NIPAP Documentation

Release 1.0

Kristian Larsson, Lukas Garberg

November 25, 2016

Contents

The majority of this documentation is generated from the Nipap Python module where most of the server side logic is placed. A thin XML-RPC layer is wrapped around the Nipap class to expose its functions over an XML-RPC interface as well as translating internal Exceptions into XML-RPC errors codes. It is feasible to implement other wrapper layers should one need a different interface, though the XML-RPC interface should serve most well.

Given that the documentation is automatically generated from this internal Nipap class, there is some irrelevant information regarding class structures - just ignore that! :)

Happy hacking!

Contents:

CHAPTER 1

Design choices

This document tries to describe the overall design goals and decisions taken in the development process of NIPAP.

Overall goals:

- Simple to interact with for users and other systems alike, you should _want_ to use it.
- Powerful, allowing the system to do things that are best performed by computers leaving human users happy.
- Easy to maintain. Tele2 does not have many developers so maintenance needs to be simple.

Out of these goals, the following set of tools and resources have been chosen for the overall design.

- Backend storage implemented using PostgreSQL
- Backend / XML-RPC API in Python with the Flask-XML-RPC framework
- CLI client in Python
- Web GUI in Python using the Pyramid framework

1.1 Why PostgreSQL?

Postgres has a native datatype called 'inet' which is able to store both IPv4 and IPv6 addresses and their prefix-length. The latter (IPv6) usually poses a problem to database storage as even long integers can only accomodate 64 bits. Hacks using two columns or some numeric type exist, but often result in cumbersome or slow solutions. Postgres inet type is indexable and using ip4r even ternary accesses (such as a longest prefix lookup) is indexable. This makes it a superior solution compared to most other databases.

PostgreSQL is an open source database under a BSD license, meaning anyone can use it and modify it. Ingres development was started in the early 1970s and Postgres (Post Ingres) later evolved into PostgreSQL when the SQL language was added in 1995 as query language. It is the oldest and the most advanced open source relational database available today.

1.2 Why Python?

Python is a modern interpreted language with an easy to use syntax and plenty of powerful features. Experienced programmers usually pick up the language within a few days, less experienced within a slightly larger time. Its clean syntax makes it easy for people to familiarize themselves with the NIPAP codebase.

1.3 Why Flask (and not Twisted)?

NIPAP was originally implemented with a Twisted powered backend but has since been rewritten to use Flask.

Twisted is one of the leading concurrency frameworks allowing developers to focus on their own application instead of labour-intensive work surrounding it. It is used by companies such as Apple (iCal server) and Spotify (playlist service) to serve hundreds of thousands of users. Twisted includes modules for serving data over XML-RPC and/or SOAP as well as a complete toolset for asynchronous calls.

Unfortunately, using Twisted asynchronous model is rocket science. Code needs to be built specifically for Twisted. The original implementation never took advantage of asynchronous calls and deferred objects and during later attempts of adding it we realised how difficult and cumbersome it is. One really needs to write code from the beginning up to suit Twisted.

Instead, we turned our eye to Flask, which together with Tornado offers a pre-forked model. We didn't need to change a line of code in our backend module yet we have now achieved a simple form of parallelism. Flask is easy! For NIPAP, this means we focus on NIPAP code and not XML-RPC and concurrency code.

1.4 Why XML-RPC?

From the very start, it was a important design goal that NIPAP remain open for interoperation with any and all other systems and so it would be centered around a small and simple API from which everything can be performed. Not intending to reinvent the wheel, especially given the plethora of already available APIs, it was up to chosing the "right one". Twisted, which was originally used for Twisteds backend, offers built-in support for SOAP (WebServices) as well as XML-RPC but given design goals such as simple, SOAP didn't quite feel right and thus XML-RPC was chosen. It should however be noted that NIPAPs XML-RPC protocol is a thin wrapper around an inner core and so exposing a SOAP interface in addition to XML-RPC can be easily achieved. XML-RPC shares a lot of similarities with SOAP but is very much less complex and it is possible for a human to read it in a tcpdump or similar while with SOAP one likely needs some interpreter or the brain of Albert Einstein. Since the original implementation with Twisted, the backend has been reimplemented using Flask-XML-RPC which is an extension to Flask. In addition to XML-RPC, it is also possible to load a JSON-RPC module with Flask to add another interface.

CHAPTER 2

NIPAP API

This module contains the Nipap class which provides most of the backend logic in NIPAP apart from that contained within the PostgreSQL database.

NIPAP contains four types of objects: ASNs, VRFs, prefixes and pools.

2.1 VRF

A VRF represents a Virtual Routing and Forwarding instance. By default, one VRF which represents the global routing table (VRF "default") is defined. This VRF always has the ID 0.

2.1.1 VRF attributes

- `id` - ID number of the VRF.
- **`vrf` - The VRF RDs administrator and assigned number subfields** (eg. 65000:123).
- `name` - A short name, such as 'VPN Customer A'.
- `description` - A longer description of what the VRF is used for.
- `tags` - Tag keywords for simple searching and filtering of VRFs.
- **`avps` - Attribute-Value Pairs. This field can be used to add** various extra attributes that a user wishes to store together with a VRF.

2.1.2 VRF functions

- *`list_vrf()`* - Return a list of VRFs.
- *`add_vrf()`* - Create a new VRF.
- *`edit_vrf()`* - Edit a VRF.
- *`remove_vrf()`* - Remove a VRF.
- *`search_vrf()`* - Search VRFs based on a formatted dict.
- *`smart_search_vrf()`* - Search VRFs based on a query string.

2.2 Prefix

A prefix object defines an IP address prefix. Prefixes can be one of three different types; reservation, assignment or host. Reservation; a prefix which is reserved for future use. Assignment; addresses assigned to a specific purpose. Host; prefix of max length within an assigment, assigned to an end host.

2.2.1 Prefix attributes

- `id` - ID number of the prefix.
- `prefix` - The IP prefix itself.
- `prefix_length` - Prefix length of the prefix.
- `display_prefix` - A more user-friendly version of the prefix.
- `family` - Address family (integer 4 or 6). Set by NIPAP.
- `vrf_id` - ID of the VRF which the prefix belongs to.
- `vrf_rt` - RT of the VRF which the prefix belongs to.
- `vrf_name` - Name of VRF which the prefix belongs to.
- `description` - A short description of the prefix.
- `comment` - A longer text describing the prefix and its use.
- `node` - Name of the node on which the address is configured.
- `pool_id` - ID of pool, if the prefix belongs to a pool.
- `pool_name` - Name of pool, if the prefix belongs to a pool.
- `type` - Prefix type, string 'reservation', 'assignment' or 'host'.
- `status` - Status, string 'assigned', 'reserved' or 'quarantine'.
- `indent` - Depth in prefix tree. Set by NIPAP.
- `country` - Two letter country code where the prefix resides.
- `order_id` - Order identifier.
- `customer_id` - Customer identifier.
- `vlan` - VLAN identifier, 0-4096.
- `tags` - Tag keywords for simple searching and filtering of prefixes.
- **`avps` - Attribute-Value Pairs. This field can be used to add** various extra attributes that a user wishes to store together with a prefix.
- **`expires` - Set a date and time for when the prefix assignment** expires. Multiple formats are supported for specifying time, for absolute time ISO8601 style dates can be used and None or the text strings 'never' or 'infinity' is treated as positive infinity and means the assignment never expires. It is also possible to specify relative time and a fuzzy parser is used to interpret strings such as "tomorrow" or "2 years" into an absolute time.
- **`external_key` - A field for use by external systems which needs to** store references to its own dataset.
- **`authoritative_source` - String identifying which system last** modified the prefix.
- **`alarm_priority` - String 'warning', 'low', 'medium', 'high' or** 'critical'.

