pynetbox Documentation

Release 7.0.1

Zach Moody

Jan 23, 2023

Contents:

class pynetbox.core.endpoint.**Endpoint**(*api*, *app*, *name*, *model=None*)

Represent actions available on endpoints in the Netbox API.

Takes `name` and `app` passed from App() and builds the correct url to make queries to and the proper Response object to return results in.

Parameters

- **api** (*obj*) – Takes *Api* created at instantiation.
- **app** (*obj*) – Takes *App*.
- **name** (*str*) – Name of endpoint passed to App().
- **model** (*obj, optional*) – Custom model for given app.

Note: In order to call NetBox endpoints with dashes in their names you should convert the dash to an underscore. (E.g. querying the ip-addresses endpoint is done with `nb.ipam.ip_addresses.all()`.)

---

all(*limit=0*, *offset=None*)

Queries the 'ListView' of a given endpoint.

Returns all objects from an endpoint.

Parameters

- **limit** (*int, optional*) – Overrides the max page size on paginated returns. This defines the number of records that will be returned with each query to the Netbox server. The queries will be made as you iterate through the result set.
- **offset** (*int, optional*) – Overrides the offset on paginated returns.

Returns A *RecordSet* object.

Examples

```
>>> devices = list(nb.dcim.devices.all())
>>> for device in devices:
...     print(device.name)
...
test1-leaf1
test1-leaf2
test1-leaf3
>>>
```

If you want to iterate over the results multiple times then encapsulate them in a list like this: >>> devices = list(nb.dcim.devices.all())

This will cause the entire result set to be fetched from the server.

choices()

Returns all choices from the endpoint.

The returned dict is also saved in the endpoint object (in `_choices` attribute) so that later calls will return the same data without recurring requests to NetBox. When using `.choices()` in long-running applications, consider restarting them whenever NetBox is upgraded, to prevent using stale choices data.

Returns Dict containing the available choices.

Example

```
>>> from pprint import pprint
>>> pprint(nb.ipam.ip_addresses.choices())
{'role': [{'display_name': 'Loopback', 'value': 'loopback'},
          {'display_name': 'Secondary', 'value': 'secondary'},
          {'display_name': 'Anycast', 'value': 'anycast'},
          {'display_name': 'VIP', 'value': 'vip'},
          {'display_name': 'VRRP', 'value': 'vrrp'},
          {'display_name': 'HSRP', 'value': 'hsrp'},
          {'display_name': 'GLBP', 'value': 'glbp'},
          {'display_name': 'CARP', 'value': 'carp'}],
 'status': [{'display_name': 'Active', 'value': 'active'},
            {'display_name': 'Reserved', 'value': 'reserved'},
            {'display_name': 'Deprecated', 'value': 'deprecated'},
            {'display_name': 'DHCP', 'value': 'dhcp'},
            {'display_name': 'SLAAC', 'value': 'slaac'}]}
>>>
```

count(**args*, ***kwargs*)

Returns the count of objects in a query.

Takes named arguments that match the usable filters on a given endpoint. If an argument is passed then it's used as a freeform search argument if the endpoint supports it. If no arguments are passed the count for all objects on an endpoint are returned.

Parameters

- ***args** (*str, optional*) – Freeform search string that's accepted on given endpoint.
- ****kwargs** (*str, optional*) – Any search argument the endpoint accepts can be added as a keyword arg.

Returns Integer with count of objects returns by query.

Examples

To return a count of objects matching a named argument filter.

```
>>> nb.dcim.devices.count(site='tst1')
5827
>>>
```

To return a count of objects on an entire endpoint.

```
>>> nb.dcim.devices.count()
87382
>>>
```

create(**args*, ***kwargs*)

Creates an object on an endpoint.

Allows for the creation of new objects on an endpoint. Named arguments are converted to json properties, and a single object is created. NetBox's bulk creation capabilities can be used by passing a list of dictionaries as the first argument.

