src Documentation

Release 3.4.2

Author

July 31, 2018

Contents

Contents:

CHAPTER 1

About Globo NetworkAPI

Description

Globo NetworkAPI is a REST API that manages IP networking resources. It is supposed to be not just an IPAM, but a centralized point of network control, allowing documentation from physical and logical network and starting configuration requests to equipments.

Globo NetworkAPI is made to support a Web User Interface features, exposing its funcionality to be used with any other client.

This web tool helps network administrator manage and automate networking resources (routers, switches and load balancers) and document logical and physical networking.

They were created to be vendor agnostic and to support different orquestrators and environments without loosing the centralized view of all network resources allocated.

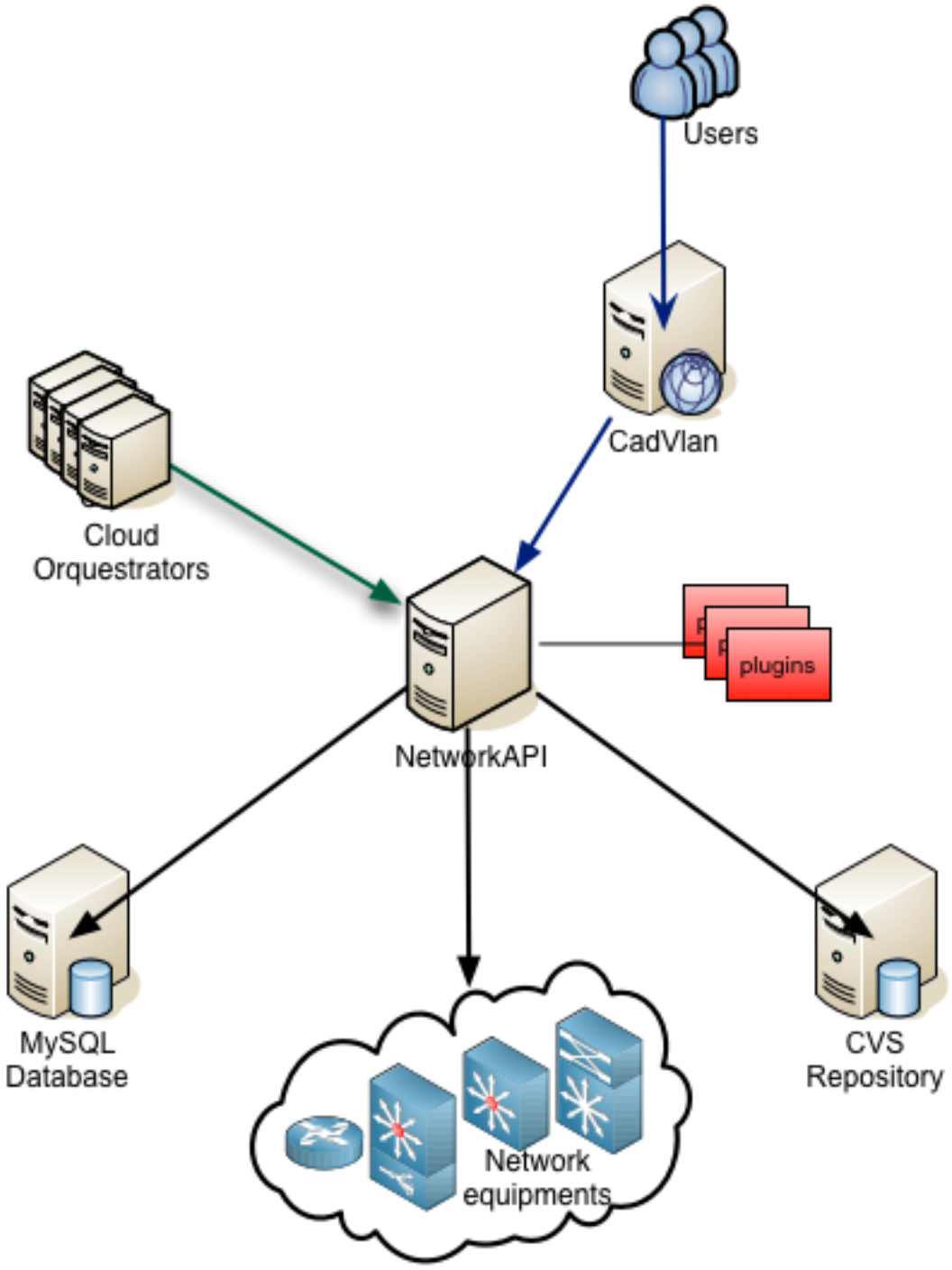
It was not created to be and inventory database, so it does not have CMDB functionalities.

You can find documentation for the Web UI in this link.

Features

- LDAP authentication
- Supports cabling documentation (including patch-panels/DIO's)
- Separated Layer 2 and Layer 3 documentation (vlan/network)
- IPv4 and IPv6 support
- Automatic allocation of Vlans, Networks and IP's
- ACL (access control list) automation (documentation/versioning/applying)
- Load-Balancer support
- Automated deploy of allocated resources on switches, routers and load balancers
- Load balancers management
- Expandable plugins for automating configuration

Architecture



Related Projects

Globo NetworkAPI WebUI.

Globo NetworkAPI Python Client.

Globo NetworkAPI Python Java.

CHAPTER 2

Pre-provisioned Server

The pre provisioned Globo NetworkAPI server uses Vagrant with a Hashicorp Ubuntu 32 bit server.

You can test it locally using this server. We don't recommend running this server in a production environment.

Root password for MySQL database is "password".

GloboNetworkAPI admin user password is "default".

By default, the server will use IP 10.0.0.2/24 in a local network. If this settings conflict with you network environment, you can modify it in the "Vagrantfile", in root directory.

Requirements

- [VirtualBox](https://www.virtualbox.org/wiki/Downloads)
- [Vagrant](http://www.vagrantup.com/downloads.html) with vagrant Omnibus plugin

`bash vagrant plugin install vagrant-omnibus) ` - [Git](http://git-scm.com/downloads)

Setting up the VM

Execute the following commands:

```
`bash $ git clone https://github.com/globocom/GloboNetworkAPI $ cd
GloboNetworkAPI $ vagrant up `
```

The GloboNetworkAPI will be available locally at http://10.0.0.2:8000

The gunicorn logs are at /tmp/gunicorn-*

The Django logs are at /tmp/networkapi.log

CHAPTER 3

Installing Globo NetworkAPI

Using pre-configured VM

In order to use the pre-configured VM you need to have *vagrant <https://www.vagrantup.com/downloads.html>* and *VirtualBox <https://www.virtualbox.org/wiki/Downloads>* installed in your machine.

After that, go to the directory you want to install and do:

```
git clone https://github.com/globocom/GloboNetworkAPI
cd GloboNetworkAPI
git submodule update --init --recursive
vagrant plugin install vagrant-omnibus
vagrant up
```

After this you'll have the GloboNetworkAPI running on http://10.0.0.2:8000/

Installing from scratch

Following examples were based on CentOS 7.0.1406 installation.

All root passwords were configured to "default".

All

Create a specific User/Group

```
useradd -m -U networkapi
passwd networkapi
visudo
        networkapi      ALL=(ALL)       ALL

sudo mkdir /opt/app/
sudo chmod 777 /opt/app/
```

Download Code

Download Globo NetworkAPI code from Globocom GitHub.

In this example we are downloading code to `/opt/app`:

```
sudo yum install git
cd /opt/app/
git clone https://github.com/globocom/GloboNetworkAPI
```

We are exporting this variable below to better document the install process:

```
export NETWORKAPI_FOLDER=/opt/app/GloboNetworkAPI/
echo "export NETWORKAPI_FOLDER=/opt/app/GloboNetworkAPI/" >> ~/.bashrc
```

Create a VirtualEnv

```
sudo yum install python-virtualenv
sudo easy_install pip
virtualenv ~/virtualenvs/networkapi_env
source ~/virtualenvs/networkapi_env/bin/activate
echo "source ~/virtualenvs/networkapi_env/bin/activate" >> ~/.bashrc
```

Install Dependencies

You will need the following packages in order to install the next python packages via `pip`:

```
sudo yum install mysql
sudo yum install mysql-devel
sudo yum install gcc
```

Install the packages listed on `$NETWORKAPI_FOLDER/requirements.txt` file:

```
pip install -r $NETWORKAPI_FOLDER/requirements.txt
```

Create a `sitecustomize.py` inside your `/path/to/lib/python2.X` folder with the following content:

```
import sys
sys.setdefaultencoding('utf-8')

echo -e "import sys\nsys.setdefaultencoding('utf-8')\n" > ~/virtualenvs/networkapi_env/lib/python2.7/
```

Install Memcached

You can run memcached locally or you can set file variable `CACHE_BACKEND` to use a remote memcached farm in file `$NETWORKAPI_FOLDER/networkapi/environment_settings.py`.

In case you need to run locally:

```
sudo yum install memcached
sudo systemctl start memcached
sudo systemctl enable memcached
```

MySQL Server Configuration

For details on MySQL installation, check MySQL Documentation.

```
sudo yum install mariadb-server mariadb
sudo systemctl start mariadb.service
sudo systemctl enable mariadb.service
sudo /usr/bin/mysql_secure_installation
```

Test installation and create a telecom database:

```
mysql -u root -p<password>
CREATE user 'telecom' IDENTIFIED BY '<password>';
GRANT ALL ON *.* TO 'telecom'@'%';
FLUSH PRIVILEGES;
```

Create the necessary tables:

```
mysql -u <user> -p <password> -h <host> <dbname> < $NETWORKAPI_FOLDER/dev/database_configuration.sql
```

If you want to load into your database the environment used for documentation examples:

```
mysql -u <user> -p <password> -h <host> <dbname> < $NETWORKAPI_FOLDER/dev/load_example_environment.sq
```

Configure the Globo NetworkAPI code to use your MySQL instance:

File `$NETWORKAPI_FOLDER/networkapi/environment_settings.py`:

```
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'your_db_name'
DATABASE_USER = 'your_db_user'
DATABASE_PASSWORD = 'your_db_password'
DATABASE_HOST = 'your_db_user_host'
DATABASE_PORT = '3306'
DATABASE_OPTIONS = {"init_command": "SET storage_engine=INNODB"}
```

HTTP Server Configuration

For a better performance, install Green Unicorn to run Globo NetworkAPI.

```
pip install gunicorn
```

There is no need to install a nginx or apache to proxy pass the requests, once there is no static files in the API.

Edit `$NETWORKAPI_FOLDER/gunicorn.conf.py` to use your log files location and user preferentes and run gunicorn:

```
cd $NETWORKAPI_FOLDER
gunicorn networkapi_wsgi:application
```

Test installation

Try to access the root location of the API:

```
http://your_location:8000/
```

This should take you a to 404 page listing available url's.

LDAP Server Configuration

If you want to use LDAP authentication, configure the following variables in `FILE`:

!TODO

Integrate with Queue

Install Dependencies:

```
Apache ActiveMQ
```

Apache ActiveMQ ™ is the most popular and powerful open source messaging and Integration Patterns server. Apache ActiveMQ Getting Started.

Example configuration on `settings.py`:

```
BROKER_DESTINATION = "/topic/queue_name"
BROKER_URI = "failover:(tcp://localhost:61613,tcp://server2:61613)?randomize=false"
```

Usage:

```
from queue_tools import queue_keys
from queue_tools.queue_manager import QueueManager

# Create new queue manager
queue_manager = QueueManager()

# Dict is the message body
obj_to_queue = {
 "id_vlan": <vlan_id>,
 "num_vlan": <num_vlan>,
 "id_environment": <environment_id>,
 "networks_ipv4": [
  {
   "id": <id>,
   "ip_formated": "<oct1>.<oct2>.<oct3>.<oct4>/<block>"
  }
 ],
 "networks_ipv6": [
  {
   "id": <id>,
   "ip_formated": "<oct1>.<oct2>.<oct3>.<oct4>.<oct5>.<oct6>.<oct7>.<oct8>/<block>"
  }
 ],
 "description": queue_keys.VLAN_REMOVE,
}

# Add in memory temporary on queue to sent
queue_manager.append(obj_to_queue)
```

```
# sent to consumer
queue_manager.send()
```

Output:

```
$VAR1 = {
 'id_vlan' => <id>,
 "num_vlan" => <num_vlan>,
 "id_environment" => <environment_id>,
 "networks_ipv4" => [
  {
   "id" => <id>,
   "ip_formated" => "<oct1>.<oct2>.<oct3>.<oct4>/<block>"
  }
 ],
 "networks_ipv6" => [
  {
   "id" => <id>,
   "ip_formated" => "<oct1>.<oct2>.<oct3>.<oct4>.<oct5>.<oct6>.<oct7>.<oct8>/<block>"
  }
 ],
 'description' => 'remove'
};
```

Features that use the `QueueManager.py`:

```
Vlan  remove()
uri: vlan/<id_vlan>/remove/

Vlan  create_ipv4()
uri: vlan/v4/create/

Vlan  create_ipv6()
uri: vlan/v6/create/

Vlan  create_acl()
uri: vlan/create/acl/

Vlan  create_script_acl()
uri: vlan/create/script/acl/

Vlan  create_vlan()
uri: vlan/create/

Vlan  criar()
uri: vlan/<id_vlan>/criar/
```

Working with Documentation

If you want to generate documentation, you need the following python modules installed:

```
pip install sphinx==1.2.2
pip install sphinx-rtd-theme==0.1.6
pip install pytest==2.2.4
```

Front End

If you want o have a Front End user application to use with Globo NetworkAPI you can install GloboNetworkAPI WebUI.

CHAPTER 4

Definitions

Contents

Access

Access is used to configure the protocol and username/password used for accessing *Equipment*. Plugins can use these informations in order to get credencials for listing and configuring *Equipment*.

You can insert many different protocols for each *Equipment*, like telnet, ssh, snmp etc.

Administrative Permission

Security functions that are used to allow administrative system functions or equipment configuration functions. These permissions are configured per *User Group*.

Notice that in order to do equipment configuration functions, the *User Group* has to have the respective administrative permission AND permissions on the *Equipment Group*.

Brand

Used for categorizing the equipments of a specific vender/brand name.

Environment

The environment defines a logical part of the infrastructure, usually a broadcast domain. It can be divided in 3 parts: "Divisao_DC", "Ambiente Logico" and "Grupo Layer3". It is expected to have all vlans in a environment routed in the same gateway/router.

I.e. In the picture below, you can see 5 different envinronments, represented by different colors:

- The Red has vlan range from 11 to 20 and *Equipment* R1, R2, R3 and SR1.
- The Blue has vlan range from 21 to 30 and *Equipment* B1, B2, B3 and SR1.
- The Green has vlan 31 and *Equipment* SR1 and Router.
- The Yellow has vlan 20 and *Equipment* SR2 and Router.
- The Orange has vlan ranges from 15 to 19 and from 21 to 40, and *Equipment* O1, O2 and O3 and SR2.

Notice that, as you have a common equipment SR1 in 3 environments Red, Green and Blue, you cannot have vlan numbers that overlaps in them. The same applies to Yellow and Orange environments.

As Orange and {Red, Green an Blue} have no equipments in common, they can have vlans that shares the same numbers.

This is automatically considered by Globo NetworkAPI when you configure your environments and their *Equipment*.

You can also have a server like S1 that needs to connect to more than environment. In this cases, you have to configure a *Filter* for those environments.

Equipment

Equipment represents any object in the infrasctructure that has to be documented. Routers, switches, patch panels, servers, load balancers, virtual servers etc. Equipments have a type, a "brand" and a "model" in order to categorize it. They can also be arranged in *Equipment Group*.

Equipments can have *IP (IPv4/IPv6)* and *Interface* and can be associated with an *Environment*.

In *example topology* above, server *S1* has IPs *172.16.0.5* and *10.0.0.5* and is part of 2 environments, Blue and Orange. Switch B1 does not have any IP address, but it is part of Blue environment. SR1 may have hundreds of IPs and it is part of 3 environments.

Equipment Group

Equipment Group is used for access restrictions on *Equipment*.

In order to be able to read/write configurations for an *Equipment*, a *User* has to be in a *User Group* that has the necessary permissions in at least one Equipment Group that the specific *Equipment* is part of.

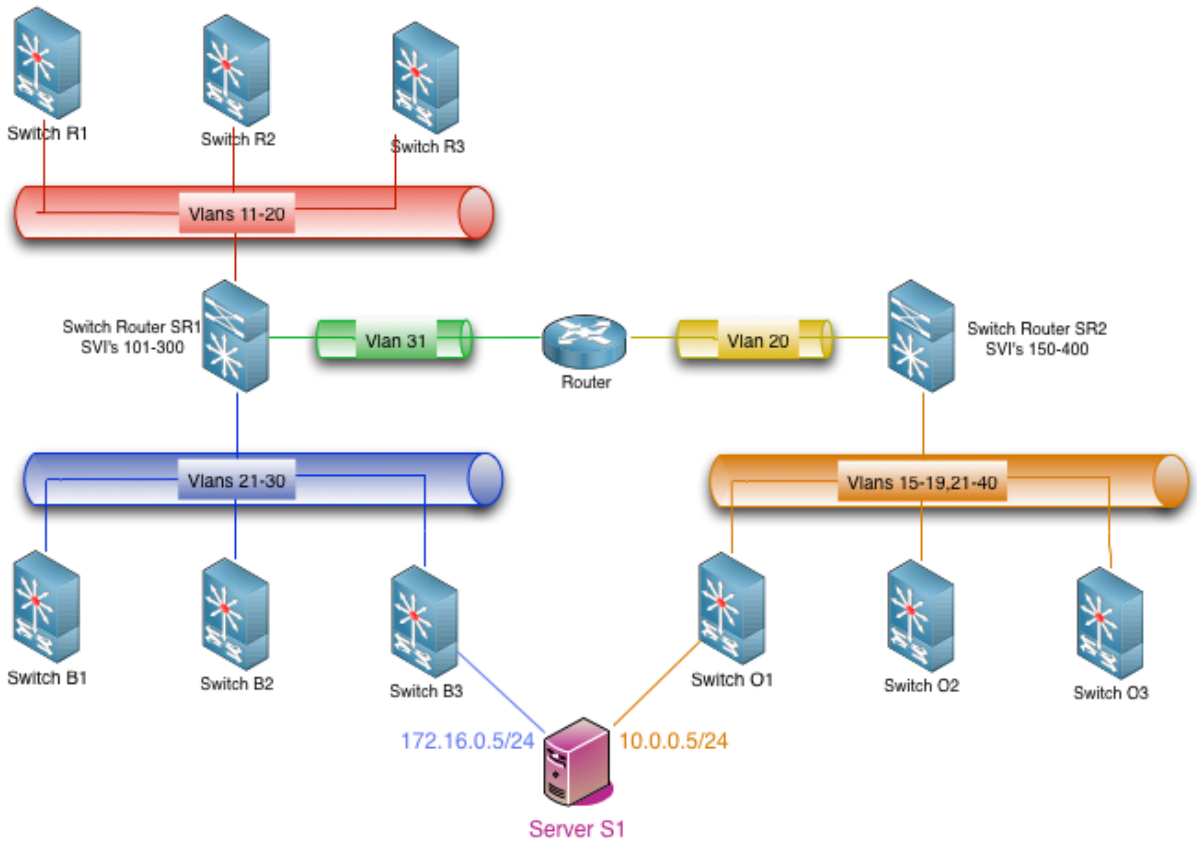


Figure 4.1: *example topology*

So, in order to be managed, an *Equipment* has to be part of at least one Equipment Group.