- **monitor - A boolean specifying whether the prefix should be** monitored or not.
- **display - Only set by the *search_prefix()* and** *smart_search_prefix()* functions, see their documentation for explanation.

2.2.2 Prefix functions

- *list_prefix()* - Return a list of prefixes.
- *add_prefix()* - Add a prefix, more or less automatically.
- *edit_prefix()* - Edit a prefix.
- *remove_prefix()* - Remove a prefix.
- *search_prefix()* - Search prefixes based on a formatted dict.
- *smart_search_prefix()* - Search prefixes based on a string.

2.3 Pool

A pool is used to group together a number of prefixes for the purpose of assigning new prefixes from that pool. *add_prefix()* can for example be asked to return a new prefix from a pool. All prefixes that are members of the pool will be examined for free space and a new prefix, of the specified prefix-length, will be returned to the user.

2.3.1 Pool attributes

- `id` - ID number of the pool.
- `name` - A short name.
- `description` - A longer description of the pool.
- `default_type` - Default prefix type (see prefix types above.
- `ipv4_default_prefix_length` - Default prefix length of IPv4 prefixes.
- `ipv6_default_prefix_length` - Default prefix length of IPv6 prefixes.
- `tags` - Tag keywords for simple searching and filtering of pools.
- **avps - Attribute-Value Pairs. This field can be used to add** various extra attributes that a user wishes to store together with a pool.

2.3.2 Pool functions

- *list_pool()* - Return a list of pools.
- *add_pool()* - Add a pool.
- *edit_pool()* - Edit a pool.
- *remove_pool()* - Remove a pool.
- *search_pool()* - Search pools based on a formatted dict.
- *smart_search_pool()* - Search pools based on a string.

2.4 ASN

An ASN object represents an Autonomous System Number (ASN).

2.4.1 ASN attributes

- `asn` - AS number.
- `name` - A name of the AS number.

2.4.2 ASN functions

- *`list_asn()`* - Return a list of ASNs.
- *`add_asn()`* - Add an ASN.
- *`edit_asn()`* - Edit an ASN.
- *`remove_asn()`* - Remove an ASN.
- *`search_asn()`* - Search ASNs based on a formatted dict.
- *`smart_search_asn()`* - Search ASNs based on a string.

2.5 The 'spec'

Central to the use of the NIPAP API is the spec – the specifier. It is used by many functions to in a more dynamic way specify what element(s) you want to select. Mainly it came to be due to the use of two attributes which can be thought of as primary keys for an object, such as a pool's `id` and `name` attribute. They are however implemented so that you can use more or less any attribute in the spec, to be able to for example get all prefixes of family 6 with type reservation.

The spec is a dict formatted as:

```
vrf_spec = {
    'id': 512
}
```

But can also be elaborated somehwat for certain objects, as:

```
prefix_spec = {
    'family': 6,
    'type': 'reservation'
}
```

If multiple keys are given, they will be ANDed together.

2.6 Authorization & accounting

With each query an object extending the BaseAuth class should be passed. This object is used in the Nipap class to perform authorization (not yet implemented) and accounting. Authentication should be performed at an earlier stage and is NOT done in the Nipap class.

Each command which alters data stored in NIPAP is logged. There are currently no API functions for extracting this data, but this will change in the future.

2.7 Classes

class nipap.backend.**Inet**(*addr*)

This works around a bug in psycopg2 version somewhere before 2.4. The __init__ function in the original class is broken and so this is merely a copy with the bug fixed.

Wrap a string to allow for correct SQL-quoting of inet values.

Note that this adapter does NOT check the passed value to make sure it really is an inet-compatible address but DOES call adapt() on it to make sure it is impossible to execute an SQL-injection by passing an evil value to the initializer.

class nipap.backend.**Nipap**(*auto_install_db=False*, *auto_upgrade_db=False*)

Main NIPAP class.

The main NIPAP class containing all API methods. When creating an instance, a database connection object is created which is used during the instance's lifetime.

add_asn(**args*, ***kwargs*)

Add AS number to NIPAP.

- ***auth*** **[BaseAuth]** AAA options.
- ***attr*** **[asn_attr]** ASN attributes.

Returns a dict describing the ASN which was added.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.add_asn()* for full understanding.

add_pool(**args*, ***kwargs*)

Create a pool according to *attr*.

- ***auth*** **[BaseAuth]** AAA options.
- ***attr*** **[pool_attr]** A dict containing the attributes the new pool should have.

Returns a dict describing the pool which was added.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.add_pool()* for full understanding.

add_prefix(**args*, ***kwargs*)

Add a prefix and return its ID.

- ***auth*** **[BaseAuth]** AAA options.
- ***attr*** **[prefix_attr]** Prefix attributes.
- ***args*** **[add_prefix_args]** Arguments explaining how the prefix should be allocated.

Returns a dict describing the prefix which was added.

Prefixes can be added in three ways; manually, from a pool or from a prefix.

Manually All prefix data, including the prefix itself is specified in the *attr* argument. The *args* argument shall be omitted.

From a pool Most prefixes are expected to be automatically assigned from a pool. In this case, the `prefix` key is omitted from the *attr* argument. Also the `type` key can be omitted and the prefix type will then be set to the pools default prefix type. The *find_free_prefix()* function is used to find available prefixes for this allocation method, see its documentation for a description of how the *args* argument should be formatted.

From a prefix A prefix can also be selected from another prefix. Also in this case the `prefix` key is omitted from the *attr* argument. See the documentation for the *find_free_prefix()* for a description of how the *args* argument is to be formatted.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.add_prefix()* for full understanding.

add_vrf(**args*, ***kwargs*)
Add a new VRF.

- ***auth* [BaseAuth]** AAA options.
- ***attr* [vrf_attr]** The news VRF's attributes.

Add a VRF based on the values stored in the *attr* dict.

Returns a dict describing the VRF which was added.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.add_vrf()* for full understanding.

edit_asn(**args*, ***kwargs*)
Edit AS number

- ***auth* [BaseAuth]** AAA options.
- ***asn* [integer]** AS number to edit.
- ***attr* [asn_attr]** New AS attributes.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.edit_asn()* for full understanding.

edit_pool(**args*, ***kwargs*)
Update pool given by *spec* with attributes *attr*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [pool_spec]** Specifies what pool to edit.
- ***attr* [pool_attr]** Attributes to update and their new values.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.edit_pool()* for full understanding.

edit_prefix(**args*, ***kwargs*)
Update prefix matching *spec* with attributes *attr*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [prefix_spec]** Specifies the prefix to edit.
- ***attr* [prefix_attr]** Prefix attributes.

Note that there are restrictions on when and how a prefix's type can be changed; reservations can be changed to assignments and vice versa, but only if they contain no child prefixes.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.edit_prefix()* for full understanding.

edit_vrf (**args*, ***kwargs*)
Update VRFs matching *spec* with attributes *attr*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [vrf_spec]** Attibutes specifying what VRF to edit.
- ***attr* [vrf_attr]** Dict specifying fields to be updated and their new values.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.edit_vrf()* for full understanding.

find_free_prefix (*auth*, *vrf*, *args*)
Finds free prefixes in the sources given in *args*.

- ***auth* [BaseAuth]** AAA options.
- ***vrf* [vrf]** Full VRF-dict specifying in which VRF the prefix should be unique.
- ***args* [find_free_prefix_args]** Arguments to the find free prefix function.

Returns a list of dicts.

Prefixes can be found in two ways: from a pool of from a prefix.

From a pool The *args* argument is set to a dict with key `from-pool` set to a pool spec. This is the pool from which the prefix will be assigned. Also the key `family` needs to be set to the adress family (integer 4 or 6) of the requested prefix. Optionally, also the key `prefix_length` can be added to the *attr* argument, and will then override the default prefix length.

