*`str`*) – A name that is passed to the source; can be a file name or directory name for a `FileSource` or, for a hypothetical KafkaSource, it could be the name of the Kafka topic to use.
- **`data_timezone`** (*`str`*) – The name of the timezone that should be used if the source data does not provide the timezone for the dates and times (first_seen, last_seen).
- **`strict`** (*`bool`*) – If set to true, the importer will throw an exception if something 'odd' is encountered in in the source data. If it is set to false, the importer will write an entry in the error log and continue loading data.
- **`kwargs`** (*`kwargs`*) – These are mainly to make this a "cooperative class" according to super() considered super.

`__module__ = 'woohoo_pdns.load'`

`__weakref__`

list of weak references to the object (if defined)

`_inspect_raw_record`(*raw_record*)

For the first record of every batch this method will be called and the raw record is passed to it. This can be used if 'something' must be determined from source data (e.g. the datetime format).

Note: This is a NOP in `Importer` and meant to be overridden by subclasses if required.

Parameters **`raw_record`** (*`str`*) – the record as it was returned from the source object.

Returns Nothing.

`_inspect_tokenised_record`(*tokenised_rec*)

For every record that was successfully tokenised (i.e. splitted into the required parts), this method will be called. Can be used to decide on further processing for example.

Note: This is a NOP in `Importer` and meant to be overridden by subclasses if required.

Parameters **`tokenised_rec`** (*`record_data`*) – The record as it was tokenised by `_tokenise_record()`.

Returns Nothing.

`_is_valid`(*entry*)

Check if the given entry is considered to be valid.

Entries with an empty rrname or rdata field are considered invalid, for example.

Parameters **`entry`** (*`record_data`*) – the entry to check for validity.

Returns True if the entry passed validation, False otherwise.

`_parse_tokenised_record`(*tokenised_rec*)

After a record was tokenised, it is passed to this method for parsing (e.g. turn a unix timestamp into a datetime, or similar).

Note: This method must be implemened by the concrete subclasses of `Importer`.

Parameters **`tokenised_rec`** (*`record_data`*) – The record as tokenised by `_tokenise_record()`.

Returns A record_data named tuple representing the final record to load. The importer also works if this method or returns None (i.e. nothing is loaded in to the database and the loading process continues) but the record is still considered 'loaded' by the statistics.

`_tokenise_record`(*rec*)

After a raw record is read from the source object, this method is called and passed the raw record to split it into the parts required for a pDNS record.

Note: This method must be implemened by the concrete subclasses of `Importer`.

Parameters

- **`rec`** (*`str`*) – The record as it was returned by the source object. This string must now be splitted into the
- **`parts of a record_data named tuple.`** (*`different`*) –

Returns A record_data named tuple that represents the record to load or None if the record could not be parsed (or should be ignored).

property has_more_data

Indicating if the importer is (potentially) able to produce more data. Mainly means that the source can fetch at least one more record; does not include any validity check(s) of that data though.

Returns True if there is more source data available, false otherwise.

load_batch(*batch_size*, *max_failed_inarow=0*)

The workhorse method of *Importer*.

The source object (self.source) will be initialised with its config (self.src_config) and for subsequent iterations the source's state will be restored (to what was returned by *Source.state* in the last iteration). Then, records will be loaded until either no more data is available or 'batch_size' records are ready for loading into the database.

For the first record in every batch, *_inspect_raw_record()* and *_inspect_tokenised_record()* will be called. For every record *_tokenise_record()* and *_parse_tokenised_record()* are called.

_tokenise_record() is meant to be the place where filtering of source records can occur (return None).

Parameters

- **batch_size** (*int*) – The maximum number of records to process at once.
- **max_failed_inarow** (*int*) – The maximum number of records that fail to import in a row before aborting the processing of this batch.

Returns A load_batch_result named tuple. This contains some statistics and a list of record_data named tuples.

class load_batch_result(*converted*, *loaded*, *ignored*, *records*)

Bases: `tuple`

A named tuple that is used to pass back some statistics as well as a list of `record_data`

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

__module__ = 'woohoo_pdns.load'

static __new__(*_cls*, *converted*, *loaded*, *ignored*, *records*)

Create new instance of load_batch_result(converted, loaded, ignored, records)

__repr__()

Return a nicely formatted representation string

__slots__ = ()

_asdict()

Return a new OrderedDict which maps field names to their values.