Equipment Type

This is field for categorizing the *Equipment*. It is also used in *Filter*.

Filter

Filters are used to permit an *Equipment* of a specific *Equipment Type* to be part of more than one *Environment* that has overlapping vlans and/or networks.

In the *example topology* above, server S1 should be part of two environments that have overlapping vlan numbers. In this case, there should have a filter that has the server S1 *Equipment Type* applied in both environments, Blue and Orange. It is recommended not to create filters with switches or routers.

IP (IPv4/IPv6)

Represents the IPs allocated for every *Equipment* in a specific *Network*. You can allocate the same IP to more than one *Equipment* in the same network (clusters, gateway redundancy protocols etc). There is no limit on the number of IPs an *Equipment* can have (if available in *Network*).

The allocated IPs are used for documentation pourposes and for automatic allocating of newly unused IPs.

Interface

Represents the physical interfaces that *Equipment* may have and the connections between them.

- You can represent patch panels in 2 ways. Generic patch panel or mirrored patch panel. When 2 patch panels are connected by their back connections in a organized way (the same interface numbers are correlated), you can represent them as a single *Equipment*, a mirrored patch panel. In other cases, you can represent each panel as a separated *Equipment*, like a generic patch panel.
- Patch panels has for each interface, 2 connections. The “Front” and “Back” connection. You can define each side as you want.
- Only patch panels have “Back” connections. All other equipments should have only “Front” connections.

Figure below show some examples of physical interfaces:

Figure 4.2: *example interfaces*

Figure 4.3: *example connections with a mirrored patch panel*

Figure 4.4: *example connections with generic patch panels*

Some equipments may have a front and back connection (i.e. patch panel) and some equipments only have 1 possible connection (ie network interface card on servers, switch interfaces etc).

Interfaces are used for documentation pourposes and to locate a switch port that a specific server is configured when you want to change that server inteface configuration on switch side. Interfaces that should not be configured in any case by the system should be configured with "protected" flag.

Model

Each *Brand* can have several models. The models are used for documenting purposes.

Network

Represent the layer 3, IPv4 or IPv6 address range.

As it is a different layer, althought not recommended by IP networking best practices, you can have multiple IPv4/IPv6 networks in the same *Vlan*.

As *Vlan* and *Environment*, you cannot have two overlapping networks (same length or subnets/supernets) in the same environment. The equipments should not be able to treat that, but you can have those in different environments. I.E. in picture *example topology* you cannot have overlapping networks 172.16.0.0/24 and 172.16.0.128/25 in Blue, Red or Green environment at the same time, because the same switch router supports all Layer3, but you can have 172.16.0.0/24 on Blue and 172.16.0.128/25 on Orange environment for example.

Network Type

Used for documentating purposes only. You can for example tell that *Network* of a specific type are used for point-to-point links, or for internal usage only, networks for NAT only etc.

Plugin (Roteiro)

The files used by the *Scripts* for performing a task on specific *Equipment*. Each plugin have a type to be categorized. You can write your scripts and how they look for your plugins, but is recommended that the plugin name be the file name used.

The plugin type is used for calling the correct *Scripts* in order to do that type of configuration.

Scripts

You can create scripts for doing anything in your environment. We recommend making them for a generic feature, and call a *Plugin (Roteiro)* from them to make equipment/brand specific syntax commands.

You associate a script with a specific *Plugin (Roteiro)* type and associate the *Plugin (Roteiro)* of that type to the *Equipment*. This way you can perform several tasks on different *Equipment* brands/models with the same plugin.

Template (ACL)

When you configure an *Environment*, you can define a base model for the access lists (ACL) for every interface vlan (SVI) that you create there. This model is the template. It is a text file with keywords that are replaced with the network/Ips created for those networks.

You can define a template for IPv4 and another for IPv6 *Network* for each *Environment*.

User

An account for a client to authenticate. You can use a locally stored password or configure to use LDAP authentication.

User Group

User groups used for access restriction. All permissions are based on user groups. There is no way to give a permission directly to a *User*.

Vlan

Represent the layer 2 Vlans on equipments. See *Environment* for restrictions on vlan numbering.

CHAPTER 5

FAQ

Is empty now.

CHAPTER 6

Globo NetworkAPI API Docs

networkapi package

Subpackages

networkapi.acl package

Submodules

networkapi.acl.Enum module

networkapi.acl.acl module

networkapi.acl.file module

Module contents

networkapi.ambiente package

Subpackages

networkapi.ambiente.resource package

Submodules

networkapi.ambiente.resource.AmbienteResource module

networkapi.ambiente.resource.DivisionDcAddResource module

networkapi.ambiente.resource.DivisionDcAlterRemoveResource module

networkapi.ambiente.resource.DivisionDcGetAllResource module

networkapi.ambiente.resource.EnvironmentBlocks module

networkapi.ambiente.resource.EnvironmentConfigurationAddResource module

networkapi.ambiente.resource.EnvironmentConfigurationListResource module

networkapi.ambiente.resource.EnvironmentConfigurationRemoveResource module

networkapi.ambiente.resource.EnvironmentGetAclPathsResource module

networkapi.ambiente.resource.EnvironmentGetByEquipResource module

networkapi.ambiente.resource.EnvironmentGetByIdResource module

networkapi.ambiente.resource.EnvironmentIpConfigResource module

networkapi.ambiente.resource.EnvironmentListResource module

networkapi.ambiente.resource.EnvironmentSetTemplateResource module

networkapi.ambiente.resource.EnvironmentVipGetAmbienteP44TxtResource module

networkapi.ambiente.resource.EnvironmentVipGetClienteTxtResource module

networkapi.ambiente.resource.EnvironmentVipGetFinalityResource module

networkapi.ambiente.resource.EnvironmentVipResource module

networkapi.ambiente.resource.EnvironmentVipSearchResource module

networkapi.ambiente.resource.GroupL3AddResource module

networkapi.ambiente.resource.GroupL3AlterRemoveResource module

networkapi.ambiente.resource.GroupL3GetAllResource module

networkapi.ambiente.resource.LogicalEnvironmentAddResource module

networkapi.ambiente.resource.LogicalEnvironmentAlterRemoveResource module

networkapi.ambiente.resource.LogicalEnvironmentGetAllResource module

networkapi.ambiente.resource.RequestAllVipsEnviromentVipResource module

Module contents

networkapi.ambiente.response package

Module contents

networkapi.ambiente.test package

Submodules

networkapi.ambiente.test.test_DivisionDc module

networkapi.ambiente.test.test_Environment module

networkapi.ambiente.test.test_EnvironmentVIP module

networkapi.ambiente.test.test_GroupL3 module

networkapi.ambiente.test.test_LogicalEnvironment module

Module contents

Submodules

networkapi.ambiente.models module

Module contents

networkapi.auth package

Module contents

networkapi.blockrules package

Subpackages

networkapi.blockrules.resource package

Submodules

networkapi.blockrules.resource.RuleGetResource module

networkapi.blockrules.resource.RuleResource module

Module contents

networkapi.blockrules.test package

Submodules

networkapi.blockrules.test.test_Block module

networkapi.blockrules.test.test_Rule module

Module contents

Submodules

networkapi.blockrules.models module

Module contents

networkapi.check package

Submodules

networkapi.check.CheckAction module

class networkapi.check.CheckAction.**CheckAction**
Bases: object

check(*request*)

Module contents

networkapi.config package

Submodules

networkapi.config.models module

Module contents

networkapi.distributedlock package

Submodules

networkapi.distributedlock.memcachedlock module

Module contents

networkapi.equipamento package

Subpackages

networkapi.equipamento.resource package

Submodules

networkapi.equipamento.resource.BrandAddResource module

networkapi.equipamento.resource.BrandAlterRemoveResource module

networkapi.equipamento.resource.BrandGetAllResource module

networkapi.equipamento.resource.EquipAccessEditResource module

networkapi.equipamento.resource.EquipAccessGetResource module

networkapi.equipamento.resource.EquipAccessListResource module

networkapi.equipamento.resource.EquipScriptListResource module

networkapi.equipamento.resource.EquipamentoAcessoResource module

networkapi.equipamento.resource.EquipamentoEditResource module

networkapi.equipamento.resource.EquipamentoGrupoResource module

networkapi.equipamento.resource.EquipamentoResource module

networkapi.equipamento.resource.EquipmentEnvironmentDeallocateResource module

networkapi.equipamento.resource.EquipmentFindResource module

networkapi.equipamento.resource.EquipmentGetAllResource module

networkapi.equipamento.resource.EquipmentGetByGroupEquipmentResource module

networkapi.equipamento.resource.EquipmentGetRealRelated module

networkapi.equipamento.resource.EquipmentListResource module

networkapi.equipamento.resource.EquipmentScriptAddResource module

networkapi.equipamento.resource.EquipmentScriptGetAllResource module

networkapi.equipamento.resource.EquipmentScriptRemoveResource module

networkapi.equipamento.resource.EquipmentTypeAddResource module

networkapi.equipamento.resource.EquipmentTypeGetAllResource module

networkapi.equipamento.resource.ModelAddResource module

networkapi.equipamento.resource.ModelAlterRemoveResource module

networkapi.equipamento.resource.ModelGetAllResource module

networkapi.equipamento.resource.ModelGetByBrandResource module

Module contents

networkapi.equipamento.response package

Module contents

networkapi.equipamento.test package

Submodules

networkapi.equipamento.test.test_Brand module

networkapi.equipamento.test.test_Equipment module

networkapi.equipamento.test.test_EquipmentAccess module

networkapi.equipamento.test.test_EquipmentEnvironment module

networkapi.equipamento.test.test_EquipmentScript module

networkapi.equipamento.test.test_EquipmentType module

networkapi.equipamento.test.test_Model module

Module contents

Submodules

networkapi.equipamento.models module

Module contents

networkapi.eventlog package

Subpackages

networkapi.eventlog.resource package

Submodules

networkapi.eventlog.resource.EventLogChoiceResource module

networkapi.eventlog.resource.EventLogFindResource module

Module contents

Submodules

networkapi.eventlog.models module

Module contents

networkapi.filter package

Subpackages

networkapi.filter.resource package

Submodules

networkapi.filter.resource.FilterAddResource module

networkapi.filter.resource.FilterAlterRemoveResource module

networkapi.filter.resource.FilterAssociateResource module

networkapi.filter.resource.FilterDissociateOneResource module

networkapi.filter.resource.FilterGetByIdResource module

networkapi.filter.resource.FilterListAllResource module

Module contents

networkapi.filter.test package

Submodules

networkapi.filter.test.test_Filter module

Module contents

Submodules

networkapi.filter.models module

Module contents

networkapi.filterequiptype package

Submodules

networkapi.filterequiptype.models module

Module contents

networkapi.grupo package

Subpackages

networkapi.grupo.resource package

Submodules

networkapi.grupo.resource.AdministrativePermissionAddResource module

networkapi.grupo.resource.AdministrativePermissionAlterRemoveResource module

networkapi.grupo.resource.AdministrativePermissionByGroupUserResource module

networkapi.grupo.resource.AdministrativePermissionGetAllResource module

networkapi.grupo.resource.AdministrativePermissionGetByIdResource module

networkapi.grupo.resource.GroupEquipmentResource module

networkapi.grupo.resource.GroupUserAddResource module

networkapi.grupo.resource.GroupUserAlterRemoveResource module

networkapi.grupo.resource.GroupUserGetAllResource module

networkapi.grupo.resource.GroupUserGetByIdResource module

networkapi.grupo.resource.GrupoEquipamentoAssociaEquipamentoResource module

networkapi.grupo.resource.GrupoEquipamentoGetByEquipResource module

networkapi.grupo.resource.GrupoEquipamentoRemoveAssociationEquipResource module

networkapi.grupo.resource.GrupoResource module

networkapi.grupo.resource.PermissionGetAllResource module

Module contents

networkapi.grupo.test package

Submodules

networkapi.grupo.test.test_EquipmentGroup module

networkapi.grupo.test.test_EquipmentGroupRights module

networkapi.grupo.test.test_GroupUser module

networkapi.grupo.test.test_Permission module

networkapi.grupo.test.test_PermissionAdministrative module

Module contents

Submodules

networkapi.grupo.models module

Module contents

networkapi.grupovirtual package

Subpackages

networkapi.grupovirtual.resource package

Submodules

networkapi.grupovirtual.resource.GrupoVirtualResource module

Module contents

Module contents

networkapi.healthcheckexpect package

Subpackages

networkapi.healthcheckexpect.resource package

Submodules

networkapi.healthcheckexpect.resource.HealthcheckAddExpectStringResource module

networkapi.healthcheckexpect.resource.HealthcheckAddResource module

networkapi.healthcheckexpect.resource.HealthcheckExpectDistinctResource module

networkapi.healthcheckexpect.resource.HealthcheckExpectGetResource module

networkapi.healthcheckexpect.resource.HealthcheckExpectResource module

Module contents

networkapi.healthcheckexpect.test package

Submodules

networkapi.healthcheckexpect.test.test_HealthcheckExpect module

Module contents

Submodules

networkapi.healthcheckexpect.models module

Module contents

networkapi.infrastructure package

Submodules

networkapi.infrastructure.datatable module

networkapi.infrastructure.ip_subnet_utils module

networkapi.infrastructure.ip_subnet_utils.**get_prefix_IPV4**(*num_hosts*)

networkapi.infrastructure.ip_subnet_utils.**get_prefix_IPV6**(*num_hosts*)

networkapi.infrastructure.ip_subnet_utils.**is_subnetwork**(*network_address_01*, *network_address_02*)

Verifica se o endereço network_address_01 é sub-rede do endereço network_address_02.

@param network_address_01: Uma tuple com os octetos do endereço, formato: (oct1, oct2, oct3, oct5) @param network_address_02: Uma tuple com os octetos do endereço e o bloco, formato: (oct1, oct2, oct3, oct5, bloco)

@return: True se network_address_01 é sub-rede de network_address_02. False caso contrário.

networkapi.infrastructure.ip_subnet_utils.**is_valid_ip**(*address*)

Verifica se address é um endereço ip válido.

networkapi.infrastructure.ip_subnet_utils.**network_mask_from_cidr_mask**(*cidr_mask*)

Calcula a máscara de uma rede a partir do número do bloco do endereço.

@param cidr_mask: Valor do bloco do endereço.

@return: Tuple com o octeto 1, 2, 3, 4 da máscara: (oct1,oct2,oct3,oct4).

networkapi.infrastructure.ipaddr module

A fast, lightweight IPv4/IPv6 manipulation library in Python.

This library is used to create/poke/manipulate IPv4 and IPv6 addresses and networks.

exception networkapi.infrastructure.ipaddr.**AddressValueError**

Bases: exceptions.ValueError

A Value Error related to the address.

networkapi.infrastructure.ipaddr.**CollapseAddrList**(*addresses*)

Collapse a list of IP objects.

Example:

collapse_address_list([IPv4('1.1.0.0/24'), IPv4('1.1.1.0/24')]) -> [IPv4('1.1.0.0/23')]

Args: addresses: A list of IPv4Network or IPv6Network objects.

Returns: A list of IPv4Network or IPv6Network objects depending on what we were passed.

Raises: TypeError: If passed a list of mixed version objects.

networkapi.infrastructure.ipaddr.**IPAddress**(*address*, *version=None*)
Take an IP string/int and return an object of the correct type.

Args:

address: A string or integer, the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default.

version: An Integer, 4 or 6. If set, don't try to automatically determine what the IP address type is. important for things like IPAddress(1), which could be IPv4, '0.0.0.1', or IPv6, '::1'.

Returns: An IPv4Address or IPv6Address object.

Raises:

ValueError: if the string passed isn't either a v4 or a v6 address.

networkapi.infrastructure.ipaddr.**IPNetwork**(*address*, *version=None*, *strict=False*)
Take an IP string/int and return an object of the correct type.

Args:

address: A string or integer, the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than 2**32 will be considered to be IPv4 by default.

version: An Integer, if set, don't try to automatically determine what the IP address type is. important for things like IPNetwork(1), which could be IPv4, '0.0.0.1/32', or IPv6, '::1/128'.

Returns: An IPv4Network or IPv6Network object.

Raises:

ValueError: if the string passed isn't either a v4 or a v6 address. Or if a strict network was requested and a strict network wasn't given.

class networkapi.infrastructure.ipaddr.**IPv4Address**(*address*)
Bases: networkapi.infrastructure.ipaddr._BaseV4, networkapi.infrastructure.ipaddr._BaseIP

Represent and manipulate single IPv4 Addresses.

class networkapi.infrastructure.ipaddr.**IPv4Network**(*address*, *strict=False*)
Bases: networkapi.infrastructure.ipaddr._BaseV4, networkapi.infrastructure.ipaddr._BaseNet

This class represents and manipulates 32-bit IPv4 networks.