Example:

```
args = {
    'from-pool': { 'name': 'CUSTOMER-' },
    'family': 6,
    'prefix_length': 64
}
```

From a prefix Instead of specifying a pool, a prefix which will be searched for new prefixes can be specified. In *args*, the key `from-prefix` is set to the prefix you want to allocate from and the key `prefix_length` is set to the wanted prefix length.

Example:

```
args = {
    'from-prefix': '192.0.2.0/24'
    'prefix_length': 27
}
```

The key `count` can also be set in the *args* argument to specify how many prefixes that should be returned. If omitted, the default value is 1000.

The internal backend function *find_free_prefix()* is used internally by the *add_prefix()* function to find available prefixes from the given sources. It's also exposed over XML-RPC, please see

the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.find_free_prefix()* for full understanding.

list_asn (*auth*, *asn=None*)
List AS numbers matching *spec*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [asn_spec]** An automous system number specification. If omitted, all ASNs are returned.

Returns a list of dicts.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.list_asn()* for full understanding.

list_pool (*auth*, *spec=None*)
Return a list of pools.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [pool_spec]** Specifies what pool(s) to list. Of omitted, all will be listed.

Returns a list of dicts.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.list_pool()* for full understanding.

list_prefix (*auth*, *spec=None*)
List prefixes matching the *spec*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [prefix_spec]** Specifies prefixes to list. If omitted, all will be listed.

Returns a list of dicts.

This is a quite blunt tool for finding prefixes, mostly useful for fetching data about a single prefix. For more capable alternatives, see the *search_prefix()* or *smart_search_prefix()* functions.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.list_prefix()* for full understanding.

list_vrf (*auth*, *spec=None*)
Return a list of VRFs matching *spec*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [vrf_spec]** A VRF specification. If omitted, all VRFs are returned.

Returns a list of dicts.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.list_vrf()* for full understanding.

remove_asn (**args*, ***kwargs*)
Remove an AS number.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [asn]** An ASN specification.

Remove ASNs matching the *asn* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.remove_asn()` for full understanding.

remove_pool(**args*, ***kwargs*)
Remove a pool.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [pool_spec]** Specifies what pool(s) to remove.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.remove_pool()` for full understanding.

remove_prefix(**args*, ***kwargs*)
Remove prefix matching *spec*.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [prefix_spec]** Specifies prefixe to remove.
- ***recursive* [bool]** When set to True, also remove child prefixes.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.remove_prefix()` for full understanding.

remove_vrf(**args*, ***kwargs*)
Remove a VRF.

- ***auth* [BaseAuth]** AAA options.
- ***spec* [vrf_spec]** A VRF specification.

Remove VRF matching the *spec* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.remove_vrf()` for full understanding.

search_asn(*auth*, *query*, *search_options=None*)
Search ASNs for entries matching 'query'

- ***auth* [BaseAuth]** AAA options.
- ***query* [dict_to_sql]** How the search should be performed.
- ***search_options* [options_dict]** Search options, see below.

Returns a list of dicts.

The *query* argument passed to this function is designed to be able to specify how quite advanced search operations should be performed in a generic format. It is internally expanded to a SQL WHERE-clause.

The *query* is a dict with three elements, where one specifies the operation to perform and the two other specifies its arguments. The arguments can themselves be *query* dicts, to build more complex queries.

The `operator` key specifies what operator should be used for the comparison. Currently the following operators are supported:

- `and` - Logical AND
- `or` - Logical OR

- `equals` - Equality; =
- `not_equals` - Inequality; !=
- `like` - SQL LIKE
- `regex_match` - Regular expression match
- `regex_not_match` - Regular expression not match

The `val1` and `val2` keys specifies the values which are subjected to the comparison. `val1` can be either any prefix attribute or an entire query dict. `val2` can be either the value you want to compare the prefix attribute to, or an entire *query* dict.

The search options can also be used to limit the number of rows returned or set an offset for the result.

The following options are available:

- `max_result` - The maximum number of prefixes to return (default `50`).
- `offset` - Offset the result list this many prefixes (default `0`).

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.search_tag()` for full understanding.

search_pool(*auth*, *query*, *search_options=None*)

Search pool list for pools matching *query*.

- ***auth*** **[BaseAuth]** AAA options.
- ***query*** **[dict_to_sql]** How the search should be performed.
- ***search_options*** **[options_dict]** Search options, see below.

Returns a list of dicts.

The *query* argument passed to this function is designed to be able to specify how quite advanced search operations should be performed in a generic format. It is internally expanded to a SQL WHERE-clause.

The *query* is a dict with three elements, where one specifies the operation to perform and the two other specifies its arguments. The arguments can themselves be *query* dicts, to build more complex queries.

The `operator` key specifies what operator should be used for the comparison. Currently the following operators are supported:

- `and` - Logical AND
- `or` - Logical OR
- `equals` - Equality; =
- `not_equals` - Inequality; !=
- `like` - SQL LIKE
- `regex_match` - Regular expression match
- `regex_not_match` - Regular expression not match

The `val1` and `val2` keys specifies the values which are subjected to the comparison. `val1` can be either any pool attribute or an entire query dict. `val2` can be either the value you want to compare the pool attribute to, or an entire *query* dict.

Example 1 - Find the pool whose name match 'test':

```
query = {
    'operator': 'equals',
    'val1': 'name',
    'val2': 'test'
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM pool WHERE name = 'test'
```

Example 2 - Find pools whose name or description regex matches 'test':

```
query = {
    'operator': 'or',
    'val1': {
        'operator': 'regex_match',
        'val1': 'name',
        'val2': 'test'
    },
    'val2': {
        'operator': 'regex_match',
        'val1': 'description',
        'val2': 'test'
    }
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM pool WHERE name ~* 'test' OR description ~* 'test'
```

The search options can also be used to limit the number of rows returned or set an offset for the result.

The following options are available:

- `max_result` - The maximum number of pools to return (default `50`).
- `offset` - Offset the result list this many pools (default `0`).

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *`nipap.xmlrpc.NipapXMLRPC.search_pool()`* for full understanding.

`search_prefix`(*auth*, *query*, *search_options=None*)

Search prefix list for prefixes matching *query*.

- ***auth*** **[BaseAuth]** AAA options.
- ***query*** **[dict_to_sql]** How the search should be performed.
- ***search_options*** **[options_dict]** Search options, see below.

Returns a list of dicts.

The *query* argument passed to this function is designed to be able to express quite advanced search filters. It is internally expanded to an SQL WHERE-clause.

The *query* is a dict with three elements, where one specifies the operation to perform and the two other specifies its arguments. The arguments can themselves be *query* dicts, i.e. nested, to build more complex queries.

The `operator` key specifies what operator should be used for the comparison. Currently the following operators are supported:

- `and` - Logical AND
- `or` - Logical OR
- `equals_any` - Equality of any element in array
- `equals` - Equality; =
- `not_equals` - Inequality; !=
- `less` - Less than; <
- `less_or_equal` - Less than or equal to; <=
- `greater` - Greater than; >
- `greater_or_equal` - Greater than or equal to; >=
- `like` - SQL LIKE
- `regex_match` - Regular expression match
- `regex_not_match` - Regular expression not match
- `contains` - IP prefix contains
- `contains_equals` - IP prefix contains or is equal to
- `contained_within` - IP prefix is contained within
- `contained_within_equals` - IP prefix is contained within or equals

The `val1` and `val2` keys specifies the values which are subjected to the comparison. `val1` can be either any prefix attribute or a query dict. `val2` can be either the value you want to compare the prefix attribute to, or a *query* dict.