Parameters

- ***args** (*list*) – A list of dictionaries containing the properties of the objects to be created.
- ****kwargs** (*str*) – key/value strings representing properties on a json object.

Returns A list or single *Record* object depending on whether a bulk creation was requested.

Examples

Creating an object on the *devices* endpoint:

```
>>> device = netbox.dcim.devices.create(
...     name='test',
...     device_role=1,
... )
>>>
```

Creating an object on the 'devices' endpoint using *.get()* to get ids.

```
>>> device = netbox.dcim.devices.create(
...      name = 'test1',
...      site = netbox.dcim.devices.get(name='site1').id,
...      location = netbox.dcim.locations.get(name='Row 1').id,
...      rack = netbox.dcim.racks.get(facility_id=1).id,
...      device_type = netbox.dcim.device_types.get(slug='server-type-1').id,
...      status='active',
...      )
>>>
```

Use bulk creation by passing a list of dictionaries:

```
>>> nb.dcim.devices.create([
...      {
...          "name": "test1-core3",
...          "device_role": 3,
...          "site": 1,
...          "device_type": 1,
...          "status": 1
...      },
...      {
...          "name": "test1-core4",
...          "device_role": 3,
...          "site": 1,
...          "device_type": 1,
...          "status": 1
...      }
... ])
```

Create a new device type.

```
>>> device_type = netbox.dcim.devices.create(
...      manufacturer = netbox.dcim.manufacturers.get(name='manufacturer-name
↪').id,
...      model = 'device-type-name',
...      slug = 'device-type-slug',
...      subdevice_role = 'child or parent', #optional field - requred if␣
↪creating a device type to be used by child devices
...      u_height = unit_height, #can only equal 0 if the device type is for a␣
↪child device - requires subdevice_role='child' if that is the case
...      custom_fields = {'cf_1' : 'custom data 1'}
...      )
```

Create a device bay and child device.

```
>>> device_bay = netbox.dcim.device_bays.create(
...     device = netbox.dcim.devices.get(name='parent device').id, # device␣
↪the device bay is located
...     name = 'Bay 1'
...     )
>>> child_device = netbox.dcim.devices.create(
...     name = 'child device',
...     site = netbox.dcim.devices.get(name='test-site').id,
...     location = netbox.dcim.locations.get(name='row-1').id,
...     tenant = netbox.tenancy.tenants.get(name='tenant-1').id,
...     manufactuer = netbox.dcim.manufacturers.get(name='test-m').id,
...     rack = netbox.dcim.racks.get(name='Test Rack').id,
...     device_type = netbox.dcim.device.types.get(slug='test-server').id,
↪#easier to get device_type id by search by its slug rather than by its name
...     )
>>> get_device_bay = netbox.dcim.device_bays.get(name='Bay 1')
>>> get_child_device = netbox.dcim.devices.get(name='child device')
>>> get_device_bay.installed_device = get_child_device
>>> get_device_bay.save()
```

Create a network interface

```
>>> interface = netbox.dcim.interfaces.get(name="interface-test", device=
↪"test-device")
>>> netbox_ip = netbox.ipam.ip_addresses.create(
... address = "ip-address",
... tenant = netbox.tenancy.tenants.get(name='tenant-1').id,
... tags = [{'name':'Tag 1'}],
... )
>>> #assign IP Address to device's network interface
>>> netbox_ip.assigned_object = interface
>>> netbox_ip.assigned_object_id = interface.id
>>> netbox_ip.assigned_object_type = 'dcim.interface'
>>> # add dns name to IP Address (optional)
>>> netbox_ip.dns_name = "test.dns.local"
>>> # save changes to IP Address
>>> netbox_ip.save()
```

delete(*objects*)

Bulk deletes objects on an endpoint.

Allows for batch deletion of multiple objects from a single endpoint

Parameters **objects** (*list*) – A list of either ids or Records or a single RecordSet to delete.

Returns True if bulk DELETE operation was successful.