Attributes: [examples for IPv4Network('1.2.3.4/27')] ._ip: 16909060 .ip: IPv4Address('1.2.3.4') .network: IPv4Address('1.2.3.0') .hostmask: IPv4Address('0.0.0.31') .broadcast: IPv4Address('1.2.3.31') .netmask: IPv4Address('255.255.255.224') .prefixlen: 27

IsLinkLocal()

IsLoopback()

IsMulticast()

IsRFC1918()

class networkapi.infrastructure.ipaddr.**IPv6Address**(*address*)
Bases: networkapi.infrastructure.ipaddr._BaseV6, networkapi.infrastructure.ipaddr._BaseIP

Represent and manipulate single IPv6 Addresses.

class networkapi.infrastructure.ipaddr.**IPv6Network**(*address*, *strict=False*)
Bases: networkapi.infrastructure.ipaddr._BaseV6, networkapi.infrastructure.ipaddr._BaseNet

This class represents and manipulates 128-bit IPv6 networks.

Attributes: [examples for IPv6('2001:658:22A:CAFE:200::1/64')] .ip: IPv6Address('2001:658:22a:cafe:200::1') .network: IPv6Address('2001:658:22a:cafe::') .hostmask: IPv6Address('::ffff:ffff:ffff:ffff') .broadcast: IPv6Address('2001:658:22a:cafe:ffff:ffff:ffff:ffff') .netmask: IPv6Address('ffff:ffff:ffff:ffff::') .prefixlen: 64

with_netmask

exception networkapi.infrastructure.ipaddr.**NetmaskValueError**
Bases: exceptions.ValueError

A Value Error related to the netmask.

networkapi.infrastructure.ipaddr.**collapse_address_list**(*addresses*)
Collapse a list of IP objects.

Example:

collapse_address_list([IPv4('1.1.0.0/24'), IPv4('1.1.1.0/24')]) -> [IPv4('1.1.0.0/23')]

Args: addresses: A list of IPv4Network or IPv6Network objects.

Returns: A list of IPv4Network or IPv6Network objects depending on what we were passed.

Raises: TypeError: If passed a list of mixed version objects.

networkapi.infrastructure.ipaddr.**get_mixed_type_key**(*obj*)
Return a key suitable for sorting between networks and addresses.

Address and Network objects are not sortable by default; they're fundamentally different so the expression

IPv4Address('1.1.1.1') <= IPv4Network('1.1.1.1/24')

doesn't make any sense. There are some times however, where you may wish to have ipaddr sort these for you anyway. If you need to do this, you can use this function as the key= argument to sorted().

Args: obj: either a Network or Address object.

Returns: appropriate key.

networkapi.infrastructure.ipaddr.**summarize_address_range**(*first*, *last*)
Summarize a network range given the first and last IP addresses.

Example:

```
>>> summarize_address_range(IPv4Address('1.1.1.0'),
    IPv4Address('1.1.1.130'))
[IPv4Network('1.1.1.0/25'), IPv4Network('1.1.1.128/31'),
IPv4Network('1.1.1.130/32')]
```

Args: first: the first IPv4Address or IPv6Address in the range. last: the last IPv4Address or IPv6Address in the range.

Returns: The address range collapsed to a list of IPv4Network's or IPv6Network's.

Raise:

TypeError: If the first and last objects are not IP addresses. If the first and last objects are not the same version.

ValueError: If the last object is not greater than the first. If the version is not 4 or 6.

networkapi.infrastructure.ipaddr.**v4_int_to_packed**(*address*)
The binary representation of this address.

Args: address: An integer representation of an IPv4 IP address.

Returns: The binary representation of this address.

Raises:

ValueError: If the integer is too large to be an IPv4 IP address.

networkapi.infrastructure.ipaddr.**v6_int_to_packed**(*address*)
The binary representation of this address.

Args: address: An integer representation of an IPv4 IP address.

Returns: The binary representation of this address.

networkapi.infrastructure.script_utils module

exception networkapi.infrastructure.script_utils.**ScriptError**(*cause*, *message*)
Bases: exceptions.Exception

Representa um erro ocorrido durante a chamada do script.

networkapi.infrastructure.script_utils.**exec_script**(*command*)

networkapi.infrastructure.xml_utils module

exception networkapi.infrastructure.xml_utils.**InvalidNodeNameXMLError**(*cause*, *message*)
Bases: networkapi.infrastructure.xml_utils.XMLError

Nome inválido para representá-lo como uma TAG de XML.

exception networkapi.infrastructure.xml_utils.**InvalidNodeTypeXMLError**(*cause*, *message*)
Bases: networkapi.infrastructure.xml_utils.XMLError

Tipo inválido para o conteúdo de uma TAG de XML.

exception networkapi.infrastructure.xml_utils.**XMLError**(*cause*, *message*)
Bases: exceptions.Exception

Representa um erro ocorrido durante o marshall ou unmarshall do XML.

networkapi.infrastructure.xml_utils.**dumps**(*map*, *root_name*, *root_attributes=None*)
Cria um string no formato XML a partir dos elementos do map.

Os elementos do mapa serão nós filhos do root_name.

Cada chave do map será um Nó no XML. E o valor da chave será o conteúdo do Nó.

Throws: XMLError, InvalidNodeNameXMLError, InvalidNodeTypeXMLError

networkapi.infrastructure.xml_utils.**dumps_networkapi**(*map*, *version='1.0'*)

networkapi.infrastructure.xml_utils.**loads**(*xml*, *force_list=None*)

Cria um dict com os dados do element root.

O dict terá como chave o nome do element root e como valor o conteúdo do element root. Quando o conteúdo de um element é uma lista de Nós então o valor do element será um dict com uma chave para cada nó. Entretanto, se existir nós, de um mesmo pai, com o mesmo nome, então eles serão armazenados uma mesma chave do dict que terá como valor uma lista.

Se o element root tem atributo, então também retorna um dict com os atributos.

Throws: XMLError

Module contents

networkapi.interface package

Subpackages

networkapi.interface.resource package

Submodules

networkapi.interface.resource.InterfaceDisconnectResource module

networkapi.interface.resource.InterfaceGetResource module

networkapi.interface.resource.InterfaceResource module

Module contents

networkapi.interface.test package

Submodules

networkapi.interface.test.test_Interface module

Module contents

Submodules

networkapi.interface.models module

Module contents

networkapi.ip package

Subpackages

networkapi.ip.resource package

Submodules

networkapi.ip.resource.IPEquipEvipResource module

networkapi.ip.resource.IPGetByEquipResource module

networkapi.ip.resource.IPv4AddResource module

networkapi.ip.resource.IPv4DeleteResource module

networkapi.ip.resource.IPv4EditResource module

networkapi.ip.resource.IPv4GetAvailableResource module

networkapi.ip.resource.IPv4GetResource module

networkapi.ip.resource.IPv4ListResource module

networkapi.ip.resource.IPv4SaveResource module

networkapi.ip.resource.IPv6AddResource module

networkapi.ip.resource.IPv6DeleteResource module

networkapi.ip.resource.IPv6EditResource module

networkapi.ip.resource.IPv6GetAvailableResource module

networkapi.ip.resource.IPv6GetResource module

networkapi.ip.resource.IPv6ListResource module

networkapi.ip.resource.IPv6SaveResource module

networkapi.ip.resource.IpCheckForVipResource module

networkapi.ip.resource.IpGetOctBlockResource module

networkapi.ip.resource.IpResource module

networkapi.ip.resource.Ipv4AssocEquipResource module

networkapi.ip.resource.Ipv4GetAvailableForVipResource module

networkapi.ip.resource.Ipv4GetByIdResource module

networkapi.ip.resource.Ipv6AssocEquipResource module

networkapi.ip.resource.Ipv6AssociateResource module

networkapi.ip.resource.Ipv6GetAvailableForVipResource module

networkapi.ip.resource.Ipv6GetByIdResource module

networkapi.ip.resource.Ipv6RemoveResource module

networkapi.ip.resource.NetworkAddResource module

networkapi.ip.resource.NetworkEditResource module

networkapi.ip.resource.NetworkIPv4AddResource module

networkapi.ip.resource.NetworkIPv4DeallocateResource module

networkapi.ip.resource.NetworkIPv4GetResource module

networkapi.ip.resource.NetworkIPv6AddResource module

networkapi.ip.resource.NetworkIPv6DeallocateResource module

networkapi.ip.resource.NetworkIPv6GetResource module

networkapi.ip.resource.NetworkRemoveResource module

networkapi.ip.resource.SearchIPv6EnvironmentResource module

Module contents

networkapi.ip.test package

Submodules

networkapi.ip.test.test_Ip module

networkapi.ip.test.test_Network module

Module contents

Submodules

networkapi.ip.ipcalc module

class networkapi.ip.ipcalc.**IP**(*ip*, *mask=None*, *version=0*)

Bases: object

Represents a single IP address.

```
>>> localhost = IP("127.0.0.1")
>>> print localhost
127.0.0.1
>>> localhost6 = IP("::1")
>>> print localhost6
0000:0000:0000:0000:0000:0000:0000:0001
```

bin()

Full-length binary representation of the IP address.

```
>>> ip = IP("127.0.0.1")
>>> print ip.bin()
01111111000000000000000000000001
```

clone()

Return a new <IP> object with a copy of this one.

```
>>> ip = IP('127.0.0.1')
>>> ip.clone()
<ipcalc.IP object at 0xb7d4d18c>
```

hex()

Full-length hexadecimal representation of the IP address.

```
>>> ip = IP("127.0.0.1")
>>> print ip.hex()
7f000001
```

info()

Show IANA allocation information for the current IP address.

```
>>> ip = IP("127.0.0.1")
>>> print ip.info()
CLASS A
```

size()

subnet()

to_ipv4()
Convert (an IPv6) IP address to an IPv4 address, if possible. Only works for IPv4-compat (::/96) and 6-to-4 (2002::/16) addresses.

```
>>> ip = IP('2002:c000:022a::')
>>> print ip.to_ipv4()
192.0.2.42
```

to_ipv6(*type='6-to-4'*)
Convert (an IPv4) IP address to an IPv6 address.

```
>>> ip = IP('192.0.2.42')
>>> print ip.to_ipv6()
2002:c000:022a:0000:0000:0000:0000:0000
```

to_tuple()
Used for comparisons.

version()
IP version.

```
>>> ip = IP("127.0.0.1")
>>> print ip.version()
4
```

class networkapi.ip.ipcalc.**Network**(*ip*, *mask=None*, *version=0*)
Bases: networkapi.ip.ipcalc.IP

Network slice calculations.

```
>>> localnet = Network('127.0.0.1/8')
>>> print localnet
127.0.0.1
```

broadcast()
Broadcast address.

```
>>> localnet = Network('127.0.0.1/8')
>>> print localnet.broadcast()
127.255.255.255
```

has_key(*ip*)
Check if the given ip is part of the network.

```
>>> net = Network('192.0.2.0/24')
>>> net.has_key('192.168.2.0')
False
>>> net.has_key('192.0.2.42')
True
```

host_first()
First available host in this subnet.

host_last()
Last available host in this subnet.

in_network(*other*)
Check if the given IP address is within this network.

netmask()
Network netmask derived from subnet size.

```
>>> localnet = Network('127.0.0.1/8')
>>> print localnet.netmask()
255.0.0.0
```

network()
Network address.

```
>>> localnet = Network('127.128.99.3/8')
>>> print localnet.network()
127.0.0.0
```

size()
Number of ip's within the network.

```
>>> net = Network('192.0.2.0/24')
>>> print net.size()
256
```

networkapi.ip.models module

Module contents

networkapi.models package

Submodules

networkapi.models.BaseManager module

networkapi.models.BaseModel module

networkapi.models.models_signal_receiver module

Module contents

networkapi.requisicaovips package

Subpackages

networkapi.requisicaovips.resource package

Submodules

networkapi.requisicaovips.resource.CreateVipResource module

networkapi.requisicaovips.resource.OptionVipAllGetByEnvironmentVipResource module

networkapi.requisicaovips.resource.OptionVipAllResource module

networkapi.requisicaovips.resource.OptionVipEnvironmentVipAssociationResource module

networkapi.requisicaovips.resource.OptionVipGetBalanceamentoByEVipResource module

networkapi.requisicaovips.resource.OptionVipGetGrupoCacheByEVipResource module

networkapi.requisicaovips.resource.OptionVipGetHealthcheckByEVipResource module

networkapi.requisicaovips.resource.OptionVipGetPersistenciaByEVipResource module

networkapi.requisicaovips.resource.OptionVipGetTimeoutByEVipResource module

networkapi.requisicaovips.resource.OptionVipResource module

networkapi.requisicaovips.resource.RemoveVipResource module

networkapi.requisicaovips.resource.RequestAllVipsIPv4Resource module

networkapi.requisicaovips.resource.RequestAllVipsIPv6Resource module

networkapi.requisicaovips.resource.RequestAllVipsResource module

networkapi.requisicaovips.resource.RequestHealthcheckResource module

networkapi.requisicaovips.resource.RequestMaxconResource module

networkapi.requisicaovips.resource.RequestPriorityResource module

networkapi.requisicaovips.resource.RequestVipGetByIdResource module

networkapi.requisicaovips.resource.RequestVipGetIdIpResource module

networkapi.requisicaovips.resource.RequestVipGetRulesByEVipResource module

networkapi.requisicaovips.resource.RequestVipL7ApplyResource module

networkapi.requisicaovips.resource.RequestVipL7Resource module

networkapi.requisicaovips.resource.RequestVipL7RollbackResource module

networkapi.requisicaovips.resource.RequestVipL7ValidateResource module

networkapi.requisicaovips.resource.RequestVipRealEditResource module

networkapi.requisicaovips.resource.RequestVipRealValidResource module

networkapi.requisicaovips.resource.RequestVipRuleResource module

networkapi.requisicaovips.resource.RequestVipValidateResource module

networkapi.requisicaovips.resource.RequestVipsRealResource module

networkapi.requisicaovips.resource.RequestVipsResource module

networkapi.requisicaovips.resource.RequisicaoVipDeleteResource module

networkapi.requisicaovips.resource.RequisicaoVipsResource module

Module contents

networkapi.requisicaovips.test package

Submodules

networkapi.requisicaovips.test.test_OptionVIP module

networkapi.requisicaovips.test.test_VipRequest module

Module contents

Submodules

networkapi.requisicaovips.models module

Module contents

networkapi.roteiro package

Subpackages

networkapi.roteiro.resource package

Submodules

networkapi.roteiro.resource.RoteiroResource module

networkapi.roteiro.resource.ScriptAddResource module

networkapi.roteiro.resource.ScriptAlterRemoveResource module

networkapi.roteiro.resource.ScriptGetAllResource module

networkapi.roteiro.resource.ScriptGetEquipmentResource module

networkapi.roteiro.resource.ScriptGetScriptTypeResource module

networkapi.roteiro.resource.ScriptTypeAddResource module

networkapi.roteiro.resource.ScriptTypeAlterRemoveResource module

networkapi.roteiro.resource.ScriptTypeGetAllResource module

Module contents

networkapi.roteiro.test package

Submodules

networkapi.roteiro.test.test_Script module

networkapi.roteiro.test.test_ScriptType module

Module contents

Submodules

networkapi.roteiro.models module

Module contents

networkapi.semaforo package

Submodules

networkapi.semaforo.model module

Module contents

networkapi.test package

Submodules

networkapi.test.assertions module

networkapi.test.functions module

networkapi.test.mock_scripts module

networkapi.test.utils module

Module contents

networkapi.tipoacesso package

Subpackages

networkapi.tipoacesso.resource package

Submodules

networkapi.tipoacesso.resource.TipoAcessoResource module

Module contents

networkapi.tipoacesso.test package

Submodules

networkapi.tipoacesso.test.test_AccessType module

Module contents

Submodules

networkapi.tipoacesso.models module

Module contents

networkapi.usuario package

Subpackages

networkapi.usuario.resource package

Submodules

networkapi.usuario.resource.AuthenticateResource module

networkapi.usuario.resource.UserAddResource module

networkapi.usuario.resource.UserAlterRemoveResource module

networkapi.usuario.resource.UserGetAllResource module

networkapi.usuario.resource.UserGetByGroupUserOutGroup module

networkapi.usuario.resource.UserGetByGroupUserResource module

networkapi.usuario.resource.UserGetByIdResource module

networkapi.usuario.resource.UserGetByLdapResource module

networkapi.usuario.resource.UserGroupAssociateResource module

networkapi.usuario.resource.UserGroupDissociateResource module

networkapi.usuario.resource.UsuarioChangePassResource module

networkapi.usuario.resource.UsuarioGetResource module

Module contents

networkapi.usuario.test package

Submodules

networkapi.usuario.test.test_User module

networkapi.usuario.test.test_UserGroup module

Module contents

Submodules

networkapi.usuario.models module

Module contents

networkapi.vlan package

Subpackages

networkapi.vlan.resource package

Submodules

networkapi.vlan.resource.NetworkTypeResource module

networkapi.vlan.resource.TipoRedeResource module

networkapi.vlan.resource.VlanAllocateIPv6Resorce module

networkapi.vlan.resource.VlanAllocateResource module

networkapi.vlan.resource.VlanApplyAcl module

networkapi.vlan.resource.VlanCheckNumberAvailable module

networkapi.vlan.resource.VlanCreateAclResource module

networkapi.vlan.resource.VlanCreateResource module

networkapi.vlan.resource.VlanCreateScriptAclResource module

networkapi.vlan.resource.VlanDeallocateResource module

networkapi.vlan.resource.VlanEditResource module

networkapi.vlan.resource.VlanFindResource module

networkapi.vlan.resource.VlanGetByEnvironmentResource module

networkapi.vlan.resource.VlanInsertResource module

networkapi.vlan.resource.VlanInvalidateResource module

networkapi.vlan.resource.VlanListResource module

networkapi.vlan.resource.VlanRemoveResource module

networkapi.vlan.resource.VlanResource module

networkapi.vlan.resource.VlanSearchResource module

networkapi.vlan.resource.VlanValidateResource module

Module contents

networkapi.vlan.test package

Submodules

networkapi.vlan.test.test_NetType module

networkapi.vlan.test.test_Vlan module

Module contents

Submodules

networkapi.vlan.models module

Module contents

Submodules

networkapi.SQLLogMiddleware module

networkapi.admin_permission module

class networkapi.admin_permission.**AdminPermission**

Bases: object

ACCESS_TYPE_MANAGEMENT = 'cadastro_de_tipo_acesso'

ACL_APPLY = 'aplicar_acl'

ACL_VLAN_VALIDATION = 'validar_acl_vlans'

AS_MANAGEMENT = 'as_management'

AUDIT_LOG = 'audit_logs'

AUTHENTICATE = 'authenticate'

BRAND_MANAGEMENT = 'cadastro_de_marca'

ENVIRONMENT_MANAGEMENT = 'cadastro_de_ambiente'