Example 1 - Find the prefixes which contains 192.0.2.0/24:

```
query = {
    'operator': 'contains',
    'val1': 'prefix',
    'val2': '192.0.2.0/24'
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM prefix WHERE prefix contains '192.0.2.0/24'
```

Example 2 - Find for all assignments in prefix 192.0.2.0/24:

```
query = {
    'operator': 'and',
    'val1': {
        'operator': 'equals',
        'val1': 'type',
        'val2': 'assignment'
    },
    'val2': {
        'operator': 'contained_within',
        'val1': 'prefix',
        'val2': '192.0.2.0/24'
    }
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM prefix WHERE (type == 'assignment') AND (prefix contained within '192.0.2.0/24
```

If you want to combine more than two expressions together with a boolean expression you need to nest them. For example, to match on three values, in this case the tag 'foobar' and a prefix-length between /10 and /24, the following could be used:

```
query = {
    'operator': 'and',
    'val1': {
        'operator': 'and',
        'val1': {
            'operator': 'greater',
            'val1': 'prefix_length',
            'val2': 9
        },
        'val2': {
            'operator': 'less_or_equal',
            'val1': 'prefix_length',
            'val2': 24
        }
    },
    'val2': {
        'operator': 'equals_any',
        'val1': 'tags',
        'val2': 'foobar'
    }
}
```

The *options* argument provides a way to alter the search result to assist in client implementations. Most options regard parent and children prefixes, that is the prefixes which contain the prefix(es) matching the search terms (parents) or the prefixes which are contained by the prefix(es) matching the search terms. The search options can also be used to limit the number of rows returned.

The following options are available:

- `parents_depth` - How many levels of parents to return. Set to `-1` to include all parents.
- `children_depth` - How many levels of children to return. Set to `-1` to include all children.
- `include_all_parents` - Include all parents, no matter what depth is specified.
- `include_all_children` - Include all children, no matter what depth is specified.
- `max_result` - The maximum number of prefixes to return (default `50`).
- `offset` - Offset the result list this many prefixes (default `0`).

The options above gives the possibility to specify how many levels of parent and child prefixes to return in addition to the prefixes that actually matched the search terms. This is done by setting the `parents_depth` and `children depth` keys in the *search_options* dict to an integer value. In addition to this it is possible to get all all parents and/or children included in the result set even though they are outside the limits set with `*_depth`. The extra prefixes included will have the attribute `display` set to `false` while the other ones (the actual search result togther with the ones included due to given depth) `display` set to `true`. This feature is usable obtain search results with some context given around them, useful for example when displaying prefixes in a tree without the need to implement client side IP address logic.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *`nipap.xmlrpc.NipapXMLRPC.search_prefix()`* for full understanding.

search_tag(*auth*, *query*, *search_options=None*)
Search Tags for entries matching 'query'

- ***auth*** **[BaseAuth]** AAA options.
- ***query*** **[dict_to_sql]** How the search should be performed.
- ***search_options*** **[options_dict]** Search options, see below.

Returns a list of dicts.

The *query* argument passed to this function is designed to be able to specify how quite advanced search operations should be performed in a generic format. It is internally expanded to a SQL WHERE-clause.

The *query* is a dict with three elements, where one specifies the operation to perform and the two other specifies its arguments. The arguments can themselves be *query* dicts, to build more complex queries.

The `operator` key specifies what operator should be used for the comparison. Currently the following operators are supported:

- `and` - Logical AND
- `or` - Logical OR
- `equals` - Equality; =
- `not_equals` - Inequality; !=
- `like` - SQL LIKE
- `regex_match` - Regular expression match
- `regex_not_match` - Regular expression not match

The `val1` and `val2` keys specifies the values which are subjected to the comparison. `val1` can be either any prefix attribute or an entire query dict. `val2` can be either the value you want to compare the prefix attribute to, or an entire *query* dict.

The search options can also be used to limit the number of rows returned or set an offset for the result.

The following options are available:

- `max_result` - The maximum number of prefixes to return (default `50`).
- `offset` - Offset the result list this many prefixes (default `0`).

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *`nipap.xmlrpc.NipapXMLRPC.search_asn()`* for full understanding.

search_vrf(*auth*, *query*, *search_options=None*)
Search VRF list for VRFs matching *query*.

- ***auth*** **[BaseAuth]** AAA options.
- ***query*** **[dict_to_sql]** How the search should be performed.
- ***search_options*** **[options_dict]** Search options, see below.

Returns a list of dicts.

The *query* argument passed to this function is designed to be able to specify how quite advanced search operations should be performed in a generic format. It is internally expanded to a SQL WHERE-clause.

The *query* is a dict with three elements, where one specifies the operation to perform and the two other specifies its arguments. The arguments can themselves be *query* dicts, to build more complex queries.

The `operator` key specifies what operator should be used for the comparison. Currently the following operators are supported:

- `and` - Logical AND
- `or` - Logical OR
- `equals` - Equality; =
- `not_equals` - Inequality; !=
- `like` - SQL LIKE
- `regex_match` - Regular expression match
- `regex_not_match` - Regular expression not match

The `val1` and `val2` keys specifies the values which are subjected to the comparison. `val1` can be either any prefix attribute or an entire query dict. `val2` can be either the value you want to compare the prefix attribute to, or an entire *query* dict.

Example 1 - Find the VRF whose VRF match '65000:123':

```
query = {
    'operator': 'equals',
    'val1': 'vrf',
    'val2': '65000:123'
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM vrf WHERE vrf = '65000:123'
```

Example 2 - Find vrf whose name or description regex matches 'test':

```
query = {
    'operator': 'or',
    'val1': {
        'operator': 'regex_match',
        'val1': 'name',
        'val2': 'test'
    },
    'val2': {
        'operator': 'regex_match',
        'val1': 'description',
        'val2': 'test'
    }
}
```

This will be expanded to the pseudo-SQL query:

```
SELECT * FROM vrf WHERE name ~* 'test' OR description ~* 'test'
```

The search options can also be used to limit the number of rows returned or set an offset for the result.

The following options are available:

- `max_result` - The maximum number of prefixes to return (default `50`).
- `offset` - Offset the result list this many prefixes (default `0`).

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *`nipap.xmlrpc.NipapXMLRPC.search_vrf()`* for full understanding.

smart_search_asn(*auth*, *query_str*, *search_options=None*, *extra_query=None*)
Perform a smart search operation among AS numbers

- ***auth* [BaseAuth]** AAA options.
- ***query_str* [string]** Search string
- ***search_options* [options_dict]** Search options. See `search_asn()`.
- ***extra_query* [dict_to_sql]** Extra search terms, will be AND:ed together with what is extracted from the query string.

Return a dict with three elements:

- `interpretation` - How the query string was interpreted.
- `search_options` - Various search_options.
- `result` - The search result.

The `interpretation` is given as a list of dicts, each explaining how a part of the search key was interpreted (ie. what ASN attribute the search operation was performed on).

The `result` is a list of dicts containing the search result.

The smart search function tries to convert the query from a text string to a *query* dict which is passed to the `search_asn()` function. If multiple search keys are detected, they are combined with a logical AND.

See the `search_asn()` function for an explanation of the *search_options* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for `nipap.xmlrpc.NipapXMLRPC.smart_search_asn()` for full understanding.

smart_search_pool(*auth*, *query_str*, *search_options=None*, *extra_query=None*)
Perform a smart search on pool list.

- ***auth* [BaseAuth]** AAA options.
- ***query_str* [string]** Search string
- ***search_options* [options_dict]** Search options. See `search_pool()`.
- ***extra_query* [dict_to_sql]** Extra search terms, will be AND:ed together with what is extracted from the query string.

Return a dict with three elements:

- `interpretation` - How the query string was interpreted.
- `search_options` - Various search_options.
- `result` - The search result.

The `interpretation` is given as a list of dicts, each explaining how a part of the search key was interpreted (ie. what pool attribute the search operation was performed on).

The `result` is a list of dicts containing the search result.

The smart search function tries to convert the query from a text string to a *query* dict which is passed to the `search_pool()` function. If multiple search keys are detected, they are combined with a logical AND.

It will basically just take each search term and try to match it against the name or description column with regex match.