Examples

Deleting all *devices*:

```
>>> netbox.dcim.devices.delete(netbox.dcim.devices.all(0))
>>>
```

Use bulk deletion by passing a list of ids:

```
>>> netbox.dcim.devices.delete([2, 243, 431, 700])
>>>
```

Use bulk deletion to delete objects eg. when filtering on a *custom_field*: >>> netbox.dcim.devices.delete([>>> d for d in netbox.dcim.devices.all(0) >>> if d.custom_fields.get('field', False) >>>]) >>>

filter(**args*, ***kwargs*)

Queries the 'ListView' of a given endpoint.

Takes named arguments that match the usable filters on a given endpoint. If an argument is passed then it's used as a freeform search argument if the endpoint supports it.

Parameters

- ***args** (*str, optional*) – Freeform search string that's accepted on given endpoint.
- ****kwargs** (*str, optional*) – Any search argument the endpoint accepts can be added as a keyword arg.
- **limit** (*int, optional*) – Overrides the max page size on paginated returns. This defines the number of records that will be returned with each query to the Netbox server. The queries will be made as you iterate through the result set.
- **offset** (*int, optional*) – Overrides the offset on paginated returns.

Returns A *RecordSet* object.

Examples

To return a list of objects matching a named argument filter.

```
>>> devices = nb.dcim.devices.filter(role='leaf-switch')
>>> for device in devices:
...     print(device.name)
...
test1-leaf1
test1-leaf2
test1-leaf3
>>>
```

```
>>> devices = nb.dcim.devices.filter(site='site-1')
>>> for device in devices:
...     print(device.name)
...
test1-a2-leaf1
test2-a2-leaf2
>>>
```

If we want to filter by site, location, tenant, or fields which have a display name and a slug, the slug has to be used for filtering.

Note: If a keyword argument is incorrect a *TypeError* will not be returned by pynetbox. Instead, all records filtered up to the last correct keyword argument. For example, if we used *site="Site 1"* instead of *site=site-1* when using filter on the devices endpoint, then pynetbox will return **all** devices across all sites instead of devices at Site 1.

Using a freeform query along with a named argument.

```
>>> devices = nb.dcim.devices.filter('a3', role='leaf-switch')
>>> for device in devices:
...     print(device.name)
...
```

(continues on next page)

(continued from previous page)

```
test1-a3-leaf1
test1-a3-leaf2
>>>
```

Chaining multiple named arguments.

```
>>> devices = nb.dcim.devices.filter(role='leaf-switch', status=True)
>>> for device in devices:
...     print(device.name)
...
test1-leaf2
>>>
```

Passing a list as a named argument adds multiple filters of the same value.

```
>>> devices = nb.dcim.devices.filter(role=['leaf-switch', 'spine-switch'])
>>> for device in devices:
...     print(device.name)
...
test1-a3-spine1
test1-a3-spine2
test1-a3-leaf1
>>>
```

To have the ability to iterate over the results multiple times then encapsulate them in a list. This will cause the entire result set to be fetched from the server.

```
>>> devices = list(nb.dcim.devices.filter(role='leaf-switch'))
```

get (**args*, ***kwargs*)

Queries the DetailsView of a given endpoint.

Parameters

- **key** (*int, optional*) – id for the item to be retrieved.
- ****kwargs** (*str, optional*) – Accepts the same keyword args as filter(). Any search argument the endpoint accepts can be added as a keyword arg.

Returns A single *Record* object or None

Raises **ValueError** – if kwarg search return more than one value.

Examples

Referencing with a kwarg that only returns one value.

```
>>> nb.dcim.devices.get(name='test1-a3-tor1b')
test1-a3-tor1b
>>>
```

Referencing with an id.

```
>>> nb.dcim.devices.get(1)
test1-edge1
>>>
```

Using multiple named arguments. For example, retriving the location when the location name is not unique and used in multiple sites.

```
>>> nb.locations.get(site='site-1', name='Row 1')
Row 1
```

update(*objects*)

Bulk updates existing objects on an endpoint.