ENVIRONMENT_VIP = 'ambiente_vip'

EQUIPMENT_GROUP_MANAGEMENT = 'cadastro_de_grupos_equipamentos'

EQUIPMENT_MANAGEMENT = 'cadastro_de_equipamentos'

EQUIP_READ_OPERATION = 'READ'

EQUIP_UPDATE_CONFIG_OPERATION = 'UPDATE_CONFIG'

EQUIP_WRITE_OPERATION = 'WRITE'

HEALTH_CHECK_EXPECT = 'healthcheck_expect'

IPS = 'ips'

LIST_CONFIG_BGP_DEPLOY_SCRIPT = 'list_config_bgp_deploy_script'

LIST_CONFIG_BGP_MANAGEMENT = 'list_config_bgp_management'

LIST_CONFIG_BGP_UNDEPLOY_SCRIPT = 'list_config_bgp_undeploy_script'

NEIGHBOR_DEPLOY_SCRIPT = 'neighbor_deploy_script'

NEIGHBOR_MANAGEMENT = 'neighbor_management'

NEIGHBOR_UNDEPLOY_SCRIPT = 'neighbor_undeploy_script'

NETWORK_FORCE = 'network_force'

NETWORK_TYPE_MANAGEMENT = 'cadastro_de_tipo_rede'

OBJ_DELETE_OPERATION = 'DELETE'

OBJ_READ_OPERATION = 'READ'

OBJ_TYPE_PEER_GROUP = 'PeerGroup'

OBJ_TYPE_POOL = 'ServerPool'

OBJ_TYPE_VIP = 'VipRequest'

OBJ_TYPE_VLAN = 'Vlan'

OBJ_UPDATE_CONFIG_OPERATION = 'UPDATE_CONFIG'

OBJ_WRITE_OPERATION = 'WRITE'

OPTION_VIP = 'opcao_vip'

PEER_GROUP_MANAGEMENT = 'peer_group_management'

POOL_ALTER_SCRIPT = 'script_alterar_pool'

POOL_CREATE_SCRIPT = 'script_criacao_pool'

POOL_DELETE_OPERATION = 'DELETE'

POOL_MANAGEMENT = 'cadastro_de_pool'

POOL_READ_OPERATION = 'READ'

POOL_REMOVE_SCRIPT = 'script_remover_pool'

POOL_UPDATE_CONFIG_OPERATION = 'UPDATE_CONFIG'

POOL_WRITE_OPERATION = 'WRITE'

READ_OPERATION = 'READ'

ROUTE_MAP_DEPLOY_SCRIPT = 'route_map_deploy_script'

ROUTE_MAP_MANAGEMENT = 'route_map_management'

ROUTE_MAP_UNDEPLOY_SCRIPT = 'route_map_undeploy_script'

SCRIPT_MANAGEMENT = 'cadastro_de_roteiro'

TELCO_CONFIGURATION = 'configuracao_telco'

USER_ADMINISTRATION = 'administracao_usuarios'

VIPS_REQUEST = 'requisicao_vips'

VIP_ALTER_SCRIPT = 'script_alterar_vip'

VIP_CREATE_SCRIPT = 'script_criacao_vip'

VIP_DELETE_OPERATION = 'DELETE'

VIP_READ_OPERATION = 'READ'

VIP_REMOVE_SCRIPT = 'script_remover_vip'

VIP_UPDATE_CONFIG_OPERATION = 'UPDATE_CONFIG'

VIP_VALIDATION = 'validar_vip'

VIP_WRITE_OPERATION = 'WRITE'

VLAN_ALLOCATION = 'alocar_vlan'

VLAN_ALTER_SCRIPT = 'script_alterar_vlan'

VLAN_CREATE_SCRIPT = 'script_criacao_vlan'

VLAN_MANAGEMENT = 'cadastro_de_vlans'

VM_MANAGEMENT = 'cadastro_de_vm'

WRITE_OPERATION = 'WRITE'

networkapi.conftest module

networkapi.cvs module

exception networkapi.cvs.**CVSCommandError**(*error*)
Bases: networkapi.cvs.CVSError

exception networkapi.cvs.**CVSError**(*error*)
Bases: exceptions.Exception

class networkapi.cvs.**Cvs**

classmethod add(*archive*)
Execute command add in cvs

@param archive: file to be add

@raise CVSCommandError: Failed to execute command

classmethod commit(*archive*, *comment*)
Execute command commit in cvs

@param archive: file to be committed @param comment: comments

@raise CVSCommandError: Failed to execute command

classmethod remove(*archive*)
Execute command remove in cvs

@param archive: file to be remove

@raise CVSCommandError: Failed to execute command

classmethod synchronization()
Execute command update in cvs

@raise CVSCommandError: Failed to execute command

networkapi.environment_settings module

networkapi.settings.**local_files**(*path*)

networkapi.error_message_utils module

networkapi.error_message_utils.**error_dumps**(*code*, **args*)

networkapi.exception module

exception networkapi.exception.**AddBlockOverrideNotDefined**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

Represents an error occurred when attempting to change a VIP that has not been created.

exception networkapi.exception.**CustomException**(*cause*, *message=None*)
Bases: exceptions.Exception

Represents an error occurred validating a value.

exception networkapi.exception.**EnvironmentEnvironmentServerPoolLinked**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

returns exception to EnvironmentEnvironmentVip error.

exception networkapi.exception.**EnvironmentEnvironmentVipDuplicatedError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

returns exception to EnvironmentEnvironmentVip duplicated.

exception networkapi.exception.**EnvironmentEnvironmentVipError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

returns exception to EnvironmentEnvironmentVip error.

exception networkapi.exception.**EnvironmentEnvironmentVipNotFoundError**(*cause*, *message=None*)

Bases: networkapi.exception.CustomException

returns exception to EnvironmentEnvironmentVip research by primary key.

exception networkapi.exception.**EnvironmentNotFoundError**(*cause*, *message=None*)

Bases: networkapi.exception.CustomException

returns exception to Environment research by primary key.

exception networkapi.exception.**EnvironmentVipAssociatedToSomeNetworkError**(*cause*, *message=None*)

Bases: networkapi.exception.EnvironmentVipError

returns exception to environment vip delete when it's associated to some Network

exception networkapi.exception.**EnvironmentVipError**(*cause*, *message=None*)

Bases: networkapi.exception.CustomException

Represents an error occurred during access to tables related to environment VIP.

exception networkapi.exception.**EnvironmentVipNotFoundError**(*cause*, *message=None*)

Bases: networkapi.exception.EnvironmentVipError

returns exception to environment research by primary key.

exception networkapi.exception.**EquipmentGroupsNotAuthorizedError**(*cause*, *message=None*)

Bases: networkapi.exception.CustomException

Represents an error when the groups of equipment registered with the IP of the VIP request is not allowed acess.

exception networkapi.exception.**InvalidValueError**(*cause*, *param=None*, *value=None*)

Bases: exceptions.Exception

Represents an error occurred validating a value.

exception networkapi.exception.**NetworkActiveError**(*cause=None*, *message=None*)

Bases: networkapi.exception.CustomException

Exception returned when network is active and someone is trying to remove it

DEFAULT_MESSAGE = "Can't remove network because it is active"

exception networkapi.exception.**NetworkInactiveError**(*cause=u'Unable to remove the network because it is inactive.'*, *message=None*)

Bases: networkapi.exception.CustomException

Returns exception when trying to disable a network disabled

exception networkapi.exception.**OptionPoolEnvironmentDuplicatedError**(*cause*, *message=None*)

Bases: networkapi.exception.OptionPoolEnvironmentError

returns exception if OptionPool is already associated with EnvironmentVip.

exception networkapi.exception.**OptionPoolEnvironmentError**(*cause*, *message=None*)

Bases: networkapi.exception.CustomException

Represents an error occurred during access to tables related to OptionPoolEnvironmentVip.

exception networkapi.exception.**OptionPoolEnvironmentNotFoundError**(*cause*, *message=None*)
Bases: networkapi.exception.OptionPoolEnvironmentError

returns exception to OptionPoolEnvironmentVip research by primary key.

exception networkapi.exception.**OptionPoolError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

Represents an error occurred during access to tables related to Option Pool.

exception networkapi.exception.**OptionPoolNotFoundError**(*cause*, *message=None*)
Bases: networkapi.exception.OptionPoolError

returns exception to Option pool research by primary key.

exception networkapi.exception.**OptionPoolServiceDownNoneError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

returns exception if OptionPool service-down-action "none" option does not exists.

exception networkapi.exception.**OptionVipEnvironmentVipDuplicatedError**(*cause*, *message=None*)
Bases: networkapi.exception.OptionVipEnvironmentVipError

returns exception if OptionVip is already associated with EnvironmentVip.

exception networkapi.exception.**OptionVipEnvironmentVipError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

Represents an error occurred during access to tables related to OptionVipEnvironmentVip.

exception networkapi.exception.**OptionVipEnvironmentVipNotFoundError**(*cause*, *message=None*)
Bases: networkapi.exception.OptionVipEnvironmentVipError

returns exception to OptionVipEnvironmentVip research by primary key.

exception networkapi.exception.**OptionVipError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

Represents an error occurred during access to tables related to Option VIP.

exception networkapi.exception.**OptionVipNotFoundError**(*cause*, *message=None*)
Bases: networkapi.exception.OptionVipError

returns exception to Option vip research by primary key.

exception networkapi.exception.**RequestVipsNotBeenCreatedError**(*cause*, *message=None*)
Bases: networkapi.exception.CustomException

Represents an error occurred when attempting to change a VIP that has not been created.

networkapi.log module

class networkapi.log.**CommonAdminEmailHandler**(*include_html=False*)
Bases: django.utils.log.AdminEmailHandler

An exception log handler that e-mails log entries to site admins. If the request is passed as the first argument to the log record, request data will be provided in the

emit(*record*)

class networkapi.log.**Log**(*module_name*)
Bases: object

Classe responsável por encapsular a API de logging. Encapsula as funcionalidades da API de logging de forma a adicionar o nome do módulo nas mensagens que forem impressas.

debug(*msg*, **args*)
Imprime uma mensagem de debug no log

error(*msg*, **args*)
Imprime uma mensagem de erro no log

info(*msg*, **args*)
Imprime uma mensagem de informação no log

classmethod init_log(*log_file_name='/tmp/networkapi.log'*, *number_of_days_to_log=10*, *log_level=10*, *log_format='%(asctime)s %(request_user)-6s %(request_path)-8s %(request_id)-6s %(levelname)-6s - %(message)s'*, *use_stdout=True*, *max_line_size=2048*)

rest(*msg*, **args*)

warning(*msg*, **args*)
Imprime uma mensagem de advertência no log

class networkapi.log.**MultiprocessTimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*)
Bases: logging.handlers.TimedRotatingFileHandler

doRollover()
do a rollover; in this case, a date/time stamp is appended to the filename when the rollover happens. However, you want the file to be named for the start of the interval, not the current time. If there is a backup count, then we have to get a list of matching filenames, sort them and remove the one with the oldest suffix.

class networkapi.log.**NetworkAPILogFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

formatException(*ei*)

networkapi.log.**convert_to_utf8**(*object*)
Converte o object informado para uma representação em utf-8

networkapi.log.**get_lock**()
Obtém lock para evitar que várias mensagens sejam sobrepostas no log

networkapi.log.**release_lock**()
Obtém lock para evitar que várias mensagens sejam sobrepostas no log

networkapi.processExceptionMiddleware module

class networkapi.processExceptionMiddleware.**LoggingMiddleware**
Bases: object

process_exception(*request*, *exception*)
HIDE PASSWORD VALUES

networkapi.rest module

networkapi.settings module

networkapi.settings.**local_files**(*path*)

networkapi.sitecustomize module

networkapi.teste module

networkapi.urls module

networkapi.util module

Module contents

CHAPTER 7

Using GloboNetworkAPI V3

Improve GET requests through some extra parameters

When making GET request in V3 routes, you can choose what fields will come into response using the following parameters: **kind**, **fields**, **include** and **exclude**. When none of these parameters are used, NetworkAPI will return a default payload for each module. Depending on your needs, the use of these extra parameters will make your requests faster mainly if you are dealing with many objects. In addition, it is possible to obtain more information about fields that acts as a foreign keys. Look at the examples in each section to understand better.

Vip Request and *Network IPv4* modules are used in the examples, consult them to obtain more information about its payload.

Kind parameter

Each module returns a default payload when none of extra parameters are used. With **kind** parameter you can change the default payload to some other two. Look the modules documentation for know about these payloads. **kind** accepts only 'basic' or 'details'. In general, the payload for 'basic' contains little information while 'details' contains so much data.

Suppose that you want to get the basic payload in *Vip Request*. Use this:

```
kind=basic
```

Fields parameter

The **fields** parameter is used when you want to get only the fields that you specify.

Suppose that you want only id and name fields in *Vip Request*. Use this:

```
fields=id,name
```

Include parameter

The include parameter is used to append some field which is not contained on the default payload. Do not use this together **fields**.

Suppose that you want to get the default payload plus 'dscp' and 'equipments' fields in *Vip Request*. Use this:

```
include=dscp,equipments
```

Exclude parameter

The exclude parameter is used to remove some field of the default payload. Do not use this together **fields**.

Suppose that you want to get the default payload except 'ipv4' and 'ipv6' fields in *Vip Request*. Use this:

```
exclude=ipv4,ipv6
```

Using Include and Exclude together

Suppose that you want to get the default payload except 'ipv4' field and plus 'dscp' field in *Vip Request*. Use this:

```
exclude=ipv4&include=dscp
```

Using Kind and Include together

Suppose that you want to get the basic payload plus 'dscp' field in *Vip Request*. Use this:

```
kind=basic&include=dscp
```

Using Kind and Exclude together

Suppose that you want to get the details payload except 'ipv4' field in *Vip Request*. Use this:

```
kind=details&exclude=ipv4
```

Using Kind, Include and Exclude together

Suppose that you want to get the basic payload plus 'dscp' field and except 'ipv4' field in *Vip Request*. Use this:

```
kind=basic&include=dscp&exclude=ipv4
```

Getting more information from fields that acts as a foreign key

Through **fields** and **include** parameters, you can obtain more information for fields that acts as a foreign key. If you are dealing with such a field, you can through this 'descend or rise' like a tree.

For a simple example, suppose that you make a GET Request for *Network IPv4 module* to get only vlan field. You certainly would use this:

```
fields=vlan
```

Doing the above, you will get only the identifier of the Vlan. But you want not only the identifier, but also the name of the Vlan. Instead of create a new request for Vlan module, you can at same Network IPv4 request obtain this information. See below how to do this:

```
fields=vlan__details
```

Now, Vlan field is not anymore an integer field, but it is a dictionary with some more information as the vlan name and the identifier of environment related to this Vlan. Let's say now you want the name of this Environment. Again you don't need to create a new request to Environment module, because using the same Network IPv4 request you can get this information. Look below the way to do this:

```
fields=vlan__details__environment__basic
```

Now you have only one JSON with information from various places. In this way you can obtain lots of information in a faster way relieving Network API and reducing time for your application to get a lot of data that is related to each other.

Datacenter module

Data Center /api/dc/

POST

Creating a Data Center object

URL:

```
/api/dc/
```

Request body:

```
{
    "dc": {
    "dcname": <string>,
    "address": <string>
    }
}
```

Request Example:

```
{
    "dc": {
    "dcname": "POP-SP",
    "address": "SP"
    }
}
```

All fields are required:

- **dcname** - It is the name of the Data Center.

  address - It is the location of the Data Center.

At the end of POST request, it will be returned a json with the Data Center object created.

Response Body:

```
{
    "dc": {
    "id": 1
    "dcname": "POP-SP",
    "address": "SP"
    }
}
```

PUT

Editing a Data Center object

GET

Obtaining list of Data Centers

URL:

```
/api/dc/
```

Default behavior

The response body will look like this:

Response body:

```
{
    "dc": [{
        "id": <integer>,
        "dcname": <string>,
        "address": <string>,
        "fabric": <list>
    },...]
}
```

DELETE

Deleting a Data Center object

URL:

```
/api/dc/<dc_id>/
```

where **dc_id** is the identifier of Data Center's desired to delete.

Example with Parameter ID:

/api/dc/1/

Fabric /api/dcrooms/

POST

Creating a Fabric object

URL:

```
/api/dcrooms/
```

Request body:

```
{
    "dcrooms": {
        "dc": <integer:dc_fk>,
        "name": <string>,
        "racks": <integer>,
        "spines": <integer>,
        "leafs": <integer>,
        "config": <dict>
    }
}
```

Request Example:

```
{
    "dcrooms": {
        "dc": 1,
        "name":"Fabric name",
        "racks": 32,
        "spines": 4,
        "leafs": 2,
        "config": {}
    }
}
```

- **dc** - It is the fk of the Data Center.

  name - It is the name of the Fabric. **racks** - Total number of the racks in a fabric. **spines** - Total number of the spines in a fabric. **leafs** - Total number of the leafes in a fabric. **config** - Json with the father's environments related to the fabric and it's peculiarities.

Only fields 'dc' and 'name' are required.

Example of config json:

```
{
  "BGP": {
    "spines": <string: AS Number>,
    "mpls": <string: AS Number>,
    "leafs": <string: AS Number>
  },
  "Gerencia": {
    "telecom": {
      "vlan": <string: Vlan Number>,
      "rede": <string: IPv4 Net>
    }
  },
  "VLT": {
    "id_vlt_lf1": <string: VLT ID Number>,
    "priority_vlt_lf1": <string: VLT priority Number>,
    "priority_vlt_lf2": <string: VLT priority Number>,
    "id_vlt_lf2": <string: VLT ID Number>
  },
  "Ambiente": [
    {
      "id": <integer: env_fk>,
      "details": [
        {
          "name": <string: Name of the new environment - E.g.: BEFE>,
          "min_num_vlan_1": <integer: Minimum number for Vlan>,
          "max_num_vlan_1": <integer: Maximum number for Vlan>,
```

```
          "config": [
            {
              "subnet": <string: IPv4 or IPv6 Net>,
              "type": <string: v4 or v6>,
              "mask": <integer: net mask>,
              "network_type": <integer: net_type_fk>,
              "new_prefix": <integer: subnet mask>
            },...
          ]
        }, ...
      ]
    },
    {
      "id": <integer: env_fk>,
      "details": []
    },...
    {
      "id": <integer: env_fk>,
      "details": [
        {
          "v4": {
            "new_prefix": <string: subnet mask>
          },
          "v6": {
            "new_prefix": <string: subnet mask>
          }
        }
      ]
    },...
  ],
  "Channel": {
    "channel": <string: Port Channel base Number>
  }
}
```

At the end of POST request, it will be returned a json with the Fabric object created.