See the *search_pool()* function for an explanation of the *search_options* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.smart_search_pool()* for full understanding.

smart_search_prefix(*auth*, *query_str*, *search_options=None*, *extra_query=None*)
Perform a smart search on prefix list.

- ***auth*** **[BaseAuth]** AAA options.
- ***query_str*** **[string]** Search string
- ***search_options*** **[options_dict]** Search options. See *search_prefix()*.
- ***extra_query*** **[dict_to_sql]** Extra search terms, will be AND:ed together with what is extracted from the query string.

Return a dict with three elements:

- `interpretation` - How the query string was interpreted.
- `search_options` - Various search_options.
- `result` - The search result.

The `interpretation` is given as a list of dicts, each explaining how a part of the search key was interpreted (ie. what prefix attribute the search operation was performed on).

The `result` is a list of dicts containing the search result.

The smart search function tries to convert the query from a text string to a *query* dict which is passed to the *search_prefix()* function. If multiple search keys are detected, they are combined with a logical AND.

It tries to automatically detect IP addresses and prefixes and put these into the *query* dict with "contains_within" operators and so forth.

See the *search_prefix()* function for an explanation of the *search_options* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.smart_search_prefix()* for full understanding.

smart_search_vrf(*auth*, *query_str*, *search_options=None*, *extra_query=None*)
Perform a smart search on VRF list.

- ***auth*** **[BaseAuth]** AAA options.
- ***query_str*** **[string]** Search string
- ***search_options*** **[options_dict]** Search options. See *search_vrf()*.
- ***extra_query*** **[dict_to_sql]** Extra search terms, will be AND:ed together with what is extracted from the query string.

Return a dict with three elements:

- `interpretation` - How the query string was interpreted.
- `search_options` - Various search_options.
- `result` - The search result.

The `interpretation` is given as a list of dicts, each explaining how a part of the search key was interpreted (ie. what VRF attribute the search operation was performed on).

The `result` is a list of dicts containing the search result.

The smart search function tries to convert the query from a text string to a *query* dict which is passed to the *search_vrf()* function. If multiple search keys are detected, they are combined with a logical AND.

It will basically just take each search term and try to match it against the name or description column with regex match or the VRF column with an exact match.

See the *search_vrf()* function for an explanation of the *search_options* argument.

This is the documentation of the internal backend function. It's exposed over XML-RPC, please also see the XML-RPC documentation for *nipap.xmlrpc.NipapXMLRPC.smart_search_vrf()* for full understanding.

nipap.backend.**requires_rw**(*f*)

Adds readwrite authorization

This will check if the user is a readonly user and if so reject the query. Apply this decorator to readwrite functions.

CHAPTER 3

pynipap - a Python NIPAP client library

pynipap is a Python client library for the NIPAP IP address planning system. It is structured as a simple ORM. To make it easy to maintain it's quite "thin", passing many arguments straight through to the backend. Thus, also the pynipap-specific documentation is quite thin. For in-depth information please look at the main *NIPAP API documentation*.

There are four ORM-classes:

- *VRF*
- *Pool*
- *Prefix*
- *Tag*

Each of these maps to the NIPAP objects with the same name. See the main *NIPAP API documentation* for an overview of the different object types and what they are used for.

There are also a few supporting classes:

- *AuthOptions* - Authentication options.

And a bunch of exceptions:

- *NipapError*
- *NipapNonExistentError*
- *NipapInputError*
- *NipapMissingInputError*
- *NipapExtraneousInputError*
- *NipapNoSuchOperatorError*
- *NipapValueError*
- *NipapDuplicateError*
- *NipapAuthError*
- *NipapAuthenticationError*
- *NipapAuthorizationError*

3.1 General usage

pynipap has been designed to be simple to use.

3.1.1 Preparations

Make sure that pynipap is accessible in your *sys.path*, you can test it by starting a python shell and running:

```
import pynipap
```

If that works, you are good to go!

To simplify your code slightly, you can import the individual classes into your main namespace:

```
import pynipap
from pynipap import VRF, Pool, Prefix
```

Before you can access NIPAP you need to specify the URL to the NIPAP XML-RPC service and the authentication options to use for your connection. NIPAP has a authentication system which is somewhat involved, see the main NIPAP documentation.

The URL, including the user credentials, is set in the pynipap module variable *xmlrpc_uri* as so:

```
pynipap.xmlrpc_uri = "http://user:pass@127.0.0.1:1337/XMLRPC"
```

If you want to access the API externally, from another host, update the corresponding lines in the nipap.conf file. Here you can also change the port.

```
listen = 0.0.0.0              ; IP address to listen on.
port = 1337             ; XML-RPC listen port (change requires restart)
```

The minimum authentication options which we need to set is the *authoritative_source* option, which specifies what system is accessing NIPAP. This is logged for each query which alters the NIPAP database and attached to each prefix which is created or edited. Well-behaved clients are required to honor this and verify that the user really want to alter the prefix, when trying to edit a prefix which last was edited by another system. The *AuthOptions* class is a class with a shared state, similar to a singleton class; that is, when a first instance is created each consecutive instances will be copies of the first one. In this way the authentication options can be accessed from all of the pynipap classes.

```
a = AuthOptions({
        'authoritative_source': 'my_fancy_nipap_client'
    })
```

After this, we are good to go!

3.1.2 Accessing data

To fetch data from NIPAP, a set of static methods (@classmethod) has been defined in each of the ORM classes. They are:

- `get()` - Get a single object from its ID.
- `list()` - List objects matching a simple criteria.
- `search()` - Perform a full-blown search.
- `smart_search()` - Perform a magic search from a string.

Each of these functions return either an instance of the requested class (*VRF*, *Pool*, *Prefix*) or a list of instances. The `search()` and `smart_search()` functions also embeds the lists in dicts which contain search meta data.

The easiest way to get data out of NIPAP is to use the `get()`-method, given that you know the ID of the object you want to fetch:

```
# Fetch VRF with ID 1 and print its name
vrf = VRF.get(1)
print(vrf.name)
```

To list all objects each object has a `list()`-function.

```
# list all pools
pools = Pool.list()

# print the name of the pools
for p in pools:
    print(p.name)
```

Each of the list functions can also take a *spec*-dict as a second argument. With the spec you can perform a simple search operation by specifying object attribute values.

```
# List pools with a default type of 'assignment'
pools = Pool.list({ 'default_type': 'assignment' })
```

3.1.3 Performing searches

Searches are easiest when using the object's `smart_search()`-method:

```
#Returns a dict which includes search metadata and
#a 'result' : [array, of, prefix, objects]
search_result = Prefix.smart_search('127.0.0.0/8')
prefix_objects = search_result['result']
prefix_objects[0].description
prefix_objects[0].prefix
```

You can also send query filters.

```
#Find the prefix for Vlan 901
vlan = 901
vlan_query = { 'val1': 'vlan', 'operator': 'equals', 'val2': vlan }
vlan_901 = Prefix.smart_search('', { }, vlan_query)['result'][0]
vlan_901.vlan
```

The following operators can be used.

```
* 'and'
* 'or'
* 'equals_any'
* '='
* 'equals'
* '<'
* 'less'
* '<='
* 'less_or_equal'
* '>'
* 'greater'
* '>='
* 'greater_or_equal'
```

```
* 'is'
* 'is_not'
* '!='
* 'not_equals'
* 'like': '
* 'regex_match'
* 'regex_not_match'
* '>>':
* 'contains'
* '>>='
* 'contains_equals'
* '<<'
* 'contained_within'
* '<<='
* 'contained_within_equals'
```

3.1.4 Saving changes

Changes made to objects are not automatically saved. To save the changes, simply run the object's `save()`-method:

```
vrf.name = "Spam spam spam"
vrf.save()
```

3.2 Error handling

As is customary in Python applications, an error results in an exception being thrown. All pynipap exceptions extend the main exception *NipapError*. A goal with the pynipap library has been to make the XML-RPC-channel to the backend as transparent as possible, so the XML-RPC Faults which the NIPAP server returns in case of errors are converted and re-thrown as new exceptions which also they extend *NipapError*, for example the NipapDuplicateError which is thrown when a duplicate key error occurs in NIPAP.