_fields = ('converted', 'loaded', 'ignored', 'records')

_fields_defaults = {}

classmethod _make(*iterable*)

Make a new load_batch_result object from a sequence or iterable

_replace(***kwds*)

Return a new load_batch_result object replacing specified fields with new values

property converted

Alias for field number 0

property ignored
Alias for field number 2

property loaded
Alias for field number 1

property records
Alias for field number 3

class woohoo_pdns.load.**SilkFileImporter**(*source_name*, ***kwargs*)
Bases: *woohoo_pdns.load.FileImporter*

Importer to read files produced by the SiLK security suite.

Note: This is a subclass of *FileImporter* as it reads data from files on disk. There are many ways to get the files, for example with the 'rwsender' program included in the SiLK suite.

__init__(*source_name*, ***kwargs*)
To correctly initialise a file source a config dict must be supplied (see 'cfg' argument documentation).

Parameters

- **source_name** (*str*) – Either the name of a file to be read or the name of a directory to scan for files to load.
- **cfg** (*dict*) – A config dictionary that contains the following two keys: * file_pattern (str): The glob pattern to use when reading files from a directory. * rename (bool): Whether or not files should be renamed (by appending '.1') after they are read.

__module__ = 'woohoo_pdns.load'

_inspect_tokenised_record(*tokenised_rec*)
Sometimes, the time in the input as millisecond resolution (for the whole source file). If so, adjust the parsing format to account for this.

Parameters **tokenised_rec** (*record_data*) – The record as tokenised by *_tokenise_record()*.

Returns Nothing.

_parse_tokenised_record(*tokenised_rec*)
Mainly convert the date and time (strings) into aware datetime objects.

Parameters **tokenised_rec** (*record_data*) – The record_data as tokenised by *_tokenise_record()*

Returns A single element list of record_data named tuple.

_tokenise_record(*rec*)
Split a line into tokens:

```
2019-05-13 18:12:44.374|2019-05-13 18:12:44.374|28|gateway.fe.apple-dns.
↪net|1|2a01:b740:0a41:0603::0010
```

becomes:

```
tokens[0] = "2019-05-13 18:12:44.374"    # first_seen
tokens[1] = "2019-05-13 18:12:44.374"    # last_seen
tokens[2] = "28"                         # DNS type
tokens[3] = "gateway.fe.apple-dns.net"   # rrname
```

(continues on next page)

(continued from previous page)

```
tokens[4] = "1"                          # hitcount
tokens[5] = "2a01:b740:0a41:0603::0010"  # rdata
```

respectively:

```
entry.first_seen
entry.last_seen
entry.rrtype
entry.rrname
entry.hitcount
entry.rdata
```

Parameters **`rec`** (*`str`*) – A record returned from the source object.

Returns A single entry list of record_data named tuple.

class `woohoo_pdns.load.`**`SingleLineFileSource`**(*config*, ***kwargs*)

Bases: *`woohoo_pdns.load.FileSource`*

A file source that reads a single line from a file at a time.

`__module__ = 'woohoo_pdns.load'`

`get_next_record`()

Read a single line from a source file (skipping empty lines).

If no line is left, try to open the next file (if available) and read a line from there.

Returns see *`FileSource.get_next_record()`*.

class `woohoo_pdns.load.`**`Source`**(*config*, ***kwargs*)

Bases: `object`

Source object(s) abstract the logic of fetching a ‘single record’ from a source.

For files, this can mean reading one or several lines (e.g. a YAML document), for other sources (e.g. an imaginary Kafka source) this could mean querying a service or calling an API or . . .

`__dict__ = mappingproxy({'__module__': 'woohoo_pdns.load', '__doc__': "\n Source obj`

`__enter__`()

`__exit__`(*exc_type*, *exc_val*, *exc_tb*)

`__init__`(*config*, ***kwargs*)

Parameters

- **`config`** (*`dict`*) – A dictionary that can hold data the source requires to configure itself.
- **`kwargs`** (*`kwargs`*) – These are mainly to make this a “cooperative class” according to super() considered super.