Response Body:

```
{
    "dcrooms": {
        "id": 1
        "dc": 1,
        "name":"Fabric name",
        "racks": 32,
        "spines": 4,
        "leafs": 2,
        "config": {}
    }
}
```

GET

Obtaining list of Fabrics

URL:

```
/api/dcrooms/
```

Get a Fabric by id

URL:

```
/api/dcrooms/<fabric_fk>
```

where **fabric_fk** is the identifier of the fabric desired to be retrieved.

Get a Fabric by the datacenters id

URL:

```
/api/dcrooms/dc/<dc_fk>
```

where **dc_fk** is the identifier of the datacenter.

Default behavior

The response body will look like this:

Response body:

```
{
    "fabric": [{
        "id": 32,
        "name": "POPF",
        "dc": 3,
        "racks": 8,
        "spines": 4,
        "leafs": 2,
        "config": {...}
    },
    ...]
}
```

PUT

Editing a Fabric object

URL:

```
/api/dcrooms/
```

Request body:

```
{
    "dcrooms": {
        "id": <interger:fabric_fk>,
        "dc": <integer:dc_fk>,
        "name": <string>,
        "racks": <integer>,
        "spines": <integer>,
```

```
        "leafs": <integer>,
        "config": <dict>
    }
}
```

Request Example:

```
{
    "dcrooms": {
        "id": 1,
        "dc": 1,
        "name":"Fabric name",
        "racks": 32,
        "spines": 4,
        "leafs": 2,
        "config": {...}
    }
}
```

Through Fabric PUT route you can update a object. These fields are required:

- **id** - It is the fk of the Fabric.

  dc - It is the fk of the Data Center. **name** - It is the name of the Fabric.

At the end of PUT request, it will be returned the Fabric object updated.

Response Body:

{

"dcrooms": { "id": 1 "dc": 1, "name":"Fabric name", "racks": 32, "spines": 4, "leafs": 2, "config": {...}

}

}

DELETE

Deleting a Fabric object

URL:

```
/api/dcrooms/
```

Racks /api/rack/

GET

Obtaining list of Racks

URL:

```
/api/rack/
```

Get a list of Racks by the fabric id URL:

```
/api/rack/fabric/<fabric_fk>
```

where **fabric_fk** is the identifier of the fabric.

Get a Rack by id URL:

```
/api/rack/<rack_fk>
```

where **rack_fk** is the identifier of the Rack desired to be retrieved.

Default behavior The response body will look like this:

Response body:

```
{
    "racks": [{
        "config": false,
        "create_vlan_amb": false,
        "dcroom": 1,
        "id": 10,
        "id_ilo": "OOB-CM-TE01",
        "id_sw1": "LF-CM-TE01-1",
        "id_sw2": "LF-CM-TE01-2",
        "mac_ilo": "3F:FF:FF:FF:FF:10",
        "mac_sw1": "1F:FF:FF:FF:FF:10",
        "mac_sw2": "2F:FF:FF:FF:FF:10",
        "nome": "TE10",
        "numero": 10
        },...
    ]
}
```

POST

Creating a Rack object

URL:

```
/api/rack/
```

Request body:

```
{
    "rack":{
        "name": <string>,
        "number": <integer>,
        "mac_sw1": <string:mac_address>,
        "mac_sw2": <string:mac_address>,
        "mac_ilo": <string:mac_address>,
        "id_sw1": <integer:equipment_fk>,
        "id_sw2": <integer:equipment_fk>,
        "id_ilo": <integer:equipment_fk>,
        "dcroom": <integer:fabric_fk>
    }
}
```

Request Example:

```
{
    "rack":{
        "name": "TE01",
        "number": 2,
        "mac_sw1": "1F:FF:FF:FF:FF:FF",
        "mac_sw2": "2F:FF:FF:FF:FF:FF",
        "mac_ilo": "3F:FF:FF:FF:FF:FF",
        "id_sw1": 1,
        "id_sw2": 2,
        "id_ilo": 3,
        "dcroom": 16
    }
}
```

- **dcroom** - It is the fk of the Fabric.

  name - It is the name of the Rack. **number** - It is the number of the Rack. **mac_sw[1,2]** - It is the mac address from each switch. **id_sw[1,2]** - It is the fk from each switch.

Only fields 'name' and 'number' are required.

At the end of POST request, it will be returned a json with the Rack object created.

Response Body:

```
{
    "rack": {
        "config": false,
        "create_vlan_amb": false,
        "dcroom": 16,
        "id": 10,
        "id_ilo": 3,
        "id_sw1": 1,
        "id_sw2": 2,
        "mac_ilo": "3F:FF:FF:FF:FF:FF",
        "mac_sw1": "1F:FF:FF:FF:FF:FF",
        "mac_sw2": "2F:FF:FF:FF:FF:FF",
        "nome": TE01,
        "numero": 2
        }
}
```

PUT

Editing a Rack object

URL:

`/api/rack/<rack_fk>`

Request body:

{

"rack":

{ “config”: <boolean>, “create_vlan_amb”: <boolean>, “fabric_id”: <integer:fabric_id>, “id”: <integer:rack_id>, “id_ilo”: <integer:equipment_id>, “id_sw1”: <integer:equipment_id>, “id_sw2”: <integer:equipment_id>, “mac_ilo”: <string:mac_address>, “mac_sw1”: <string:mac_address>, “mac_sw2”: <string:mac_address>, “nome”: “PUT10”, “numero”: <integer>

}

}

Request Example:

{

“rack”:

{ “config”: false, “create_vlan_amb”: false, “fabric_id”: 1, “id”: 10, “id_ilo”: 3, “id_sw1”: 1, “id_sw2”: 2, “mac_ilo”: “3F:FF:FF:FF:FF:FF”, “mac_sw1”: “1F:FF:FF:FF:FF:FF”, “mac_sw2”: “2F:FF:FF:FF:FF:FF”, “nome”: “PUT10”, “numero”: 10

}

}

Through PUT route you can update a rack object. These fields are required:

- **id** - It is the fk of the rack.

  numero - It is the number of the rack. **nome** - It is the name of the rack.

At the end of PUT request, it will be returned the rack object updated.

Response Body:

{

“rack”: { “config”: false, “create_vlan_amb”: false, “dcroom”: 1, “id”: 10, “id_ilo”: “OOB-CM-TE01”, “id_sw1”: “LF-CM-TE01-1”, “id_sw2”: “LF-CM-TE01-2”, “mac_ilo”: “3F:FF:FF:FF:FF:FF”, “mac_sw1”: “1F:FF:FF:FF:FF:FF”, “mac_sw2”: “2F:FF:FF:FF:FF:FF”, “nome”: null, “numero”: null

}

}

DELETE

Deleting a Rack object

URL:

```
/api/rack/<rack_fk>/
```

where **rack_fk** is the identifier of Rack’s desired to delete.

Example with Parameter ID:

```
/api/rack/1/
```

Environment module

/api/v3/environment/

GET

Obtaining list of Environments

It is possible to specify in several ways fields desired to be retrieved in Environment module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Environment module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- name
- **grupo_l3**
- **ambiente_logico**
- **divisao_dc**
- **filter**
- acl_path
- ipv4_template
- ipv6_template
- link
- min_num_vlan_1
- max_num_vlan_1
- min_num_vlan_2
- max_num_vlan_2
- vrf
- **default_vrf**
- *father_environment*
- *children*
- **configs**
- *routers*
- *equipments*

Obtaining list of Environments through id's URL:

```
/api/v3/environment/[environment_ids]/
```

where **environment_ids** are the identifiers of Environments desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/environment/1/
```

Many IDs:

```
/api/v3/environment/1;3;8/
```

Obtaining list of Environments through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/environment/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/environment/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "divisao_dc": 1,
        "ambiente_logico__nome": "AmbLog"
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and grupo_l3:

```
fields=id,name,grupo_l3
```

Using kind GET parameter

The Environment module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "environments": [{
        "id": <integer>,
        "name": <string>
    }]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "environments": [{
        "id": <integer>,
        "name": <string>,
        "grupo_l3": {
            "id": <integer>,
            "name": <string>
        },
        "ambiente_logico": {
            "id": <integer>,
            "name": <string>
        },
        "divisao_dc": {
            "id": <integer>,
            "name": <string>
        },
        "filter": <integer>,
        "acl_path": <string>,
        "ipv4_template": <string>,
        "ipv6_template": <string>,
        "link": <string>,
        "min_num_vlan_1": <integer>,
        "max_num_vlan_1": <integer>,
        "min_num_vlan_2": <integer>,
        "max_num_vlan_2": <integer>,
        "default_vrf": {
            "id": <integer>,
            "internal_name": <string>,
            "vrf": <string>
        },
        "father_environment": <recurrence-to:environment>
    }]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "environments": [{
        "id": <integer>,
        "name": <string>,
        "grupo_l3": <integer>,
        "ambiente_logico": <integer>,
        "divisao_dc": <integer>,
        "filter": <integer>,
        "acl_path": <string>,
        "ipv4_template": <string>,
        "ipv6_template": <string>,
        "link": <string>,
        "min_num_vlan_1": <integer>,
        "max_num_vlan_1": <integer>,
        "min_num_vlan_2": <integer>,
        "max_num_vlan_2": <integer>,
        "default_vrf": <integer>,
        "father_environment": <integer>
    },...]
}
```

/api/v3/environment/environment-vip/

GET

Obtaining environments associated to environment vip

URL:

```
/api/v3/environment/environment-vip/<environment_vip_id>/
```

where **environment_vip_id** is the identifier of the environment vip used as an argument to retrieve associated environments. Only one **environment_vip_id** can be assigned. The instruction related to use of extra GET parameters (**kind**, **fields**, **include** and **exclude**) and the default response body is the same as described in *Environment GET Module*

Example:

```
/api/v3/environment/environment-vip/1/
```

Environment Vip module

/api/v3/environment-vip/

GET

Obtaining list of Environment Vip

It is possible to specify in several ways fields desired to be retrieved in Environment Vip module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Environment Vip module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable):

- id
- finalidade_txt
- cliente_txt
- ambiente_p44_txt
- description
- name
- conf
- **optionsvip**
  - **option**
  - *environment_vip*
- **environments**
  - *environment*
  - *environment_vip*

Obtaining list of Environment Vip through id's URL:

```
/api/v3/environment-vip/[environment_vip_ids]/
```

where **environment_vip_ids** are the identifiers of Environments Vip desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/environment-vip/1/
```

Many IDs:

```
/api/v3/environment-vip/1;3;8/
```

Obtaining list of Environment Vip through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/environment-vip/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/environment-vip/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "description__icontains": "BE",
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and environments:

```
fields=id,name,environments
```

Using kind GET parameter

The Environment Vip module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "environments_vip": [{
        "id": <integer>,
```

```
        "name": <string>
    },...]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "environments_vip": [{
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>,
        "name": <string>,
        "conf": <string>
    },...]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "environments_vip": [{
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>
    },...]
}
```

/api/v3/environment-vip/step

GET

Obtaining finality list

URL:

```
/api/v3/environment-vip/step/
```

Optional GET Parameter:

```
environmentp44=[string]
```

Example:

Without environmentp44 GET Parameter:

```
/api/v3/environment-vip/step/
```

With environmentp44 GET Parameter:

```
/api/v3/environment-vip/step/?environmentp44=[string]
```

where **environmentp44** is a characteristic of environment vips. This argument is not case sensitive. The URL above accepts other GET Parameters, but the type of response will be different depending on what GET Parameters are sent to API. Therefore, to obtain finality list, the URL should have no argument or have the optional environmentp44 argument. Don't forget to encode URL.

Response body:

```
[
    {
        "finalidade_txt": <string>
    },...
]
```

Obtaining client list through finality

URL:

```
/api/v3/environment-vip/step/
```

Required GET Parameter:

```
finality=[string]
```

Example:

```
/api/v3/environment-vip/step/?finality=[string]
```

where **finality** is a characteristic of environment vips. This argument is not case sensitive. The URL above accepts other GET Parameters, but the type of response will be different depending on what GET Parameters are sent to API. Therefore, to obtain client list ONLY pass **finality** parameter into URL. Don't forget to encode URL.

Response body:

```
[
    {
        "cliente_txt": <string>
    },...
]
```

Obtaining environment vip list through finality and client

URL:

```
/api/v3/environment-vip/step/
```

Required GET Parameters:

```
finality=[string]
client=[string]
```

Example:

```
/api/v3/environment-vip/step/?finality=[string]&client=[string]
```

where **finality** and **client** are characteristics of environment vips. These arguments are not case sensitive. The URL above accepts other GET Parameters, but the type of response will be different depending on what GET Parameters are sent to API. Therefore, to obtain environment list ONLY pass **finality** and **client** parameters into URL. Don't forget to encode URL. The instruction related to use of extra GET parameters (**kind**, **fields**, **include** and **exclude**) and the default response body is the same as described in *Environment Vip GET Module*.

Response body:

```
[
    {
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>
    },...
]
```

Obtaining environment vip through finality, client and environmentp44

URL:

```
/api/v3/environment-vip/step/
```

Required GET Parameters:

```
finality=[string]
client=[string]
environmentp44=[string]
```

Example:

```
/api/v3/environment-vip/step/?finality=[string]&client=[string]&environmentp44=[string]
```

where **finality**, **client** and **environmentp44** are characteristics of environment vips. These arguments are not case sensitive . To obtain only one environment vip you must pass the three parameters described above into URL. Don't forget to encode URL. The instruction related to use of extra GET parameters (**kind**, **fields**, **include** and **exclude**) and the default response body is the same as described in *Environment Vip GET Module*.

Response body:

```
[
    {
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>
```

```
    },...
]
```

Equipment module

/api/v3/equipment/

GET

Obtaining list of Equipments

It is possible to specify in several ways fields desired to be retrieved in Equipment module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Equipment module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- name
- maintenance
- **equipment_type**
- **model**
  - name
  - **brand**
    - id
    - name
- *ipv4*
- *ipv6*
- **environments**
  - *environment*
  - *equipment*
- **groups**

Obtaining list of Equipments through some Optional GET Parameters URL:

```
/api/v3/equipment/
```

Optional GET Parameters:

```
rights_write=[string]
environment=[integer]
ipv4=[string]
ipv6=[string]
```

```
is_router=[integer]
name=[string]
```

Where:

- **rights_write** must receive 1 if desired to obtain the equipments where at least one group to which the user logged in is related has write access.
- **environment** is some environment identifier.
- **ipv4** and **ipv6** are IP's must receive some valid IP Adresss.
- **is_router** must receive 1 if only router equipments are desired, 0 if only equipments that is not routers are desired.
- **name** is a unique string that only one equipment has.

Example:

With environment and ipv4 GET Parameter:

```
/api/v3/equipment/?ipv4=192.168.0.1&environment=5
```

Obtaining list of Equipments through id's URL:

```
/api/v3/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of Equipments desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/equipment/1/
```

Many IDs:

```
/api/v3/equipment/1;3;8/
```

Obtaining list of Equipments through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/equipment/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/equipment/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "maintenance": false,
        "tipo_equipamento": 1
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and maintenance:

```
fields=id,name,maintenance
```

Using kind GET parameter

The Equipment module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field equipment_type for *basic* will contain only the identifier and for *details* will contain also the description.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "equipments": [{
        "id": <integer>,
        "name": <string>
    }]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "equipments": [{
        "id": <integer>,
        "name": <string>,
        "maintenance": <boolean>,
        "equipment_type": {
```

```
            "id": <integer>,
            "equipment_type": <string>
        },
        "model": {
            "id": <integer>,
            "name": <string>
        },
        "ipv4": [{
            "id": <integer>,
            "oct1": <integer>,
            "oct2": <integer>,
            "oct3": <integer>,
            "oct4": <integer>,
            "networkipv4": <integer>,
            "description": <string>
        },...],
        "ipv6": [{
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "networkipv6": <integer>,
            "description": <string>
        },...],
        "environments": [{
            "is_router": <boolean>,
            "environment": {
                "id": <integer>,
                "name": <name>
                "grupo_l3": <integer>,
                "ambiente_logico": <integer>,
                "divisao_dc": <integer>,
                "filter": <integer>,
                "acl_path": <string>,
                "ipv4_template": <string>,
                "ipv6_template": <string>,
                "link": <string>,
                "min_num_vlan_1": <integer>,
                "max_num_vlan_1": <integer>,
                "min_num_vlan_2": <integer>,
                "max_num_vlan_2": <integer>,
                "vrf": <string>,
                "default_vrf": <integer>
            }
        },...],
        "groups": [{
            "id": <integer>,
            "name": <string>
        },...]
    },...]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "equipments": [{
        "id": <integer>,
        "name": <string>,
        "maintenance": <boolean>,
        "equipment_type": <integer>,
        "model": <integer>
    },...]
}
```

POST

Creating list of equipments

URL:

```
/api/v3/equipment/
```

Request body:

```
{
    "equipments": [{
        "environments": [
            {
                "id": <integer:environment_fk>,
                "is_router": <boolean>
            },...
        ],
        "equipment_type": <integer:equip_type_fk>,
        "groups": [
            {
                "id": <integer:group_fk>
            },...
        ],
        "ipv4": [
            {
                "id": <integer:ipv4_fk>
            }
        ],
        "ipv6" [
            {
                "id": <integer:ipv6_fk>
```

```
            }
        ],
        "maintenance": <boolean>,
        "model": <integer:model_fk>,
        "name": <string>
    },...]
}
```

- **environments** - You can associate environments to new Equipment and specify if your equipment in each association will act as a router for specific environment.
- **equipment_type** - You must specify if your Equipment is a Switch, a Router, a Load Balancer...
- **groups** - You can associate the new Equipment to one or more groups of Equipments.
- **ipv4** - You can assign to the new Equipment how many IPv4 addresses is needed.
- **ipv6** - You can assign to the new Equipment how many IPv6 addresses is needed.
- **maintenance** - You must assign to the new Equipment a flag saying if the Equipment is or not in maintenance mode.
- **model** - You must assign to the Equipment some model (Cisco, Dell, HP, F5, ...).
- **name** - You must assign to the Equipment any name.

URL Example:

```
/api/v3/equipment/
```

PUT

Updating list of equipments in database

URL:

```
/api/v3/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of equipments. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/equipment/1/
```

Many IDs:

```
/api/v3/equipment/1;3;8/
```

Request body:

```
{
    "equipments": [{
        "id": <integer>,
        "environments": [
            {
                "id": <integer:environment_fk>,
                "is_router": <boolean>
            },...
        ],
```

```
        "equipment_type": <integer:equip_type_fk>,
        "groups": [
            {
                "id": <integer:group_fk>
            },...
        ],
        "ipv4": [
            {
                "id": <integer:ipv4_fk>
            }
        ],
        "ipv6" [
            {
                "id": <integer:ipv6_fk>
            }
        ],
        "maintenance": <boolean>,
        "model": <integer:model_fk>,
        "name": <string>
    },..]
}
```

- **environments** - You can associate environments to new Equipment and specify if your equipment in each association will act as a router for specific environment.
- **equipment_type** - You must specify if your Equipment is a Switch, a Router, a Load Balancer...
- **groups** - You can associate the new Equipment to one or more groups of Equipments.
- **ipv4** - You can assign to the new Equipment how many IPv4 addresses is needed.
- **ipv6** - You can assign to the new Equipment how many IPv6 addresses is needed.
- **maintenance** - You must assign to the new Equipment a flag saying if the Equipment is or not in maintenance mode.
- **model** - You must assign to the Equipment some model (Cisco, Dell, HP, F5, ...).
- **name** - You must assign to the Equipment any name.

Remember that if you don't provide the not mandatory fields, actual information (e.g. associations between Equipment and Environments) will be deleted. The effect of PUT Request is always to replace actual data by what you provide into fields in this type of request.

URL Example:

```
/api/v3/equipment/1/
```

DELETE

Deleting a list of equipments in database

Deleting list of equipments and all relationships URL:

```
/api/v3/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of equipments desired to delete. It can use multiple id's separated by semicolons. Doing this, all associations between Equipments and IP addresses, Access, Script (Roteiro), Interface, Environment and Group will be deleted.