3.3 Classes

class pynipap.**AuthOptions**(*options=None*)
A global-ish authentication option container.

Note that this essentially is a global variable. If you handle multiple queries from different users, you need to make sure that the AuthOptions-instances are set to the current user's.

exception pynipap.**NipapAuthError**
General NIPAP AAA error

exception pynipap.**NipapAuthenticationError**
Authentication failed.

exception pynipap.**NipapAuthorizationError**
Authorization failed.

exception pynipap.**NipapDuplicateError**
A duplicate entry was encountered

exception pynipap.**NipapError**
A generic NIPAP model exception.

All errors thrown from the NIPAP model extends this exception.

exception pynipap.**NipapExtraneousInputError**
Extraneous input

Most input is passed in dicts, this could mean an unknown key in a dict.

exception pynipap.**NipapInputError**
Something wrong with the input we received

A general case.

exception pynipap.**NipapMissingInputError**
Missing input

Most input is passed in dicts, this could mean a missing key in a dict.

exception pynipap.**NipapNoSuchOperatorError**
A non existent operator was specified.

exception pynipap.**NipapNonExistentError**
Thrown when something can not be found.

For example when a given ID can not be found in the NIPAP database.

exception pynipap.**NipapValueError**
Something wrong with a value we have

For example, trying to send an integer when an IP address is expected.

class pynipap.**Pool**
An address pool.

classmethod from_dict(*parm*, *pool=None*)
Create new Pool-object from dict.

Suitable for creating objects from XML-RPC data. All available keys must exist.

classmethod get(*id*)
Get the pool with id 'id'.

classmethod list(*spec=None*)
List pools.

Maps to the function *nipap.backend.Nipap.list_pool()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

remove()
Remove pool.

Maps to the function *nipap.backend.Nipap.remove_pool()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

save()
Save changes made to pool to NIPAP.

If the object represents a new pool unknown to NIPAP (attribute *id* is *None*) this function maps to the function *nipap.backend.Nipap.add_pool()* in the backend, used to create a new pool. Otherwise it maps to the function *nipap.backend.Nipap.edit_pool()* in the backend, used to modify the pool. Please see the documentation for the backend functions for information regarding input arguments and return values.

classmethod search(*query*, *search_opts=None*)
Search pools.

Maps to the function *nipap.backend.Nipap.search_pool()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

classmethod smart_search(*query_string*, *search_options=None*, *extra_query=None*)
Perform a smart pool search.

Maps to the function *nipap.backend.Nipap.smart_search_pool()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

class pynipap.**Prefix**
A prefix.

classmethod find_free(*vrf*, *args*)
Finds a free prefix.

Maps to the function *nipap.backend.Nipap.find_free_prefix()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

classmethod from_dict(*pref*, *prefix=None*)
Create a Prefix object from a dict.

Suitable for creating Prefix objects from XML-RPC input.

classmethod get(*id*)
Get the prefix with id 'id'.

classmethod list(*spec=None*)
List prefixes.

Maps to the function *nipap.backend.Nipap.list_prefix()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

remove(*recursive=False*)
Remove the prefix.

Maps to the function *nipap.backend.Nipap.remove_prefix()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

save(*args=None*)
Save prefix to NIPAP.

If the object represents a new prefix unknown to NIPAP (attribute *id* is *None*) this function maps to the function *nipap.backend.Nipap.add_prefix()* in the backend, used to create a new prefix. Otherwise it maps to the function *nipap.backend.Nipap.edit_prefix()* in the backend, used to modify the VRF. Please see the documentation for the backend functions for information regarding input arguments and return values.

classmethod search(*query*, *search_opts=None*)
Search for prefixes.

Maps to the function *nipap.backend.Nipap.search_prefix()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

classmethod smart_search(*query_string*, *search_options=None*, *extra_query=None*)
Perform a smart prefix search.

Maps to the function *nipap.backend.Nipap.smart_search_prefix()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

class pynipap.**Pynipap**(*id=None*)
A base class for the pynipap model classes.

All Pynipap classes which maps to data in NIPAP (*VRF*, *Pool*, *Prefix*) extends this class.

id = None
Internal database ID of object.

class pynipap.**Tag**(*id=None*)
A Tag.

classmethod from_dict(*tag=None*)
Create new Tag-object from dict.

Suitable for creating objects from XML-RPC data. All available keys must exist.

name = None
The Tag name

classmethod search(*query*, *search_opts=None*)
Search tags.

For more information, see the backend function *nipap.backend.Nipap.search_tag()*.

class pynipap.**VRF**
A VRF.

description = None
VRF description, as a string.

free_addresses_v4 = None
Number of free IPv4 addresses in this VRF

free_addresses_v6 = None
Number of free IPv6 addresses in this VRF

classmethod from_dict(*parm*, *vrf=None*)
Create new VRF-object from dict.

Suitable for creating objects from XML-RPC data. All available keys must exist.

classmethod get(*id*)
Get the VRF with id 'id'.

classmethod list(*vrf=None*)
List VRFs.

Maps to the function *nipap.backend.Nipap.list_vrf()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

name = None
The name of the VRF, as a string.

num_prefixes_v4 = None
Number of IPv4 prefixes in this VRF

num_prefixes_v6 = None
Number of IPv6 prefixes in this VRF

remove()
Remove VRF.

Maps to the function *nipap.backend.Nipap.remove_vrf()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

rt = None
The VRF RT, as a string (x:y or x.x.x.x:y).

save()
Save changes made to object to NIPAP.

If the object represents a new VRF unknown to NIPAP (attribute *id* is *None*) this function maps to the function *nipap.backend.Nipap.add_vrf()* in the backend, used to create a new VRF. Otherwise it maps to the function *nipap.backend.Nipap.edit_vrf()* in the backend, used to modify the VRF. Please see the documentation for the backend functions for information regarding input arguments and return values.

classmethod search(*query*, *search_opts=None*)
Search VRFs.

Maps to the function *nipap.backend.Nipap.search_vrf()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

classmethod smart_search(*query_string*, *search_options=None*, *extra_query=None*)
Perform a smart VRF search.

Maps to the function *nipap.backend.Nipap.smart_search_vrf()* in the backend. Please see the documentation for the backend function for information regarding input arguments and return values.

total_addresses_v4 = None
Total number of IPv4 addresses in this VRF

total_addresses_v6 = None
Total number of IPv6 addresses in this VRF

used_addresses_v4 = None
Number of used IPv4 addresses in this VRF

used_addresses_v6 = None
Number of used IPv6 addresses in this VRF

class pynipap.**XMLRPCConnection**
Handles a shared XML-RPC connection.

pynipap.**nipap_db_version**()
Get schema version of database we're connected to.

Maps to the function nipap.backend.Nipap._get_db_version() in the backend. Please see the documentation for the backend function for information regarding the return value.

pynipap.**nipapd_version**()
Get version of nipapd we're connected to.

Maps to the function *nipap.xmlrpc.NipapXMLRPC.version()* in the XML-RPC API. Please see the documentation for the XML-RPC function for information regarding the return value.

CHAPTER 4

NIPAP XML-RPC

This module contains the actual functions presented over the XML-RPC API. All functions are quite thin and mostly wrap around the functionality provided by the backend module.

class nipap.xmlrpc.**NipapXMLRPC**

NIPAP XML-RPC API

add_asn(**args*, ***kwargs*)

Add a new ASN.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***attr* [struct]** ASN attributes.

Returns the ASN.

add_pool(**args*, ***kwargs*)

Add a pool.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***attr* [struct]** Attributes which will be set on the new pool.