`__module__ = 'woohoo_pdns.load'`

`__weakref__`

list of weak references to the object (if defined)

`get_next_record`()

This method is called by the importer whenever it is ready to load the next record. What is returned will be passed into *`Importer._tokenise_record()`*.

Note: Subclasses must implement this method as it is not implemented here.

Returns A raw record from the source as string.

property state

A source can have 'state' which allows it to resume at the correct next record after a batch of data was processed.

Note: Importers will request state from the source when a batch is about to be finished and will pass whatever the source provided back to the source before starting the next batch.

For a source reading from a file this can for example mean to return the value of `tell()` and then `seek()` to this position when state is passed in again.

exception `woohoo_pdns.load.`**`WoohooImportError`**

Bases: `Exception`

`__module__ = 'woohoo_pdns.load'`

`__weakref__`

list of weak references to the object (if defined)

class `woohoo_pdns.load.`**`YamlFileSource`**(*config*, ***kwargs*)

Bases: *`woohoo_pdns.load.FileSource`*

Read a YAML document from a file on disk.

`__module__ = 'woohoo_pdns.load'`

`get_next_record`()

This method is called by the importer whenever it is ready to load the next record. What is returned will be passed into *`Importer._tokenise_record()`*.

Note: Subclasses must implement this method as it is not implemented here.

Returns A raw record from the source as string.

6.1.4 woohoo_pdns.meta module

class `woohoo_pdns.meta.`**`LookupDict`**(*name=None*)

Bases: `dict`

Dictionary lookup object.

TODO: understand this… https://github.com/kennethreitz/requests/blob/master/requests/structures.py

`__dict__ = mappingproxy({'__module__': 'woohoo_pdns.meta', '__doc__': '\n Dictionary`

`__getitem__`(*key*)

x.__getitem__(y) <==> x[y]

`__init__`(*name=None*)

Initialize self. See help(type(self)) for accurate signature.

`__module__ = 'woohoo_pdns.meta'`

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

get(*key*, *default=None*)
Return the value for key if key is in the dictionary, else default.

woohoo_pdns.meta.**_init**()

6.1.5 woohoo_pdns.pdns module

class woohoo_pdns.pdns.**Database**(*db_url*)
Bases: `object`

The Database object is the interface to the database holding pDNS records.

This object is designed as a context manager, it can be used with `with`.

__dict__ = mappingproxy({'__module__': 'woohoo_pdns.pdns', '__doc__': '\n The Databa

__enter__()

__exit__(*exc_type*, *exc_value*, *traceback*)

__init__(*db_url*)
Initialise the connection to the database.

Parameters **db_url** (*string*) – The URL to the database, e.g. `postgresql+psycopg2://user:password@hostname/database_name`

__module__ = 'woohoo_pdns.pdns'

__weakref__
list of weak references to the object (if defined)

_query_for_ip(*q*)
Query the "rdata" for an IP address.

Parameters **q** (*str*) – the IP address (as a string) to search for.

Returns A list of *Record* objects for records found (can be empty)

_query_for_name(*q*, *rdata*)
Query the "rrname" or the "rdata" in the DB.

Note: This is for string queries only (no IP address queries).

Parameters

- **q** (*str*) – The search term, can contain "*" as a wildcard.
- **rdata** (*bool*) – If True and the query is a text query, search the right hand side instead of the left hand
- **side.** –

Returns A list of *Record* objects for records found (can be empty)

add_record(*rrtype*, *rrname*, *rdata*, *first_seen=None*, *last_seen=None*, *num_hits=1*)
Add a (new) record to the database.

If a record with that rrtype, rrname, rdata already exists in the database, the hitcount is increased by num_hits, first_seen or last_seen are updated if necessary and the existing object is returned. Otherwise a new object will be created and returned (with hitcount 1, fist_seen = last_seen = sighted_at (or "now" if sighted_at is not provided)).