Allows for bulk updating of existing objects on an endpoint. Objects is a list whic contain either json/dicts or Record derived objects, which contain the updates to apply. If json/dicts are used, then the id of the object *must* be included

Parameters **objects** (*list*) – A list of dicts or Record.

Returns True if the update succeeded

Examples

Updating objects on the *devices* endpoint:

```
>>> devices = nb.dcim.devices.update([
...     {'id': 1, 'name': 'test'},
...     {'id': 2, 'name': 'test2'},
... ])
>>> devices
[test2, test]
>>>
```

Use bulk update by passing a list of Records:

```
>>> devices = list(nb.dcim.devices.filter())
>>> devices
[Device1, Device2, Device3]
>>> for d in devices:
...     d.name = d.name+'-test'
...
>>> nb.dcim.devices.update(devices)
[Device1-test, Device2-test, Device3-test]
>>>
```

class pynetbox.core.endpoint.**DetailEndpoint**(*parent_obj*, *name*, *custom_return=None*)

Enables read/write operations on detail endpoints.

Endpoints like `available-ips` that are detail routes off traditional endpoints are handled with this class.

create(*data=None*)

The write operation for a detail endpoint.

Creates objects on a detail endpoint in NetBox.

Parameters **data** (*dict/list,optional*) – A dictionary containing the key/value pair of the items you're creating on the parent object. Defaults to empty dict which will create a single item with default values.

Returns A *Record* object or list of *Record* objects created from data created in NetBox.

list(***kwargs*)

The view operation for a detail endpoint

Returns the response from NetBox for a detail endpoint.

Args ****kwargs** key/value pairs that get converted into url parameters when passed to the endpoint. E.g. `.list(method='get_facts')` would be converted to `.../?method=get_facts`.

Returns A *Record* object or list of *Record* objects created from data retrieved from NetBox.

CHAPTER 1

Response

class pynetbox.core.response.**Record**(*values*, *api*, *endpoint*)

Create Python objects from NetBox API responses.

Creates an object from a NetBox response passed as `values`. Nested dicts that represent other endpoints are also turned into `Record` objects. All fields are then assigned to the object's attributes. If a missing attr is requested (e.g. requesting a field that's only present on a full response on a `Record` made from a nested response) then pynetbox will make a request for the full object and return the requested value.

Examples

Default representation of the object is usually its name:

```
>>> x = nb.dcim.devices.get(1)
>>> x
test1-switch1
>>>
```

Querying a string field:

```
>>> x = nb.dcim.devices.get(1)
>>> x.serial
'ABC123'
>>>
```

Querying a field on a nested object:

```
>>> x = nb.dcim.devices.get(1)
>>> x.device_type.model
'QFX5100-24Q'
>>>
```

Casting the object as a dictionary:

```
>>> from pprint import pprint
>>> pprint(dict(x))
```

(continues on next page)

(continued from previous page)

```
{'asset_tag': None,
 'cluster': None,
 'comments': '',
 'config_context': {},
 'created': '2018-04-01',
 'custom_fields': {},
 'device_role': {'id': 1,
                 'name': 'Test Switch',
                 'slug': 'test-switch',
                 'url': 'http://localhost:8000/api/dcim/device-roles/1/'},
 'device_type': {...},
 'display_name': 'test1-switch1',
 'face': {'label': 'Rear', 'value': 1},
 'id': 1,
 'name': 'test1-switch1',
 'parent_device': None,
 'platform': {...},
 'position': 1,
 'primary_ip': {'address': '192.0.2.1/24',
                'family': 4,
                'id': 1,
                'url': 'http://localhost:8000/api/ipam/ip-addresses/1/'},
 'primary_ip4': {...},
 'primary_ip6': None,
 'rack': {'display_name': 'Test Rack',
          'id': 1,
          'name': 'Test Rack',
          'url': 'http://localhost:8000/api/dcim/racks/1/'},
 'serial': 'ABC123',
 'site': {'id': 1,
          'name': 'TEST',
          'slug': 'TEST',
          'url': 'http://localhost:8000/api/dcim/sites/1/'},
 'status': {'label': 'Active', 'value': 1},
 'tags': [],
 'tenant': None,
 'vc_position': None,
 'vc_priority': None,
 'virtual_chassis': None}
 >>>
```

Iterating over a `Record` object:

```
>>> for i in x:
...  print(i)
...
('id', 1)
('name', 'test1-switch1')
('display_name', 'test1-switch1')
>>>
```

delete()

Deletes an existing object.