Returns ID of created pool.

add_prefix(**args*, ***kwargs*)

Add a prefix.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***attr* [struct]** Attributes to set on the new prefix.
- ***args* [srgs]** Arguments for addition of prefix, such as what pool or prefix it should be allocated from.

Returns ID of created prefix.

add_vrf(**args*, ***kwargs*)

Add a new VRF.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***attr* [struct]** VRF attributes.

Returns the internal database ID for the VRF.

db_version(**args*, ***kwargs*)
Returns schema version of nipap psql db

Returns a string.

echo(**args*, ***kwargs*)
An echo function

An API test function which simply echoes what is is passed in the 'message' element in the args-dict..

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***message* [string]** String to echo.
- ***sleep* [integer]** Number of seconds to sleep before echoing.

Returns a string.

edit_asn(**args*, ***kwargs*)
Edit an ASN.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***asn* [integer]** An integer specifying which ASN to edit.
- ***attr* [struct]** ASN attributes.

edit_pool(**args*, ***kwargs*)
Edit pool.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***pool* [struct]** Specifies pool attributes to match.
- ***attr* [struct]** Pool attributes to set.

edit_prefix(**args*, ***kwargs*)
Edit prefix.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***prefix* [struct]** Prefix attributes which describes what prefix(es) to edit.
- ***attr* [struct]** Attribuets to set on the new prefix.

edit_vrf(**args*, ***kwargs*)
Edit a VRF.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***vrf* [struct]** A VRF spec specifying which VRF(s) to edit.
- ***attr* [struct]** VRF attributes.

find_free_prefix(**args*, ***kwargs*)
Find a free prefix.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***args* [struct]** Arguments for the find_free_prefix-function such as what prefix or pool to allocate from.

list_asn(**args*, ***kwargs*)
List ASNs.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***asn* [struct]** Specifies ASN attributes to match (optional).

Returns a list of ASNs matching the ASN spec as a list of structs.

list_pool(**args*, ***kwargs*)
List pools.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***pool* [struct]** Specifies pool attributes which will be matched.

Returns a list of structs describing the matching pools.

list_prefix(**args*, ***kwargs*)
List prefixes.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***prefix* [struct]** Prefix attributes to match.

Returns a list of structs describing the matching prefixes.

Certain values are casted from numbers to strings because XML-RPC simply cannot handle anything bigger than an integer.

list_vrf(**args*, ***kwargs*)
List VRFs.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***vrf* [struct]** Specifies VRF attributes to match (optional).

Returns a list of structs matching the VRF spec.

remove_asn(**args*, ***kwargs*)
Removes an ASN.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***asn* [integer]** An ASN.

remove_pool(**args*, ***kwargs*)
Remove a pool.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***pool* [struct]** Specifies what pool(s) to remove.

remove_prefix(**args*, ***kwargs*)
Remove a prefix.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***prefix* [struct]** Attributes used to select what prefix to remove.
- ***recursive* [boolean]** When set to 1, also remove child prefixes.

remove_vrf(**args*, ***kwargs*)
Removes a VRF.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***vrf* [struct]** A VRF spec.

search_asn(**args*, ***kwargs*)
Search ASNs.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query* [struct]** A struct specifying the search query.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result and the search options used.

search_pool(**args*, ***kwargs*)
Search for pools.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query* [struct]** A struct specifying the search query.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result and the search options used.

search_prefix(**args*, ***kwargs*)
Search for prefixes.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query* [struct]** A struct specifying the search query.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing the search result together with the search options used.

Certain values are casted from numbers to strings because XML-RPC simply cannot handle anything bigger than an integer.

search_vrf(**args*, ***kwargs*)
Search for VRFs.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query* [struct]** A struct specifying the search query.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result and the search options used.

smart_search_asn(**args*, ***kwargs*)
Perform a smart search among ASNs.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query_string* [string]** The search string.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result, interpretation of the search string and the search options used.

smart_search_pool(**args*, ***kwargs*)
Perform a smart search.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query* [string]** The search string.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result, interpretation of the query string and the search options used.

smart_search_prefix(**args*, ***kwargs*)
Perform a smart search.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.
- ***query_string* [string]** The search string.
- ***search_options* [struct]** Options for the search query, such as limiting the number of results returned.
- ***extra_query* [struct]** Extra search terms, will be AND:ed together with what is extracted from the query string.

Returns a struct containing search result, interpretation of the query string and the search options used.

Certain values are casted from numbers to strings because XML-RPC simply cannot handle anything bigger than an integer.

smart_search_vrf(**args*, ***kwargs*)
Perform a smart search.

Valid keys in the *args*-struct:

- ***auth* [struct]** Authentication options passed to the `AuthFactory`.

- ***query_string*** **[string]** The search string.
- ***search_options*** **[struct]** Options for the search query, such as limiting the number of results returned.

Returns a struct containing search result, interpretation of the search string and the search options used.

version(**args*, ***kwargs*)
Returns nipapd version

Returns a string.

nipap.xmlrpc.**authenticate**()
Sends a 401 response that enables basic auth

nipap.xmlrpc.**requires_auth**(*f*)
Class decorator for XML-RPC functions that requires auth

CHAPTER 5

Authentication library

A base authentication & authorization module.

Includes the base class BaseAuth.

5.1 Authentication and authorization in NIPAP

NIPAP offers basic authentication with two different backends, a simple two-level authorization model and a trust-system for simplifying system integration.

Readonly users are only authorized to run queries which do not modify any data in the database. No further granularity of access control is offered at this point.

Trusted users can perform operations which will be logged as performed by another user. This feature is meant for system integration, for example to be used by a NIPAP client which have its own means of authentication users; say for example a web application supporting the NTLM single sign-on feature. By letting the web application use a trusted account to authenticate against the NIPAP service, it can specify the username of the end-user, so that audit logs will be written with the correct information. Without the trusted-bit, all queries performed by end-users through this system would look like they were performed by the system itself.

The NIPAP auth system also has a concept of authoritative source. The authoritative source is a string which simply defines what system is the authoritative source of data for a prefix. Well-behaved clients SHOULD present a warning to the user when trying to alter a prefix with an authoritative source different than the system itself, as other system might depend on the information being unchanged. This is however, by no means enforced by the NIPAP service.

5.2 Authentication backends

Two authentication backends are shipped with NIPAP:

- LdapAuth - authenticates users against an LDAP server
- SqliteAuth - authenticates users against a local SQLite-database

The authentication classes presented here are used both in the NIPAP web UI and in the XML-RPC backend. So far only the SqliteAuth backend supports trusted and readonly users.

What authentication backend to use can be specified by suffixing the username with @‘backend‘, where *backend* is set in the configuration file. If not defined, a (configurable) default backend is used.

5.3 Authentication options

With each NIPAP query authentication options can be specified. The authentication options are passed as a dict with the following keys taken into account:

- `authoritative_source` - Authoritative source for the query.
- `username` - Username to impersonate, requires authentication as trusted user.
- `full_name` - Full name of impersonated user.
- `readonly` - True for read-only users

5.4 Classes

exception `nipap.authlib.`**`AuthError`**
General auth exception.

class `nipap.authlib.`**`AuthFactory`**
An factory for authentication backends.

`get_auth`(*username*, *password*, *authoritative_source*, *auth_options=None*)
Returns an authentication object.

Examines the auth backend given after the '@' in the username and returns a suitable instance of a subclass of the BaseAuth class.

- ***username*** **[string]** Username to authenticate as.
- ***password*** **[string]** Password to authenticate with.
- ***authoritative_source*** **[string]** Authoritative source of the query.
- ***auth_options*** **[dict]** A dict which, if authenticated as a trusted user, can override *username* and *authoritative_source*.