Parameters

- **rrtype** (*int*) – the id for the DNS record type (e.g. 1 for A, 28 for AAAA, etc. See https://en.wikipedia.org/wiki/List_of_DNS_record_types)
- **rrname** (*string*) – the "left hand side" of the record; a trailing dot will be removed
- **rdata** (*string*) – the "right hand side" of the record; a trailing dot will be removed
- **first_seen** (*datetime*) – the date and time of the first (oldest) sighting; if omitted and also no last_seen is provided "now" will be used
- **last_seen** (*datetime*) – the date and time of the most recent sighting; if omitted and also no first_seen is provided "now" will be used
- **num_hits** (*int*) – the number of times this record was seen (will be added to an existing records hitcount)

Returns A *Record* object representing this record.

close()
Close the connection to the database. It is important to call this method after you are done. Will be called automagically when used with the context manager.

property count
The total number of pDNS records in the database.

find_record(*rrtype*, *rrname*, *rdata=None*)
Search for a record (by type and left hand side, optionally also right hand side).

Parameters

- **rrtype** (*int*) – The id for the DNS record type (e.g. 1 for A, 28 for AAAA, etc. See https://en.wikipedia.org/wiki/List_of_DNS_record_types).
- **rrname** (*string*) – The "left hand side" of the record.
- **rdata** (*string*) – The "right hand side" of the record.

Returns The *Record* object representing the record.

Raises **NoResultFound** –

load(*source_name*, *batch_size=10000*, *cfg=None*, *data_timezone='UTC'*, *strict=False*, *loader='woohoo_pdns.load.SilkFileImporter'*)
Load data into the database.

The actual work is done by the class referenced in the "loader" argument.

Parameters

- **source_name** (*str*) – The directory or filename or other reference to the source (e.g. a Kafka topic name) where data should be loaded from.
- **batch_size** (*int*) – For more efficient loading into the database, records are inserted/updated in batches; this defines the maximum number of records to process at once.

- **`cfg`** (*`dict`*) – A dictionary with config items that will be passed to the constructor of the `Importer`.
- **`strict`** (*`bool`*) – If true, abort loading if "errors" are detected in the input. If false, try to "fix" the error(s) and/or to continue loading remaining data. Default is *False*.
- **`data_timezone`** (*`timezone string`*) – If source data without a timezone specification is found, assume the timezone is this.
- **`loader`** (`Importer`) – Defines what class is used for the actual loading of data.

`property most_recent`
The most recent record in the database, i.e. the one with the most recent "last_seen" datetime.

`query`(*q*, *rdata=False*)
Issue a query against the database.

When

Parameters

- **`q`** (*`str`*) – the query, can be an IP address (v4 or v6) or text.
- **`rdata`** (*`bool`*) – text queries look for matches on the "left hand side" (rrname) unless this option is set which makes the query search for matches on the "right hand side". Use it for example to search for domains that share a common name server (NS record). For IP address queries, this is ignored; defaults to False.

Returns A list of records that matches the query term.

Throws: *`MissingEntry`* if no match is found for the query.

`property records`
A list of all pDNS records in the database.

`exception` `woohoo_pdns.pdns.`**`InvalidEntry`**
Bases: `ValueError`

When SQLAlchemy fails to commit a record to the database, this exception is raised.

The details produced by SQLAlchemy will be included in the exceptions description.

`__module__ = 'woohoo_pdns.pdns'`

`__weakref__`
list of weak references to the object (if defined)

`exception` `woohoo_pdns.pdns.`**`MissingEntry`**
Bases: `ValueError`

When a query does not yield any result, this exception is raised.

`__module__ = 'woohoo_pdns.pdns'`

`__weakref__`
list of weak references to the object (if defined)

`class` `woohoo_pdns.pdns.`**`Record`**(***kwargs*)
Bases: `sqlalchemy.ext.declarative.api.Base`

Database representation of a record in the pDNS system.

A record can be of any DNS type (A, AAAA, TXT, PTR, . . .) and has a "left side" (`rrname`) and a "right side" (`rdata`). More information about "left hand side" and "right hand side" is available on the Farsight website for example.

first_seen
The date and time (incl. timezone) when a record was first seen by this pDNS system.

Type DateTime

last_seen
The date and time (incl. timezone) when a record was last seen by this pDNS system (i.e. the most recent "sighting").