Returns True if DELETE operation was successful.

Example

```
>>> x = nb.dcim.devices.get(name='test1-a3-tor1b')
>>> x.delete()
True
>>>
```

full_details()
Queries the hyperlinked endpoint if 'url' is defined.

This method will populate the attributes from the detail endpoint when it's called. Sets the class-level *has_details* attribute when it's called to prevent being called more than once.

Returns True

save()
Saves changes to an existing object.

Takes a diff between the objects current state and its state at init and sends them as a dictionary to Request.patch().

Returns True if PATCH request was successful.

Example

```
>>> x = nb.dcim.devices.get(name='test1-a3-tor1b')
>>> x.serial
''
>>> x.serial = '1234'
>>> x.save()
True
>>>
```

serialize(*nested=False*, *init=False*)
Serializes an object

Pulls all the attributes in an object and creates a dict that can be turned into the json that netbox is expecting.

If an attribute's value is a `Record` type it's replaced with the `id` field of that object.

Note: Using this to get a dictionary representation of the record is discouraged. It's probably better to cast to dict() instead. See Record docstring for example.

Returns dict.

update(*data*)
Update an object with a dictionary.

Accepts a dict and uses it to update the record and call save(). For nested and choice fields you'd pass an int the same as if you were modifying the attribute and calling save().

Parameters **data** (*dict*) – Dictionary containing the k/v to update the record object with.

Returns True if PATCH request was successful.

Example

```
>>> x = nb.dcim.devices.get(1)
>>> x.update({
...   "name": "test-switch2",
...   "serial": "ABC321",
```

(continues on next page)

(continued from previous page)

```
... })
True
```

updates()

Compiles changes for an existing object into a dict.

Takes a diff between the objects current state and its state at init and returns them as a dictionary, which will be empty if no changes.

Returns dict.

Example

```
>>> x = nb.dcim.devices.get(name='test1-a3-tor1b')
>>> x.serial
''
>>> x.serial = '1234'
>>> x.updates()
{'serial': '1234'}
>>>
```

class pynetbox.core.response.**RecordSet**(*endpoint*, *request*, ***kwargs*)

Iterator containing Record objects.

Returned by *Endpoint.all()* and *Endpoint.filter()* methods. Allows iteration of and actions to be taken on the results from the aforementioned methods. Contains *Record* objects.

Examples

To see how many results are in a query by calling `len()`:

```
>>> x = nb.dcim.devices.all()
>>> len(x)
123
>>>
```

Simple iteration of the results:

```
>>> devices = nb.dcim.devices.all()
>>> for device in devices:
...     print(device.name)
...
test1-leaf1
test1-leaf2
test1-leaf3
>>>
```

delete()

Bulk deletes objects in a RecordSet.

Allows for batch deletion of multiple objects in a RecordSet

Returns True if bulk DELETE operation was successful.

Examples

Deleting offline *devices* on site 1:

```
>>> netbox.dcim.devices.filter(site_id=1, status="offline").delete()
>>>
```

update(***kwargs*)

Updates kwargs onto all Records in the RecordSet and saves these.

Updates are only sent to the API if a value were changed, and only for the Records which were changed

Returns True if the update succeeded, None if no update were required

Example

```
>>> result = nb.dcim.devices.filter(site_id=1).update(status='active')
True
>>>
```

CHAPTER 2

Request

class pynetbox.core.query.**RequestError**(*req*)
Basic Request Exception

More detailed exception that returns the original requests object for inspection. Along with some attributes with specific details from the requests object. If return is json we decode and add it to the message.