`reload`()
Reload AuthFactory.

exception `nipap.authlib.`**`AuthSqliteError`**
Problem with the Sqlite database

exception `nipap.authlib.`**`AuthenticationFailed`**
Authentication failed.

exception `nipap.authlib.`**`AuthorizationFailed`**
Authorization failed.

class `nipap.authlib.`**`BaseAuth`**(*username*, *password*, *authoritative_source*, *auth_backend*, *auth_options=None*)
A base authentication class.

All authentication modules should extend this class.

`authenticate`()
Verify authentication.

Returns True/False dependant on whether the authentication succeeded or not.

`authorize`()
Verify authorization.

Check if a user is authorized to perform a specific operation.

class nipap.authlib.**LdapAuth**(*name*, *username*, *password*, *authoritative_source*, *auth_options=None*)
An authentication and authorization class for LDAP auth.

authenticate()
Verify authentication.

Returns True/False dependant on whether the authentication succeeded or not.

class nipap.authlib.**SqliteAuth**(*name*, *username*, *password*, *authoritative_source*, *auth_options=None*)
An authentication and authorization class for local auth.

add_user(*username*, *password*, *full_name=None*, *trusted=False*, *readonly=False*)
Add user to SQLite database.

- ***username*** **[string]** Username of new user.
- ***password*** **[string]** Password of new user.
- ***full_name*** **[string]** Full name of new user.
- ***trusted*** **[boolean]** Whether the new user should be trusted or not.
- ***readonly*** **[boolean]** Whether the new user can only read or not

authenticate()
Verify authentication.

Returns True/False dependant on whether the authentication succeeded or not.

get_user(*username*)
Fetch the user from the database

The function will return None if the user is not found

list_users()
List all users.

modify_user(*username*, *data*)
Modify user in SQLite database.

Since username is used as primary key and we only have a single argument for it we can't modify the username right now.

remove_user(*username*)
Remove user from the SQLite database.

- ***username*** **[string]** Username of user to remove.

CHAPTER 6

NIPAP release handling

This document tries to describe most aspects of the release handling of NIPAP.

6.1 Packaging

NIPAP is packaged into a number of packages. There is the backend parts in form of the NIPAP XML-RPC daemon (hereinafter referred to as nipapd) that is just the actual daemon. Since it depends on PostgreSQL it has its own package while most of its actual code lies in the shared 'nipap' Python module which is packaged into nipap-common. The same Python module is used by the web frontend and it is for this reason it is packaged into the nipap-common package. nipapd and nipap-common share a common source directory (nipap) and thus also share version number.

The client library/module in pynipap is packaged into a package with the same name and has its own version number, ie it is not correlated in any way with the version of the backend parts.

6.2 Version numbering

Version numbering of NIPAP is in the form of major.minor.patch. Major releases are milestones for when a number of large improvements have been implemented and are stable while minor releases will increment for most new features. Patch releases will only include smaller bug fixes or other similarily small changes.

Major releases should generally be released after a number of features have proven to be stable and fairly bug free. For example, we are at 1.0.0. A couple of features are implemented over a period of time and for each, a new minor version is released so we are now at 1.7.0. After some time in production and seeing that these features behave as expect it, version 2.0.0 can be released as a "trusted release" with basically the same feature set as 1.7.0 but now marked as a stable and major version.

This implies that major version can be trusted, while the risk for bugs are higher in minor versions and again smaller with patch releases.

6.3 Debian repository

The repo itself is hosted by GitHub through their support for building a webpage via the branch gh-pages, please see http://help.github.com/pages/ for more information on that.

The Makefile includes a two targets (debrepo-testing & debrepo-stable) to build the necessary files for a debian repo and put this in the correct place. As soon as a commit is pushed, github will copy the files and produce a webpage

accessible via http://<github user>.github.com/<project name> (ie http://spritelink.github.com/NIPAP). We use this to build a simple apt repository hosted on GitHub.

To update the apt repo, build the debian packages, then run 'make debrepo-testing' in the project root. This will put the packages in the testing repo. Commit to the gh-pages branch and then push and it should all work! :) Once a version is considered stable, run 'make debrepo-stable' to copy the packages from the testing branch into stable. Again, commit to gh-pages and so forth. Please see "Rolling the deb repo" section for more details.

6.4 NEWS / Changelog

There is a NEWS file outlining the differences with every version. It can either be updated as changes are made or just before a new release is rolled by going through the git log since the last version and making sure everything worth mentioning is in the NEWS file.

Note how a NEWS file is usually used to document changes between versions of a package while a Changelog file is used to convey information about changes between commits. The Debian changelog included with packages normally do not follow this "changelog principle" but rather they are usually used to document changes to the actual packaging or to patches and changes made by the maintainer of a package.

As documented on http://www.debian.org/doc/debian-policy/footnotes.html#f16, it is under certain circumstances perfectly fine to essentially have the same file as Debian changelog and the project "changelog" (or NEWS file as is more correct). One such instance is when the Debian package closely follows the project, as is the case with NIPAP. Thus, the NEWS file will be very similar to the Debian changelog.

Debian style package managers are able to fetch the Debian changelog file from repositories and can thus display the changes between versions before installing a package.

6.5 Build prerequisites

Install the following debian packages:

```
apt-get install debhelper python-docutils python-setuptools python3-all \
        python3-docutils python3-setuptools reprepro
```

6.6 Rolling a new version

Update the NEWS file as described above.

If you have changes to the database, don't forget to increment the version number in sql/ip_net.sql.

From the project root, run:: make bumpversion

This will automatically update the debian changelog based on the content of the NEWS file. You can bump the version for a single component (such as pynipap) by running the same command in the directory of that component.

After having built packages for the new version, tag the git repo with the new version number:

```
git tag vX.Y.Z
```

And for pushing to git:: git push origin refs/tags/vX.Y.Z

6.7 Rolling the deb repo

Debian stable is the primary production platform targeted by NIPAP and new releases should be put in our Debian repo.

To update the deb repo, make sure you are on branch 'master' and then build the bebian packages with:

```
make builddeb
```

Then checkout the 'gh-pages' branch and add them to the repo:: git checkout gh-pages

Start by adding the packages the testing repo:: make debrepo-testing

Once the new version has been tested out for a bit, it is time to copy it to stable, using:

```
make debrepo-stable
```

Regardless if you are putting the packages in testing or stable, you need to actually push them to the github repo. Make sure the new files are added to git, commit and push:

```
git add --all repos
git commit -a -m "Add nipapd vX.Y.Z to debian STABLE|TESTING repo"
git push
```

Once a stable version is release, update readthedocs.org to point to the latest tag and write a post on Google+ in the NIPAP community and share it from the NIPAP account.

6.8 Uploading to PyPi

pynipap should be available on PyPi:: cd pynipap python setup.py sdist upload

6.9 Manually rolling a new version

You probably don't want to roll a new release manually but this might help in understanding what happens behind the scenes.

The different packages are first built as Python easy_install / distutils packages which are later mangled into a debian package. To roll a new version there are thus two places that need updating; the first is where easy_install gets its version number. You can look into setup.py and see the version line and which file & variable it refers too.

See the following files for version info: nipap/nipap/__init__.py pynipap/pynipap.py nipap-cli/nipap_cli/__init__.py nipap-www/nipapwww/__init__.py

To roll a new release, update the Python file with the new version number according to the above instructions. After that, run 'dch -v <version>', where version is the version number previously entered into the Python file postfixed with -1. Ie, if you want to release 1.0.0, set that in the Python file and use 1.0.0-1 for dch. The -1 is the version of the debian package for non-native packages. Non-native packages are all packages that are not exlusively packaged for debian. If you want to release a new debian release, for example if you made changes to the actual packaging but not the source of the project, just increment the -x number.

When dch launches an editor for editing the changelog. Copy the content of the NEWS file into the Debian changelog (see previous chapten "NEWS / Changelog" for more information). Make sure the formatting aligns and save the file.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

n

p

N

P

R

S

T

U

V

X