Type DateTime

rrtype
The type of the record (A, AAAA, TXT, . . .) according to the official list of DNS types.

Type int

hitcount
The number of times this record was "sighted" by this pDNS system.

Type int

__init__(***kwargs*)
The init method is just setting up a logger for the class.

The kwargs just make it a "cooperative class" according to super() considered super.

__mapper__ = <Mapper at 0x7f36a37dbe10; Record>

__module__ = 'woohoo_pdns.pdns'

__repr__()
Return repr(self).

__table__ = Table('record', MetaData(bind=None), Column('first_seen', DateTime(timezon

__tablename__ = 'record'

_rdata

_rrname

_sa_class_manager = {'_rdata': <sqlalchemy.orm.attributes.InstrumentedAttribute objec

ensure_aware_dt()
When reconstructing a `Record` from the database, ensure that the datetimes (first_seen and last_seen) are "aware" objects (i.e. have a timezone).

This is mainly an issue when using sqlite (e.g. for testing) as sqlite does not store timezone information. In case the timezone information is missing, UTC is assumed and added.

first_seen

hitcount

last_seen

property rdata
The "rdata", i.e. the "right hand side" of the record (cf. class attribute documentation).

property rrname
The "rrname", i.e. the "left hand side" of the record (cf. class attribute documentation).

Note: When setting this property, the value will be sanitized by `woohoo_pdns.util.sanitise_input()`; this means that a trailing dot will be removed unless the value is just a dot.

rrtype

to_dict()
Convert the record object to a dictionary representation that is suitable for SQLAlchemy bulk operations.

to_jsonable()
Convert the record object to a JSON-friendly dictionary representation.

Note: This dict is compatible with the Passive DNS - Common Output Format.

update(*rec*)
Update a record with (potentially) new information from a different record.

This means updating (adding) the hitcount as well as updating first_seen and/or last_seen if required.

Parameters **rec** (*Record*) – The record to take the new information from.

6.1.6 woohoo_pdns.util module

class woohoo_pdns.util.**LoaderCache**
Bases: object

This class implements the cache used when loading entries into the database.

Because pDNS databases have to ingest high volumes of data with high redundancy (never seen before entries are comparatively rare) it can be expected that caching substantially improves performance.

The cache internally holds values in dictionaries with a key derived from the actual data. To add records to the cache the named tuple 'record_data' should be used.

Note: When adding a record to the cache, four modes are available: cache_only, new, updated and auto. For a description of the modes, see the documentation of the "modes" named tuple.

MODES = cache_modes(cache_only=1, new=2, updated=4, auto=8)

__contains__(*item*)
Checks if the cache contains the entry represented by the named tuple (or dict) passed in.

Parameters **item** (*record_data*) – The record which should be checked for presence in the cache.

Returns True if the item is in the cache, false otherwise.

__dict__ = mappingproxy({'__module__': 'woohoo_pdns.util', '__doc__': '\n This class

__init__()
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'woohoo_pdns.util'

__weakref__
list of weak references to the object (if defined)

_add_to_cache_only(*item*)

_add_to_new(*item*)

_add_to_update(*item*)

static _dictionise(*item*)

Convert 'item' (named tuple class:*record_data*) into a dictionary *{ key: item }*

Parameters **(class** (*item*) – *record_data*): the item to 'convert' into a dictionary

Returns dict with one key (item.key, if it was set) and item as its value

static _tupelise(*item_key*, *item_value*)

Convert 'item_key, item_value' (value of type dictionary) into a named tuple class:*record_data*

Parameters

- **item_key** (*str*) – the key that is used for the 'value dict'
- **item_value** (*dict*) – a dictionary that holds the relevant data to create a class:*record_data*

Returns *record_data*): the item that results from 'converting' the dictionary

Return type item (class

add(*item*, *mode=8*)

Add a new item to the cache.

Parameters

- **item** (*record_data*) – The representation of the item to add to the cache.
- **mode** (*mode*) – What mode to use (see documentation of mode for details) if the record is not yet in the cache.

Returns Nothing.

clear()

Clear the cache (i.e. remove all entries).