Example

```
>>> try:
...   nb.dcim.devices.create(name="destined-for-failure")
... except pynetbox.RequestError as e:
...   print(e.error)
```

class pynetbox.core.query.**ContentError**(*req*)
Content Exception

If the API URL does not point to a valid NetBox API, the server may return a valid response code, but the content is not json. This exception is raised in those cases.

class pynetbox.core.query.**AllocationError**(*req*)
Allocation Exception

Used with available-ips/available-prefixes when there is no room for allocation and NetBox returns 409 Conflict.

CHAPTER 3

IPAM

class pynetbox.models.ipam.**Prefixes**(*values*, *api*, *endpoint*)

available_ips

Represents the `available-ips` detail endpoint.

Returns a DetailEndpoint object that is the interface for viewing and creating IP addresses inside a prefix.

Returns *DetailEndpoint*

Examples

```
>>> prefix = nb.ipam.prefixes.get(24)
>>> prefix.available_ips.list()
[10.0.0.1/24, 10.0.0.2/24, 10.0.0.3/24, 10.0.0.4/24, 10.0.0.5/24, ...]
```

To create a single IP:

```
>>> prefix = nb.ipam.prefixes.get(24)
>>> prefix.available_ips.create()
10.0.0.1/24
```

To create multiple IPs:

```
>>> prefix = nb.ipam.prefixes.get(24)
>>> create = prefix.available_ips.create([{} for i in range(2)])
>>> create
[10.0.0.2/24, 10.0.0.3/24]
```

available_prefixes

Represents the `available-prefixes` detail endpoint.

Returns a DetailEndpoint object that is the interface for viewing and creating prefixes inside a parent prefix.

Very similar to `available_ips()` , except that dict (or list of dicts) passed to `.create()` needs to have a `prefix_length` key/value specifed.

Returns *DetailEndpoint*

Examples

```
>>> prefix = nb.ipam.prefixes.get(3)
>>> prefix
10.0.0.0/16
>>> prefix.available_prefixes.list()
[10.0.1.0/24, 10.0.2.0/23, 10.0.4.0/22, 10.0.8.0/21, 10.0.16.0/20, 10.0.32.0/
↪19, 10.0.64.0/18, 10.0.128.0/17]
```

Creating a single child prefix:

```
>>> prefix = nb.ipam.prefixes.get(1)
>>> prefix
10.0.0.0/24
>>> new_prefix = prefix.available_prefixes.create(
...     {"prefix_length": 29}
... )
>>> new_prefix
10.0.0.16/29
```

class pynetbox.models.ipam.**VlanGroups**(*values*, *api*, *endpoint*)

available_vlans

Represents the `available-vlans` detail endpoint.

Returns a DetailEndpoint object that is the interface for viewing and creating VLANs inside a VLAN group.

Returns *DetailEndpoint*

Examples

```
>>> vlan_group = nb.ipam.vlan_groups.get(1)
>>> vlan_group.available_vlans.list()
[10, 11, 12]
```

To create a new VLAN:

```
>>> vlan_group.available_vlans.create({"name": "NewVLAN"})
NewVLAN (10)
```

CHAPTER 4

TL;DR

Instantiate the *Api*. Use the methods available on *Endpoint* to return *Record* objects.

CHAPTER 5

API

class pynetbox.core.api.**Api**(*url*, *token=None*, *threading=False*)

The API object is the point of entry to pynetbox.

After instantiating the Api() with the appropriate named arguments you can specify which app and endpoint you wish to interact with.

Valid attributes currently are:

- dcim
- ipam
- circuits
- tenancy
- extras
- virtualization
- users
- wireless

Calling any of these attributes will return *App* which exposes endpoints as attributes.

Additional Attributes:

- **http_session(requests.Session):** Override the default session with your own. This is used to control a number of HTTP behaviors such as SSL verification, custom headers, retires, and timeouts. See custom sessions for more info.