Example with Parameter IDs:

One ID:

```
/api/v3/equipment/1/
```

Many IDs:

```
/api/v3/equipment/1;3;8/
```

Option Pool module

/api/v3/option-pool/environment

GET

Obtaining options pools associated to environment

URL:

```
/api/v3/option-pool/environment/<environment_id>/
```

where **environment_id** is the identifier of the environment used as an argument to retrieve associated option pools. It's mandatory to assign one and only one **environment_id**.

Example:

```
/api/v3/option-pool/environment/1/
```

Response body:

```
{
    "options_pool": [{
        "id": <integer>,
        "type": "string",
        "name": "string"
    }, ...]
}
```

Option Vip module

/api/v3/option-vip/environment-vip

GET

Obtaining list of Options Vip

It is possible to specify in several ways fields desired to be retrieved in Options Vip module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Options Vip module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- **option**
- *environment_vip*

Obtaining options vip through environment vip URL:

```
/api/v3/option-vip/environment-vip/<environment_vip_id>
```

where **environment_vip_id** is the identifier of environment vip used as an argument to retrieve associated options vip. It's mandatory to assign one and only one identifier to **environment_vip_id**.

Example:

```
/api/v3/option-vip/environment-vip/1
```

Default Response body:

```
[
    {
        "option": {
            "id": <integer>,
            "tipo_opcao": <string>,
            "nome_opcao_txt": <string>
        },
        "environment-vip": <integer>
    },...
]
```

/api/v3/option-vip/environment-vip/type-option

GET

Obtaining options vip through environment vip and type option

URL:

```
/api/v3/option-vip/environment-vip/<environment_vip_id>/type-option/<type_option>/
```

where **environment_vip_id** is the identifier of environment vip used as an argument to retrieve associated options vip and **type_option** is a string that filter the result by some type option. It' mandatory to assign one and only one identifier for **environment_vip_id** and a string for **type_option**. String **type_option** is not case sensitive.

It is possible to specify in several ways fields desired to be retrieved using the above route through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for its route (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- tipo_opcao
- nome_opcao_txt

Example:

```
/api/v3/option-vip/environment-vip/1/type-option/balanceamento/
```

Response body:

```
{

    "optionsvip": [
        [{
            "id": <integer>,
            "tipo_opcao": <string>,
            "nome_opcao_txt": <string>
        },...]
    ]

}
```

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id and tipo_opcao:

```
fields=id,tipo_opcao
```

Using kind GET parameter

The above route also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "optionsvip": [{
        "id": <integer>,
        "tipo_opcao": <string>
    },...]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "optionsvip": [{
        "id": <integer>,
        "tipo_opcao": <string>,
        "nome_opcao_txt": <string>
    },...]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "optionsvip": [{
        "id": <integer>,
        "tipo_opcao": <string>
    },...]
}
```

Server Pool module

/api/v3/pool/

GET

Obtaining list of Server Pool

It is possible to specify in several ways fields desired to be retrieved in Server Pool module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Server Pool module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- identifier
- default_port
- *environment*
- **servicedownaction**
- lb_method
- **healthcheck**
- default_limit
- **server_pool_members**
    - id

        - *server_pool*
        - identifier
        - *ip*
        - *ipv6*
        - priority
        - weight
        - limit
        - port_real
        - member_status
        - last_status_update
        - last_status_update_formated
        - *equipments*
        - *equipment*
    - pool_created
    - *vips*
    - dscp
    - *groups_permissions*

Obtaining list of Server Pools through id's URL:

```
/api/v3/pool/<pool_ids>/
```

where **pool_ids** are the identifiers of each pool desired to be obtained. To obtain more than one pool, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/1/
```

Many IDs:

```
/api/v3/pool/1;3;8/
```

Obtaining list of Server Pools through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/pool/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/pool/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "environment": 1
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **search** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, identifier and pool_created:

```
fields=id,identifier,pool_created
```

Using kind GET parameter

The Server Pool module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "server_pools": [{
        "id": <integer>,
        "identifier": <string>,
        "pool_created": <boolean>
    },...]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "server_pools": [{
        "id": <integer>,
        "identifier": <string>,
        "default_port": <integer>,
        "environment": {
            "id": <integer>,
            "name": <string>
        },
        "servicedownaction": {
            "id": <integer>,
            "type": <string>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <integer>,
            "identifier": <string>,
            "ip": {
                "id": <integer>,
                "ip_formated": <string>
            },
            "ipv6": {
                "id": <integer>,
                "ip_formated": <string>
            },
            "priority": <integer>,
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "member_status": <integer>,
            "last_status_update_formated": <string>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            }
        }],
        "pool_created": <boolean>
    }]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "identifier": <string>,
        "default_port": <integer>,
        environmentvip": <environment_id>,
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "last_status_update_formated": <string>,
            "member_status": <integer>
        },...],
        "pool_created": <boolean>
    },...]
}
```

POST

Creating list of pools in database:

This only affects database, if flag **"created"** is assigned true, it will be ignored.

URL:

```
/api/v3/pool/
```

Request body:

```
{
    "server_pools": [{
        "id": <null>,
        "identifier": <string>,
        "default_port": <integer>,
        "environmentvip": <environment_id>,
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "last_status_update_formated": <string>,
            "member_status": <integer>
        }],
        "pool_created": <boolean>
    },...]
}
```

URL Example:

```
/api/v3/pool/
```

More information about the POST request can be obtained in:

```
/api/v3/help/pool_post/
```

PUT

Updating list of server pools in database

URL:

```
/api/v3/pool/<pool_ids>
```

where **pool_ids** are the identifiers of each pool desired to be updated. Only pools not deployed to equipments can be updated in this way. To update more than one pool, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/1/
```

Many IDs:

```
/api/v3/pool/1;3;8/
```

Request body:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "identifier": <string>,
        "default_port": <integer>,
        environmentvip": <environment_id>,
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
```

```
            "port_real": <integer>,
            "last_status_update_formated": <string>,
            "member_status": <integer>
        }],
        "pool_created": <boolean>
    },...]
}
```

More information about the PUT request can be obtained in:

```
/api/v3/help/pool_put/
```

DELETE

Deleting a list of server pools in database

URL:

```
/api/v3/pool/<pool_ids>/
```

where **pool_ids** are the identifiers of each pool desired to be deleted. To delete more than one pool, semicolons between the identifiers should be used. If at least one pool assigned to **pool_ids** exists in equipment, an exception will be raised.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/1/
```

Many IDs:

```
/api/v3/pool/1;3;8/
```

/api/v3/pool/deploy

GET

Obtaining list of pools with member states updated

URL:

```
/api/v3/pool/deploy/<pool_ids>/member/status/
```

where **pool_ids** are the identifiers of each pool desired to be obtained. To obtain more than one pool, semicolons between the identifiers should be used.

GET Param:

```
checkstatus=[0|1]
```

To obtain member states **updated**, checkstatus should be assigned to 1. If it is assigned to 0, server pools will be retrieved but the real status of the equipments will not be checked in the equipment.

Response body:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "identifier": <string>,
        "default_port": <integer>,
        environmentvip": <environment_id>,
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "last_status_update_formated": <string>,
            "member_status": <integer>
        }],
        "pool_created": <boolean>
    },...]
}
```

Pool Member

- **member_status** in "server_pool_members must receive a octal numeric value (0 to 7). This value will be converted into binary format with each bit representing one status. On PUT, most significant bit (2^2) will be ignored because it's read-only in the equipments.
- **member_status binary format: NNN where N is 0 or 1.**
  - **First bit (2^0):**
    - User up - 1 - new connections allowed, check second bit
    - User down - 0 - allow existing connections to time out, but no new connections are allowed, ignore second bit

- **Second bit (2^1):**
  - Enabled member - 1 - new connections allowed
  - Disabled member - 0 - member only process persistent and active connections
- **Third bit (read-only)(2^2):**
  - Healthcheck status is up - 1 - new connections allowed
  - Healthcheck status is down - 0 - no new connections are send in this state

POST

Creating list of pools in equipments

URL:

```
/api/v3/pool/deploy/<pool_ids>/
```

where **pool_ids** are the identifiers of each pool desired to be deployed. These pools must exist in database. To deploy more than one pool, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/1/
```

Many IDs:

```
/api/v3/pool/1;3;8/
```

PUT

Enabling/Disabling pool member by list of server pool

URL:

```
/api/v3/pool/deploy/<pool_ids>/member/status/
```

where **pool_ids** are the identifiers of each pool desired to be updated. To update more than one pool, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/deploy/1/member/status/
```

Many IDs:

```
/api/v3/pool/deploy/1;3;8/member/status/
```

Request body:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "server_pool_members": [{
```

```
            "id": <server_pool_member_id>,
            "member_status": <integer>
        }]
    },...]
}
```

More information about the PUT request can be obtained in:

```
/api/v3/help/pool_put/
```

Updating pools by list in equipments

URL:

```
/api/v3/pool/deploy/<pool_ids>/
```

Request body:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "identifier": <string>,
        "default_port": <integer>,
        environmentvip": <environment_id>,
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "last_status_update_formated": <string>,
```

```
            "member_status": <integer>
        }],
        "pool_created": <boolean>
    },...]
}
```

URL Example:

```
/api/v3/pool/
```

More information about the PUT request can be obtained in:

```
/api/v3/help/pool_put/
```

DELETE

Deleting a list of server pools in equipments

URL:

```
/api/v3/pool/deploy/<pool_ids>/
```

where **pool_ids** are the identifiers of each pool desired to be deleted only in equipment. In database these server pools will not be deleted, but only flag "created" of each server pool will be changed to "false". To delete more than one pool in equipment, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/deploy/1/
```

Many IDs:

```
/api/v3/pool/deploy/1;3;8/
```

/api/v3/pool/details

GET

Obtaining server pools with some more details through id's

URL:

```
/api/v3/pool/details/<pool_ids>/
```

where **pool_ids** are the identifiers of each pool desired to be obtained. To obtain more than one pool, semicolons between the identifiers should be used.

Example with Parameter IDs:

One ID:

```
/api/v3/pool/details/1/
```

Many IDs:

```
/api/v3/pool/details/1;3;8/
```

Response body:

```
{
    "server_pools": [{
        "id": <server_pool_id>,
        "identifier": <string>,
        "default_port": <integer>,
        "environmentvip": {
            "id": <environment_id>,
            "finalidade_txt": <string>,
            "cliente_txt": <string>,
            "ambiente_p44_txt": <string>,
            "description": <string>
        }
        "servicedownaction": {
            "id": <optionvip_id>,
            "name": <string>
        },
        "lb_method": <string>,
        "healthcheck": {
            "identifier": <string>,
            "healthcheck_type": <string>,
            "healthcheck_request": <string>,
            "healthcheck_expect": <string>,
            "destination": <string>
        },
        "default_limit": <integer>,
        "server_pool_members": [{
            "id": <server_pool_member_id>,
            "identifier": <string>,
            "ipv6": {
                "ip_formated": <ipv6_formated>,
                "id": <ipv6_id>
            },
            "ip": {
                "ip_formated": <ipv4_formated>,
                "id": <ipv4_id>
            },
            "priority": <integer>,
            "equipment": {
                "id": <integer>,
                "name": <string>
            },
            "weight": <integer>,
            "limit": <integer>,
            "port_real": <integer>,
            "last_status_update_formated": <string>,
            "member_status": <integer>
        }],
        "pool_created": <boolean>
    },...]
}
```

Obtaining server pools with some more details through extended search

URL:

```
/api/v3/pool/details/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/pool/details/?search=[dict encoded]
```

Request body:

```
{
    'extends_search': [{
        'environment': <environment_id>
    }],
    'start_record': <integer>,
    'custom_search': '<string>',
    'end_record': <integer>,
    'asorting_cols': [<string>,..],
    'searchable_columns': [<string>,..]
}
```

Request body example:

```
{
    'extends_search': [{
        'environment': 1
    }],
    'start_record': 0,
    'custom_search': 'pool_123',
    'end_record': 25,
    'asorting_cols': ['identifier'],
    'searchable_columns': [
        'identifier',
        'default_port',
        'pool_created',
        'healthcheck__healthcheck_type'
    ]
}
```

Response body:

```
{
    "total": <integer>,
    "server_pools": [...]
}
```

/api/v3/pool/environment-vip

GET

Obtaining server pools associated to environment vips

URL:

```
/api/v3/pool/environment-vip/<environment_vip_id>/
```

where **environment_vip_id** is the identifier of the environment vip used as an argument to retrieve associated server pools. It's mandatory to assign one and only one identifier to **environment_vip_id**. The instruction related to use of extra GET parameters (**kind**, **fields**, **include** and **exclude**) and the default response body is the same as described in *Server Pool GET Module*. The only difference is that **fields** GET parameter is always initialized by default with 'id' and 'identifier' fields.

Example:

```
/api/v3/pool/environment-vip/1/
```

/api/v3/pool/deploy/async/

POST

Deploying list of Server Pool asynchronously

URL:

```
/api/v3/pool/deploy/async/[pool_ids]/
```

You can also deploy Server Pool objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **pool_ids** are the identifiers of Server Pool objects desired to be deployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Server Pool desired to be deployed in response, but for each Server Pool you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Server Pool objects be deployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/pool/deploy/async/
```

URL Example with one identifier:

```
/api/v3/pool/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Deploying two Server Pool objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
```

```
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating and Redeploying list of Server Pool asynchronously

URL:

```
/api/v3/pool/deploy/async/[pool_ids]/
```

You can also update and redeploy Server Pool objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Server Pool Update and Redeploy*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Server Pool desired to be updated and redeployed in response, but for each Server Pool you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Server Pool objects be updated and redeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/pool/deploy/async/[pool_ids]/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Updating and Redeploying two Server Pool objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Undeploying list of Server Pool asynchronously

URL:

```
/api/v3/pool/deploy/async/[pool_ids]/
```

You can also undeploy Server Pool objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **pool_ids** are the identifiers of Server Pool objects desired to be undeployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Server Pool desired to be undeployed in response, but for each Server Pool you will receive an

identifier for the created task. Since this is an asynchronous request, it may be that Server Pool objects be undeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/pool/deploy/async/
```

URL Example with one identifier:

```
/api/v3/pool/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Undeploying two Server Pool objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

Type Option module

/api/v3/type-option/environment-vip

GET

Obtaining options vip through environment vip and type option

URL:

```
/api/v3/type-option/environment-vip/<environment_vip_id>/
```

where **environment_vip_id** are the identifiers of environment vips used as an argument to retrieve associated type options. It can use multiple id's separated by semicolons.

Example:

```
/api/v3/type-option/environment-vip/1/
```

Response body:

```
[
    [
        <string>,...
    ],...
]
```

Vip Request module

/api/v3/vip-request/

GET

Obtaining list of Vip Request

It is possible to specify in several ways fields desired to be retrieved in Vip Request module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Vip Request module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- name
- service
- business
- *environmentvip*
- *ipv4*
- *ipv6*
- *equipments*
- default_names
- dscp
- ports
- **options**
- *groups_permissions*
- created

Where:

- **"environmentvip"** attribute is an integer that identifies the environment vip associated to the retrieved vip request.
- **"options" are the configured options vip associated to the retrieved vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports associated to the retrieved vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in **"options"** discussed above.
  - **"server_pool"** is the identifier of the server-pool port associated to the retrieved vip request.

Obtaining list of Vip Request through id's URL:

```
/api/v3/vip-request/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vip-request/1/
```

Many IDs:

```
/api/v3/vip-request/1;3;8/
```

Obtaining list of Vip Request through extended search Extended search permits a search with multiple options, according with user desires. The following two examples are shown to demonstrate how easy is to use this resource. In the first example, **extended-search** attribute receives an array with two dicts where the expected result is a list of vip requests where the ipv4 "192.168.x.x" are created or the ipv4 "x.168.17.x" are not created in each associated server pools. Remember that an OR operation is made to each element in an array and an AND operation is made to each element in a dict. An array can be a value associated to some key into a dict as well as a dict can be an element of an array.

In the second example, **extended-search** attribute receives an array with only one dict where the expected result is a list of vip requests where the ipv4 "192.x.x.x" are created on each associated server pools and the name of each virtual lan associated with each ipv4 contains the word "G1". This is one of many possibilities offered by Django QuerySet API. Due to use of **icontains**, the search of "G1" is not case sensitive.