Returns Nothing.

get_new_entries(*for_bulk=False*)

Return the list of items that are considered not to be present in the pDNS database yet.

Note: The main reason for differentiating between new and updated entries (with respect to the pDNS database, not the cache) is to allow bulk operations in SQLAlchemy; it must be known if 'INSERT' or 'UPDATE' statements should be used.

Parameters **for_bulk** (*bool*) – If true, a list of dictionaries will be returned (suitable for SQLAlchemy bulk operations), if false, a list of record_data named tuples will be returned. For more information about SQLAlchemy bulk operations, see the SQLAlchemy documentation on bulk operations.

Returns A list of either named tuples or dictionaries (see 'for_bulk' argument).

get_to_update(*for_bulk=False*)

The same as *get_new_entries()* but for entries considered to already be present in the pDNS database (not necessarily the cache).

Parameters **for_bulk** (*bool*) – see argument with the same name documented for *get_new_entries()*

Returns A list of either named tuples or dictionaries (see 'for_bulk' argument).

static merge(*existing*, *new*)

Merge two cached items by updating the *new* item's hitcount (add the existing item's count to it) and set the *new* item's first_seen and last_seen to the minimum (maximum) of the new and the existing item's values.

Parameters

- **existing** (*dict*) – An item present in the cache
- **new** (*dict*) – An item that should be updated with the info already present in the cache.

Returns Nothing, the *new* item will be updated in place.

modes

The mode is relevant when adding records to the cache that are not already present in the cache. 'cache_only' should only be used to (pre) populate the cache. This is mainly useful if the 'auto' mode should be used later on. 'auto' assumes that the cache already holds *all* relevant entries; therefore, when adding an entry it will be cached as 'new' if it was not present in the cache before and as 'updated' if it already was in the cache. If the mode is set to 'new', the entry will be considered to be new (i.e. returned by `get_new_entries()`) whereas with the mode set to 'updated' it will be considered to already be known by the pDNS database (but not necessarily the cache, i.e. it will be returned by `get_to_update()`).

alias of `cache_modes`

rollover()

Should be called after the pDNS database is updated with the currently cached entries (i.e. after the bulk operations are done for the lists returned by `get_new_entries()` and `get_to_update()`).

This will 'move' all cached entries into the 'cache_only' status, indicating that they are 'known' but not 'dirty' (in a cache's way of using that word).

Returns Nothing.

`woohoo_pdns.util.`**record_data**

A named tuple holding the values of a single entry in the pDNS database.

Note: first_seen and last_seen can be both, datetimes or timestamps (integers) but it must be consistent. The 'key' field can be left empty; it will then be auto-populated by the cache in a consistent way. If it is non-empty the passed in key is kept and it is the caller's responsibility to guarantee uniqueness of the key(s).

alias of `woohoo_pdns.util.pdns_entry_tokens`

`woohoo_pdns.util.`**record_to_nt**(*rec*)

Takes a dictionary style cache entry and returns a corresponding named tuple.

Note: The "key" is not set because the other functions in this module do not require it (DRY).

`woohoo_pdns.util.`**sanitise_input**(*str_in*)

DNS entries technically end in a dot but for pDNS purposes the dot is mainly cruft, so we remove it.

Note: If the input string to this function is just a dot, it is kept. While a single dot might be 'surprising' it is still better than an empty string.

6.1.7 Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

EIGHT

WHAT WOOHOO PDNS IS

woohoo pDNS is a database to store and query passive DNS data. It is written in Python 3 and aims to support data collected by the SiLK NetSA security suite although it can work with other source data as well.

In addition, it has a Web GUI component that can be installed separately.

If you want to know in more detail what (a) passive DNS (data(base)) is, the FAQ on the Farsight Security website is a valid resource to read up on the topic.

CHAPTER

NINE

WHAT WOOHOO PDNS IS *NOT*

woohoo pDNS is *not* optimised for speed. It is geared towards small-ish installations (read: up to a couple of thoundand inserts per minute?).

9.1 Benefits

- Uses SQLAlchemy for the database stuff: let the pros handle SQL
- Adheres to the Passive DNS - Common Output Format.
- Provides a RESTful interface

PYTHON MODULE INDEX

INDEX