Parameters

- **url** (*str*) – The base URL to the instance of NetBox you wish to connect to.
- **token** (*str*) – Your NetBox token.
- **threading** (*bool, optional*) – Set to True to use threading in `.all()` and `.filter()` requests.

Raises **AttributeError** – If app doesn't exist.

Examples

```
>>> import pynetbox
>>> nb = pynetbox.api(
...     'http://localhost:8000',
...     token='d6f4e314a5b5fefd164995169f28ae32d987704f'
... )
>>> list(nb.dcim.devices.all())
[test1-leaf1, test1-leaf2, test1-leaf3]
```

create_token(*username*, *password*)

Creates an API token using a valid NetBox username and password. Saves the created token automatically in the API object.

Returns The token as a `Record` object.

Raises *RequestError* if the request is not successful.

Example

```
>>> import pynetbox
>>> nb = pynetbox.api("https://netbox-server")
>>> token = nb.create_token("admin", "netboxpassword")
>>> nb.token
'96d02e13e3f1fdcd8b4c089094c0191dcb045bef'
>>> from pprint import pprint
>>> pprint(dict(token))
{'created': '2021-11-27T11:26:49.360185+02:00',
 'description': '',
 'display': '045bef (admin)',
 'expires': None,
 'id': 2,
 'key': '96d02e13e3f1fdcd8b4c089094c0191dcb045bef',
 'url': 'https://netbox-server/api/users/tokens/2/',
 'user': {'display': 'admin',
          'id': 1,
          'url': 'https://netbox-server/api/users/users/1/',
          'username': 'admin'},
 'write_enabled': True}
>>>
```

openapi()

Returns the OpenAPI spec.

Quick helper function to pull down the entire OpenAPI spec.

Returns dict

Example

```
>>> import pynetbox
>>> nb = pynetbox.api(
...     'http://localhost:8000',
...     token='d6f4e314a5b5fefd164995169f28ae32d987704f'
... )
>>> nb.openapi()
{...}
>>>
```

status()

Gets the status information from NetBox.

Returns Dictionary as returned by NetBox.

Raises *RequestError* if the request is not successful.

Example

```
>>> pprint.pprint(nb.status())
{'django-version': '3.1.3',
 'installed-apps': {'cacheops': '5.0.1',
                    'debug_toolbar': '3.1.1',
                    'django_filters': '2.4.0',
                    'django_prometheus': '2.1.0',
                    'django_rq': '2.4.0',
                    'django_tables2': '2.3.3',
                    'drf_yasg': '1.20.0',
                    'mptt': '0.11.0',
                    'rest_framework': '3.12.2',
                    'taggit': '1.3.0',
                    'timezone_field': '4.0'},
 'netbox-version': '2.10.2',
 'plugins': {},
 'python-version': '3.7.3',
 'rq-workers-running': 1}
>>>
```

version

Gets the API version of NetBox.

Can be used to check the NetBox API version if there are version-dependent features or syntaxes in the API.

Returns Version number as a string.

Example

```
>>> import pynetbox
>>> nb = pynetbox.api(
...     'http://localhost:8000',
...     token='d6f4e314a5b5fefd164995169f28ae32d987704f'
... )
>>> nb.version
'3.1'
>>>
```

CHAPTER 6

App

class pynetbox.core.app.**App**(*api*, *name*)

Represents apps in NetBox.

Calls to attributes are returned as Endpoint objects.

Returns *Endpoint* matching requested attribute.

Raises *RequestError* if requested endpoint doesn't exist.

config()

Returns config response from app

Returns Raw response from NetBox's config endpoint.

Raises *RequestError* if called for an invalid endpoint.

Example

```
>>> pprint.pprint(nb.users.config())
{'tables': {'DeviceTable': {'columns': ['name',
                                        'status',
                                        'tenant',
                                        'device_role',
                                        'site',
                                        'primary_ip',
                                        'tags']}}}
```

CHAPTER 7

Indices and tables

- genindex

Index