More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/vip-request/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/vip-request/?search=[encoded dict]
```

First request body example:

```
{
    "extends_search": [{
        "ipv4__oct1": "192",
        "ipv4__oct2": "168",
        "created": true
        },
    {
        "ipv4__oct2": "168",
        "ipv4__oct3": "17",
        "created": false
    }],
    "start_record": 0,
```

```
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

Second request body example:

```
{
    "extends_search": [{
        "ipv4__vlan__nome__icontains": "G1",
        "ipv4__oct1": "192",
        "created": true
        }
    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

URL encoded for first request body example:

```
/api/v3/vip-request/?search=%22%7B+++++%22extends_search%22%3A+%5B%7B+++++++++%22ipv4__oct1%22%3A+%22
```

URL encoded for second request body example:

```
/api/v3/vip-request/?search=%7B+++++++++%22extends_search%22%3A+%5B%7B++++++++++++%22ipv4__vlan__nor
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and created:

```
fields=id,name,created
```

Using kind GET parameter

The Vip Request module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field ipv4 for *basic* will contain only the identifier and for *details* will contain name, the ip formated and description.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "vips": [{
        "id": <integer>,
        "name": <string>,
        "ipv4": <integer>,
        "ipv6": <integer>
    }]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "vips": [{
        "id": <integer>,
        "name": <string>,
        "service": <string>,
        "business": <string>,
        "environmentvip": {
            "id": <integer>,
            "finalidade_txt": <string>,
            "cliente_txt": <string>,
            "ambiente_p44_txt": <string>,
            "description": <string>
        },
        "ipv4": {
            "id": <integer>,
            "ip_formated": <string>,
            "description": <string>
        },
        "ipv6": {
            "id": <integer>,
            "ip_formated": <string>,
            "description": <string>
        },
        "equipments": [{
            "id": <integer>,
            "name": <string>,
            "maintenance": <boolean>,
            "equipment_type": {
                "id": <integer>,
                "equipment_type": <string>
            },
            "model": {
                "id": <integer>,
                "name": <string>
            }
        },...],
        "default_names": [
            <string>,...
        ],
        "dscp": <integer>,
        "ports": [{
            "id": <integer>,
            "port": <integer>,
            "options": {
```

```
        "l4_protocol": {
            "id": <integer>,
            "tipo_opcao": <string>,
            "nome_opcao_txt": <string>
        },
        "l7_protocol": {
            "id": <integer>,
            "tipo_opcao": <string>,
            "nome_opcao_txt": <string>
        }
    },
    "pools": [{
        "id": <integer>,
        "server_pool": {
            "id": <integer>,
            "identifier": <string>,
            "default_port": <integer>,
            "environment": {
                "id": <integer>,
                "name": <string>
            },
            "servicedownaction": {
                "id": <integer>,
                "type": <string>,
                "name": <string>
            },
            "lb_method": <string>,
            "healthcheck": {
                "identifier": <string>,
                "healthcheck_type": <string>,
                "healthcheck_request": <string>,
                "healthcheck_expect": <string>,
                "destination": <string>
            },
            "default_limit": <integer>,
            "server_pool_members": [{
                "id": <integer>,
                                        "server_pool": <integer>,
                                        "identifier": <string>,
                                        "ip": {
                                                "id": <integer>,
                                                "ip_formated": <string>
                                        },
                                        "ipv6": {
                                                "id": <integer>,
                                                "ip_formated": <string>
                                        },
                                        "priority": <integer>,
                                        "weight": <integer>,
                                        "limit": <integer>,
                                        "port_real": <integer>,
                                        "member_status": <integer>,
                                        "last_status_update_formated": <string>,
                                        "equipment": {
                                                "id": <integer>,
                                                "name": <string>
                                        }
            },...],
```

```
                    "pool_created": <boolean>
                },
                "l7_rule": {
                    "id": <integer>,
                    "tipo_opcao": <string>,
                    "nome_opcao_txt": <string>
                },
                "l7_value": <integer>,
                "order": <integer>
            }]
        },...],
        "options": {
            "cache_group": {
                "id": <integer>,
                "tipo_opcao": <string>,
                "nome_opcao_txt": <string>
            },
            "traffic_return": {
                "id": <integer>,
                "tipo_opcao": <string>,
                "nome_opcao_txt": <string>
            },
            "timeout": {
                "id": <integer>,
                "tipo_opcao": <string>,
                "nome_opcao_txt": <string>
            },
            "persistence": {
                "id": <integer>,
                "tipo_opcao": <string>,
                "nome_opcao_txt": <string>
            }
        },
        "groups_permissions": [{
            "group": {
                "id": <integer>,
                "name": <string>
            },
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...],
        "created": <boolean>
    },...]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

```
{
    "vips": [{
        "id": <integer>,
        "name": <string>,
        "service": <string>,
        "business": <string>,
        "environmentvip": <integer>,
        "ipv4": <integer>,
        "ipv6": <integer>,
        "ports": [{
            "id": <integer>,
            "port": <integer>,
            "options": {
                "l4_protocol": <integer>,
                "l7_protocol": <integer>
            },
            "pools": [{
                "id": integer,
                "server_pool": <integer>,
                "l7_rule": <integer>,
                "l7_value": <integer>,
                "order": <integer>
            }, ...]
        }, ...],
        "options": {
            "cache_group": <integer>,
            "traffic_return": <integer>,
            "timeout": <integer>,
            "persistence": <integer>
        },
        "created": <boolean>
    },...]
}
```

POST

Creating list of vip request

URL:

```
/api/v3/vip-request/
```

Request body:

```
{
    "vips": [{
        "business": [string],
        "created": [boolean],
        "environmentvip": [environmentvip_id],
        "id": [null],
        "ipv4": [ipv4_id],
        "ipv6": [ipv6_id],
        "name": [string],
```

```
        "options": {
            "cache_group": [optionvip_id],
            "persistence": [optionvip_id],
            "timeout": [optionvip_id],
            "traffic_return": [optionvip_id]
        },
        "ports": [{
            "id": [vip_port_id],
            "options": {
                "l4_protocol": [optionvip_id],
                "l7_protocol": [optionvip_id]
            },
            "pools": [{
                    "l7_rule": [optionvip_id],
                    "l7_value": [string],
                    "order": [integer],
                    "server_pool": [server_pool_id]
                },..],
            "port": [integer]
            },..],
        "service": [string]
    },..]
}
```

- **"environmentvip"** attribute is an integer that identifies the environment vip that is desired to associate to the new vip request.
- **"options" are the configured options vip that is desired to associate to the new vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports that is desired to asssociate to the new vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in **"options"** discussed above.
  - **"server_pool"** is the identifier of the server-pool port associated to the new vip request.

URL Example:

```
/api/v3/vip-request/
```

More information about the POST request can be obtained in:

```
/api/v3/help/vip_request_post/
```

PUT

Updating list of vip request in database

URL:

```
/api/v3/vip-request/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vip-request/1/
```

Many IDs:

```
/api/v3/vip-request/1;3;8/
```

Request body:

```
{
    "vips": [{
        "business": [string],
        "created": [boolean],
        "environmentvip": [environmentvip_id],
        "id": [vip_id],
        "ipv4": [ipv4_id],
        "ipv6": [ipv6_id],
        "name": [string],
        "options": {
            "cache_group": [optionvip_id],
            "persistence": [optionvip_id],
            "timeout": [optionvip_id],
            "traffic_return": [optionvip_id]
        },
        "ports": [{
            "id": [vip_port_id],
            "options": {
                "l4_protocol": [optionvip_id],
                "l7_protocol": [optionvip_id]
            },
            "pools": [{
                    "l7_rule": [optionvip_id],
                    "l7_value": [string],
                    "order": [integer],
                    "server_pool": [server_pool_id]
                },..],
            "port": [integer]
            },..],
        "service": [string]
    },..]
}
```

- **"environmentvip"** attribute is an integer that identifies the environment vip that is desired to associate to the existent vip request.
- **"options" are the configured options vip that is desired to associate to the existent vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports that is desired to asssociate to the existent vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in **"options"** discussed above.
  - **"server_pool"** is the identifier of the server-pool port associated to the existent vip request.

URL Example:

```
/api/v3/vip-request/1/
```

More information about the PUT request can be obtained in:

```
/api/v3/help/vip_request_put/
```

DELETE

Deleting a list of vip request in database

Deleting list of vip-request and associated IP's URL:

```
/api/v3/vip-request/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests desired to delete. It can use multiple id's separated by semicolons. Doing this, the IP associated with each server pool desired to be deleted will also be deleted if this IP is not associated with any other vip request not contained in list of vip request that the user want to delete.

Example with Parameter IDs:

One ID:

```
/api/v3/vip-request/1/
```

Many IDs:

```
/api/v3/vip-request/1;3;8/
```

Deleting list of vip-request keeping associated IP's If desired to delete some vip-request keeping it's associated IP's, you must use an additional parameter in URL.

GET Param:

```
keepip=[0|1]
```

where:

- 1 - Keep IP in database
- 0 - Delete IP in database when it hasn't other relationship (the same as not use keepip parameter)

URL Examples:

```
/api/v3/vip-request/1/
```

With keepip parameter assigned to 1:

```
/api/v3/vip-request/1/?keepip=1
```

/api/v3/vip-request/deploy/

POST

Deploying list of vip request in equipments

URL:

```
/api/v3/vip-request/deploy/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests desired to be deployed. These selected vip requests must exist in the database. **vip_request_ids** can also be assigned to multiple id's separated by semicolons.

Examples:

One ID:

```
/api/v3/vip-request/deploy/1/
```

Many IDs:

```
/api/v3/vip-request/deploy/1;3;8/
```

PUT

Updating list of vip requests in equipments

URL:

```
/api/v3/vip-request/deploy/[vip_request_ids]
```

where **vip_request_ids** are the identifiers of vip requests desired to be updated. It can use multiple ids separated by semicolons.

Request body:

```
{
    "vips": [{
        "business": <string>,
        "created": <boolean>,
        "environmentvip": <environmentvip_id>,
        "id": <vip_id>,
        "ipv4": <ipv4_id>,
        "ipv6": <ipv6_id>,
        "name": <string>,
        "options": {
            "cache_group": <optionvip_id>,
            "persistence": <optionvip_id>,
            "timeout": <optionvip_id>,
            "traffic_return": <optionvip_id>
        },
        "ports": [{
            "id": <vip_port_id>,
            "options": {
                "l4_protocol": <optionvip_id>,
                "l7_protocol": <optionvip_id>
            },
            "pools": [{
                    "l7_rule": <optionvip_id>,
                    "l7_value": <string>,
                    "server_pool": <server_pool_id>
                },..],
            "port": <integer>
            },..],
        "service": <string>
    },..]
}
```

- "environmentvip" attribute is an integer that identifies the environment vip that is desired to associate to the existent vip request.
- **"options" are the configured options vip that is desired to associate to the existent vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports that is desired to asssociate to the existent vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in "options" discussed above.
  - "server_pool" is the identifier of the server-pool port associated to the existent vip request.

URL Example:

```
/api/v3/vip-request/1;3;8/
```

More information about the PUT request can be obtained in:

```
/api/v3/help/vip_request_put/
```

DELETE

Deleting list of vip requests in equipments

URL:

```
/api/v3/vip-request/deploy/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests desired to be deleted. It can use multiple id's separated by semicolons. Doing this, the IP associated with each server pool desired to be deleted will also be deleted if this IP is not associated with any other vip request not contained in list of vip request that the user want to delete.

Example with Parameter IDs:

One ID:

```
/api/v3/vip-request/deploy/1/
```

Many IDs:

```
/api/v3/vip-request/deploy/1;3;8/
```

/api/v3/vip-request/details/

GET

Obtaining list of vip request

Obtaining list of vip request with some more details through id's URL:

```
/api/v3/vip-request/details/[vip_request_ids]/
```

where **vip_request_ids** are the identifiers of vip requests desired to be retrieved with details. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vip-request/details/1/
```

Many IDs:

```
/api/v3/vip-request/details/1;3;8/
```

Response body:

```
{
    "vips": [{
        "id": (vip_id),
        "name": (string),
        "service": (string),
        "business": (string),
        "environmentvip": {
            "id": (environmentvip_id),
            "finalidade_txt": (string),
            "cliente_txt": (string),
            "ambiente_p44_txt": (string),
            "description": (string)
        },
        "ipv4": {
            "id": (ipv4_id)
            "ip_formated": (ipv4_formated),
            "description": (string)
        },
        "ipv6": null,
        "equipments": [{
            "id": (equipment_id),
            "name": (string),
            "equipment_type": (equipment_type_id),
            "model": (model_id),
            "groups": [(group_id),..]
        }],
        "default_names": [(string),..],
        "dscp": (vip_dscp_id),
        "ports": [{
            "id": (vip_port_id),
            "port": (integer),
            "options": {
                "l4_protocol": {
                    "id": (optionvip_id),
                    "tipo_opcao": (string),
                    "nome_opcao_txt": (string)
                },
                "l7_protocol": {
                    "id": (optionvip_id),
                    "tipo_opcao": (string),
                    "nome_opcao_txt": (string)
                }
            },
            "pools": [{
                "id": (vip_port_pool_id),
                "server_pool": {
                    'id': (server_pool_id),
                    ...information from the pool, same as GET Pool*
                },
                "l7_rule": {
```

```
                    "id": (optionvip_id),
                    "tipo_opcao": (string),
                    "nome_opcao_txt": (string)
                },
                "order": (integer|null),
                "l7_value": (string)
            },...]
        },...],
        "options": {
            "cache_group": {
                "id": (optionvip_id),
                "tipo_opcao": (string),
                "nome_opcao_txt": (string)
            },
            "traffic_return": {
                "id": (optionvip_id),
                "tipo_opcao": (string),
                "nome_opcao_txt": (string)
            },
            "timeout": {
                "id": (optionvip_id),
                "tipo_opcao": (string),
                "nome_opcao_txt": (string)
            },
            "persistence": {
                "id": (optionvip_id),
                "tipo_opcao": (string),
                "nome_opcao_txt": (string)
            }
        },
        "created": (boolean)
    },...]
}
```

- **"environmentvip"** attribute receives a dict with some information about the environment vip associated with the retrieved vip request.
- **"options" are the configured options vip associated to the retrieved vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports associated to the retrieved vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in **"options"** discussed above.
  - **"server_pool"** attribute receives a dict with some information about the server pool associated to the retrieved vip request.

Obtaining list of vip request with some more details through extended search Extended search permits a search with multiple options, according with user desires. The following two examples are shown to demonstrate how easy is to use this resource. In the first example, **extended-search** attribute receives an array with two dicts where the expected result is a list of vip requests where the ipv4 "192.168.x.x" are created or the ipv4 "x.168.17.x" are not created in each associated server pools. Remember that an OR operation is made to each element in an array and an AND operation is made to each element in a dict. An array can be a value associated to some key into a dict as well as a dict can be an element of an array.

In the second example, **extended-search** attribute receives an array with only one dict where the expected result is a list of vip requests where the ipv4 "192.x.x.x" are created on each associated server pools and the name of each virtual lan associated with each ipv4 contains the word "G1". This is one of many possibilities offered by Django QuerySet API. Due to use of **icontains**, the search of "G1" is not case sensitive.

More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/vip-request/details/
```

GET Param:

```
search=[encoded dict]
```

Example:

```
/api/v3/vip-request/details/?search=[encoded dict]
```

First request body example:

```
{
    "extends_search": [{
        "ipv4__oct1": "192",
        "ipv4__oct2": "168",
        "created": true
        },
    {
        "ipv4__oct2": "168",
        "ipv4__oct3": "17",
        "created": false
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

Second request body example:

```
{
    "extends_search": [{
        "ipv4__vlan__nome__icontains": "G1",
        "ipv4__oct1": "192",
        "created": true
        }
    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

URL encoded for first request body example:

```
/api/v3/vip-request/details/?search=%22%7B+++++%22extends_search%22%3A+%5B%7B+++++++++%22ipv4__oct1%2
```

URL encoded for second request body example:

```
/api/v3/vip-request/details/?search=%7B+++++++++%22extends_search%22%3A+%5B%7B++++++++++++%22ipv4__v
```

Response body:

```
{
    "total": [integer],
    "vips": [..]
}
```

- When **"search"** is used, "total" property is also retrieved.
- **"environmentvip"** attribute receives a dict with some information about the environment vip associated with the retrieved vip request.
- **"options" are the configured options vip associated to the retrieved vip request.**
  - cache-group, persistence, timeout and traffic_return are some values present in the database. These values are configured to a set of restricted values.
- **"ports" are the configured ports associated to the retrieved vip request.**
  - l4_protocol and l7_protocol in options and l7_rule in pools work as well as the values present in **"options"** discussed above.
  - **"server_pool"** attribute receives a dict with some information about the server pool associated to the retrieved vip request.

/api/v3/vip-request/deploy/async/

POST

Deploying list of Vip Request asynchronously

URL:

```
/api/v3/vip-request/deploy/async/[vip_request_ids]/
```

You can also deploy Vip Request objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **vip_request_ids** are the identifiers of Vip Request objects desired to be deployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Vip Request desired to be deployed in response, but for each Vip Request you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vip Request objects be deployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/vip-request/deploy/async/
```

URL Example with one identifier:

```
/api/v3/vip-request/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Deploying two Vip Request objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating and Redeploying list of Vip Request asynchronously

URL:

```
/api/v3/vip-request/deploy/async/[vip_request_ids]/
```

You can also update and redeploy Vip Request objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Vip Request Update and Redeploy*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Vip Request desired to be updated and redeployed in response, but for each Vip Request you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vip Request objects be updated and redeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/vip-request/deploy/async/[vip_request_ids]/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Updating and Redeploying two Vip Request objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Undeploying list of Vip Request asynchronously

URL:

```
/api/v3/vip-request/deploy/async/[vip_request_ids]/
```

You can also undeploy Vip Request objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **vip_request_ids** are the identifiers of Vip Request objects desired to be undeployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Vip Request desired to be undeployed in response, but for each Vip Request you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vip Request objects be undeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/vip-request/deploy/async/
```

URL Example with one identifier:

```
/api/v3/vip-request/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Undeploying two Vip Request objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

/api/v3/vip-request/pool

GET

Obtaining vip requests associated to server pool

URL:

```
/api/v3/vip-request/pool/<pool_id>/
```

where **pool_id** is the identifier of the server pool used as an argument to retrieve associated vip requests. Only one **pool_id** can be assigned. The instruction related to use of extra GET parameters (**kind**, **fields**, **include** and **exclude**) and the default response body is the same as described in *Vip Request GET Module*

Example:

```
/api/v3/vip-request/pool/1/
```

Vlan module

/api/v3/vlan

GET

Obtaining list of Vlans

It is possible to specify in several ways fields desired to be retrieved in Vlan module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Vlan module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- name
- num_vlan
- *environment*
- description
- acl_file_name
- acl_valida
- acl_file_name_v6
- acl_valida_v6
- active
- vrf
- acl_draft
- acl_draft_v6
- *networks_ipv4*
- *networks_ipv6*
- *vrfs*
- *groups_permissions*

Obtaining list of Vlans through id's URL:

```
/api/v3/vlan/[vlan_ids]/
```

where **vlan_ids** are the identifiers of Vlans desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vlan/1/
```

Many IDs:

```
/api/v3/vlan/1;3;8/
```

Obtaining list of Vlans through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/vlan/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/vlan/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "num_vlan": 1,
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and num_vlan:

```
fields=id,name,num_vlan
```

Using kind GET parameter

The Vlan module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "vlans": [{
        "id": <integer>,
        "name": <string>,
        "num_vlan": <integer>
    }]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "vlans": [{
        "id": <integer>,
        "name": <string>,
        "num_vlan": <integer>,
        "environment": {
            "id": <integer>,
            "name": <string>,
            "grupo_l3": {
                "id": <integer>,
                "name": <string>
            },
            "ambiente_logico": {
                "id": <integer>,
                "name": <string>
            },
            "divisao_dc": {
                "id": <integer>,
                "name": <string>
            },
            "filter": <integer>,
            "acl_path": <string>,
            "ipv4_template": <string>,
            "ipv6_template": <string>,
            "link": <string>,
            "min_num_vlan_1": <integer>,
            "max_num_vlan_1": <integer>,
            "min_num_vlan_2": <integer>,
            "max_num_vlan_2": <integer>,
            "default_vrf": {
                "id": <integer>,
                "internal_name": <string>,
                "vrf": <string>
            },
            "father_environment": <recurrence-to:environment>
        },
        "description": <string>,
        "acl_file_name": <string>,
        "acl_valida": <boolean>,
        "acl_file_name_v6": <string>,
        "acl_valida_v6": <boolean>,
```

```
        "active": <boolean>,
        "vrf": <string>,
        "acl_draft": <string>,
        "acl_draft_v6": <string>
    }]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "vlans": [{
        "id": <integer>,
        "name": <string>,
        "num_vlan": <integer>,
        "environment": <integer>,
        "description": <string>,
        "acl_file_name": <string>,
        "acl_valida": <boolean>,
        "acl_file_name_v6": <string>,
        "acl_valida_v6": <boolean>,
        "active": <boolean>,
        "vrf": <string>,
        "acl_draft": <string>,
        "acl_draft_v6": <string>
    },...]
}
```

POST

Creating list of vlans

URL:

```
/api/v3/vlan/
```

Request body:

```
{
    "vlans": [{
        "name": [string],
        "num_vlan": [integer],
        "environment": [environment_id:integer],
        "description": [string],
```

```
        "acl_file_name": [string],
        "acl_valida": [boolean],
        "acl_file_name_v6": [string],
        "acl_valida_v6": [boolean],
        "active": [boolean],
        "vrf": [string],
        "acl_draft": [string],
        "acl_draft_v6": [string],
        "create_networkv4": {
            "network_type": [network_type_id:integer],
            "environmentvip": [environmentvip_id:integer],
            "prefix": [integer]
        },
        "create_networkv6": {
            "network_type": [network_type_id:integer],
            "environmentvip": [environmentvip_id:integer],
            "prefix": [integer]
        }
    },..]
}
```

Request Example with only required fields:

```
{
    "vlans": [{
        "name": "Vlan for NetworkAPI",
        "environment": 5,
    }]
}
```

Request Example with some more fields:

```
{
    "vlans": [{
        "name": "Vlan for NetworkAPI",
        "num_vlan": 3,
        "environment": 5,
        "active": True,
        "create_networkv4": {
            "network_type": 6,
            "environmentvip": 2,
            "prefix": 24
        }
    }]
}
```

Through Vlan POST route you can create one or more Vlans. Only "name" and "environment" fields are required. You can specify other fields such as:

- **name** - As said, it will be Vlan name.
- **num_vlan** - You can specify manually the number of Vlan. However NetworkAPI can create it automatically for you.
- **environment** - You are required to associate Vlan with some environment.
- **acl_file_name** and **acl_file_name_v6** - You can give ACL names for associated NetworkIPv4 and NetworkIPv6.
- **acl_valida** and **acl_valida_v6** - If not specified ACLs will not be validated by default.
- **active** - If not specified, Vlan will be set to not active.

- **vrf** - Define in what VRF Vlan will be placed.
- **acl_draft** and **acl_draft_v6** - String to define acl draft.
- **create_networkv4 and create_networkv6 - Through these objects you can create NetworkIPv4 or NetworkIPv6 and autom**
  - **network_type** - You can specify the type of Network that is desired to create, but you are not required to do that.
  - **environmentvip** - You can associate Network with some Environment Vip, but you are not required to do that.
  - **prefix** - You are required to specify the prefix of Network. For NetworkIPv4 it ranges from 0 to 31 and for NetworkIPv6 it ranges from 0 to 127.

At the end of POST request, it will be returned the identifiers of new Vlans created.

Response Body:

```
[
    {
        "id": [integer]
    },...
]
```

Response Example for two Vlans created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/vlan/
```

PUT

Updating list of Vlans in database

URL:

```
/api/v3/vlan/[vlan_ids]/
```

where **vlan_ids** are the identifiers of Vlans. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vlan/1/
```

Many IDs:

```
/api/v3/vlan/1;3;8/
```

Request body:

```
{
    "vlans": [{
        "name": [string],
        "num_vlan": [integer],
        "environment": [environment_id:integer],
        "description": [string],
        "acl_file_name": [string],
        "acl_valida": [boolean],
        "acl_file_name_v6": [string],
        "acl_valida_v6": [boolean],
        "active": [boolean],
        "vrf": [string],
        "acl_draft": [string],
        "acl_draft_v6": [string]
    },..]
}
```

Request Example:

```
{
    "vlans": [{
        "id": 1,
        "name": "Vlan changed",
        "num_vlan": 4,
        "environment": 2,
        "description": "",
        "acl_file_name": "",
        "acl_valida": false ,
        "acl_file_name_v6": "",
        "acl_valida_v6": false,
        "active": false,
        "vrf": 'VrfBorda',
        "acl_draft": "",
        "acl_draft_v6": ""
    }]
}
```

In Vlan PUT request, you need to specify all fields even you don't want to change some of them.

- **id** - Identifier of Vlan that will be changed.
- **name** - As said, it will be Vlan name.
- **num_vlan** - You can specify manually the number of Vlan. However NetworkAPI can create it automatically for you.
- **environment** - You are required to associate Vlan with some environment.
- **acl_file_name** and **acl_file_name_v6** - You can give ACL names for associated NetworkIPv4 and NetworkIPv6.
- **acl_valida** and **acl_valida_v6** - If not specified ACLs will not be validated by default.
- **active** - If not specified, Vlan will be set to not active.
- **vrf** - Define in what VRF Vlan will be placed.
- **acl_draft** and **acl_draft_v6** - String to define acl draft.
- **create_networkv4 and create_networkv6 - Through these objects you can create NetworkIPv4 or NetworkIPv6 and autom**

- **network_type** - You can specify the type of Network that is desired to create, but you are not required to do that.
- **environmentvip** - You can associate Network with some Environment Vip, but you are not required to do that.
- **prefix** - You are required to specify the prefix of Network. For NetworkIPv4 it ranges from 0 to 31 and for NetworkIPv6 it ranges from 0 to 127.

URL Example:

```
/api/v3/vlan/1/
```

DELETE

Deleting a list of vlan in database

Deleting list of vlan and associated Networks and Object Group Permissions URL:

```
/api/v3/vlan/[vlan_ids]/
```

where **vlan_ids** are the identifiers of vlans desired to delete. It can use multiple id's separated by semicolons. Doing this, all NetworkIPv4 and NetworkIPv6 associated with Vlan desired to be deleted will be deleted too. All Object Group Permissions will also be deleted.

Example with Parameter IDs:

One ID:

```
/api/v3/vlan/1/
```

Many IDs:

```
/api/v3/vlan/1;3;8/
```

/api/v3/vlan/async/

POST

Creating list of vlans asynchronously

URL:

```
/api/v3/vlan/async/
```

You can also create Vlans asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Vlan Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Vlan desired to be created in response, but for each Vlan you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vlans have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/vlan/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Vlans:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of vlans asynchronously

URL:

```
/api/v3/vlan/async/
```

You can also update Vlans asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Vlan Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Vlan desired to be updated in response, but for each Vlan you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vlans have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/vlan/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Vlans:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of vlans asynchronously

URL:

```
/api/v3/vlan/async/
```

You can also delete Vlans asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous Vlan Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in the synchronous request, but for each Vlan you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Vlans have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/vlan/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Vlans:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

NetworkIPv4 module

/api/v3/networkv4/

GET

Obtaining list of Network IPv4 objects

It is possible to specify in several ways fields desired to be retrieved in Network IPv4 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Network IPv4 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- oct1
- oct2
- oct3

- oct4
- prefix
- networkv4
- mask_oct1
- mask_oct2
- mask_oct3
- mask_oct4
- mask_formated
- broadcast
- *vlan*
- **network_type**
- *environmentvip*
- active
- dhcprelay
- cluster_unit

Obtaining list of Network IPv4 objects through id's URL:

```
/api/v3/networkv4/[networkv4_ids]/
```

where **networkv4_ids** are the identifiers of Network IPv4 objects desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv4/1/
```

Many IDs:

```
/api/v3/networkv4/1;3;8/
```

Obtaining list of Network IPv4 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/networkv4/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/networkv4/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "oct1": 10,
        },
        {
            "oct1": 172,
        }

    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, networkv4 and mask_formated:

```
fields=id,networkv4,mask_formated
```

Using kind GET parameter

The Network IPv4 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "networks": [
        {
            "id": <integer>,
            "networkv4": <string>,
            "mask_formated": <string>,
            "broadcast": <string>,
            "vlan": {
                "id": <integer>,
                "name": <string>,
                "num_vlan": <integer>
            },
            "network_type": <integer>,
```

```
            "environmentvip": <integer>
        }
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "networks": [
        {
            "id": <integer>,
            "oct1": <integer>,
            "oct2": <integer>,
            "oct3": <integer>,
            "oct4": <integer>,
            "prefix": <integer>,
            "networkv4": <string>,
            "mask_oct1": <integer>,
            "mask_oct2": <integer>,
            "mask_oct3": <integer>,
            "mask_oct4": <integer>,
            "mask_formated": <string>,
            "broadcast": <string>,
            "vlan": {
                "id": <integer>,
                "name": <string>,
                "num_vlan": <integer>,
                "environment": <integer>,
                "description": <string>,
                "acl_file_name": <string>,
                "acl_valida": <boolean>,
                "acl_file_name_v6": <string>,
                "acl_valida_v6": <boolean>,
                "active": <boolean>,
                "vrf": <string>,
                "acl_draft": <string>,
                "acl_draft_v6": <string>
            },
            "network_type": {
                "id": <integer>,
                "tipo_rede": <string>
            },
            "environmentvip": {
                "id": <integer>,
                "finalidade_txt": <string>,
                "cliente_txt": <string>,
                "ambiente_p44_txt": <string>,
                "description": <string>
            },
            "active": <boolean>,
            "dhcprelay": [
                <string>, ...
            ],
            "cluster_unit": <string>
        }
```

```
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "networks": [
        {
            "id": <integer>,
            "oct1": <integer>,
            "oct2": <integer,
            "oct3": <integer>,
            "oct4": <integer>,
            "prefix": <integer>,
            "mask_oct1": <integer>,
            "mask_oct2": <integer>,
            "mask_oct3": <integer>,
            "mask_oct4": <integer>,
            "broadcast": <string>,
            "vlan": <integer>,
            "network_type": <integer>,
            "environmentvip": <integer>,
            "active": <boolean>,
            "cluster_unit": <string>
        }
    ]
}
```

POST

Creating list of IPv4 objects

URL:

```
/api/v3/networkv4/
```

Request body:

```
{
    "networks": [{
        "oct1": <integer>,
        "oct2": <integer>,
        "oct3": <integer>,
```

```
        "oct4": <integer>,
        "prefix": <integer>,
        "mask_oct1": <integer>,
        "mask_oct2": <integer>,
        "mask_oct3": <integer>,
        "mask_oct4": <integer>,
        "vlan": <integer>,
        "network_type": <integer>,
        "environmentvip": <integer>,
        "cluster_unit": <string>,
    },..]
}
```

Request Example with only required fields:

```
{
    "networks": [{
        "vlan": 10
    }]
}
```

Request Example with some more fields:

```
{
    "networks": [{
        "oct1": 10,
        "oct2": 0,
        "oct3": 0,
        "oct4": 0,
        "prefix": 24,
        "network_type": 5,
        "environmentvip": 5,
        "vlan": 5
    }]
}
```

Through Network IPv4 POST route you can create one or more Network IPv4 objects. Only "vlan" field are required. You can specify other fields such as:

- **oct1**, **oct2**, **oct3**, **oct4** - Are the octets of Network IPv4. Given an Vlan, API can provide automatically a Network IPv4 range to you, but it's possible to assign a Network IPv4 range respecting limits defined in Vlan. If you specify some octet, you need to specify all the others.
- **mask_oct1**, **mask_oct2**, **mask_oct3**, **mask_oct4** and **prefix** - If you specify octets of Network IPv4, it' mandatory to specify the mask by octets or by prefix.
- **network_type** - Says if it's a valid/invalid network of Vip Requests, Equipments or NAT.
- **environmentvip** - Use it to associate a new Network IPv4 to an existent Environment Vip

At the end of POST request, it will be returned the identifiers of new Network IPv4 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Network IPv4 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/networkv4/
```

PUT

Updating list of Network IPv4 objects in database

URL:

```
/api/v3/networkv4/[networkv4_ids]/
```

where **networkv4_ids** are the identifiers of Network IPv4 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv4/1/
```

Many IDs:

```
/api/v3/networkv4/1;3;8/
```

Request body:

```
{
    "networks": [{
        "id": <integer>,
        "network_type": <integer>,
        "environmentvip": <integer>,
        "cluster-unit": <string>
    },..]
}
```

Request Example:

```
{
    "networks": [{
        "id": 1,
        "network_type": 2,
        "environmentvip": 2,
        "cluster-unit": ""
    }]
}
```

In Network IPv4 PUT request, you can only change cluster-unit, environmentvip and network_type. If you don't provide at your request some of attributes below, this attribute will be changed to Null in database.

- **id** - Identifier of Network IPv4 that will be changed. It's mandatory.

- **network_type** - Says if it's a valid/invalid network of Vip Requests, Equipments or NAT.
- **environmentvip** - Use it to associate Network IPv4 to an existent Environment Vip.

URL Example:

```
/api/v3/networkv4/1/
```

DELETE

Deleting a list of Network IPv4 objects in database

Deleting list of Network IPv4 objects and associated IPv4 addresses URL:

```
/api/v3/networkv4/[networkv4_ids]/
```

where **networkv4_ids** are the identifiers of Network IPv4 objects desired to delete. It can use multiple id's separated by semicolons. Doing this, all IP addresses of Network IPv4 desired to be deleted will be also deleted. Remember that you can't delete Network IPv4 in database if it is deployed or if it exists Vip Request using some IP address of this Network IPv4.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv4/1/
```

Many IDs:

```
/api/v3/networkv4/1;3;8/
```

/api/v3/networkv4/deploy/

POST

Deploying list of Network IPv4 in equipments

URL:

```
/api/v3/networkv4/deploy/[networkv4_ids]/
```

where **networkv4_ids** are the identifiers of Network IPv4 desired to be deployed. These selected Network IPv4 objects must exist in the database. **networkv4_ids** can also be assigned to multiple id's separated by semicolons.

Examples:

One ID:

```
/api/v3/networkv4/deploy/1/
```

Many IDs:

```
/api/v3/networkv4/deploy/1;3;8/
```

DELETE

Undeploying list of Network IPv4 objects from equipments

URL:

```
/api/v3/networkv4/deploy/[networkv4_ids]/
```

where **networkv4_ids** are the identifiers of Network IPv4 objects desired to be undeployed from equipments. It can use multiple id's separated by semicolons. The undeployed Network IPv4 will continue existing in database as inactive.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv4/deploy/1/
```

Many IDs:

```
/api/v3/networkv4/deploy/1;3;8/
```

/api/v3/networkv4/async/

POST

Creating list of Network IPv4 asynchronously

URL:

```
/api/v3/networkv4/async/
```

You can also create Network IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Network IPv4 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv4 desired to be created in response, but for each Network IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv4 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
```

```
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of Network IPv4 asynchronously

URL:

```
/api/v3/networkv4/async/
```

You can also update Network IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Network IPv4 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv4 desired to be updated in response, but for each Network IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of Network IPv4 asynchronously

URL:

```
/api/v3/networkv4/async/
```

You can also delete Network IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous Network IPv4 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in

the synchronous request, but for each Network IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

/api/v3/networkv4/deploy/async/

POST

Deploying list of Network IPv4 asynchronously

URL:

```
/api/v3/networkv4/deploy/async/[networkv4_ids]/
```

You can also deploy Network IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **networkv4_ids** are the identifiers of Network IPv4 objects desired to be deployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv4 desired to be deployed in response, but for each Network IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv4 objects be deployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/networkv4/deploy/async/
```

URL Example with one identifier:

```
/api/v3/networkv4/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
```

```
    },...
]
```

Response Example for Deploying two Network IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Undeploying list of Network IPv4 asynchronously

URL:

```
/api/v3/networkv4/deploy/async/[networkv4_ids]/
```

You can also undeploy Network IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **networkv4_ids** are the identifiers of Network IPv4 objects desired to be undeployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv4 desired to be undeployed in response, but for each Network IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv4 objects be undeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/networkv4/deploy/async/
```

URL Example with one identifier:

```
/api/v3/networkv4/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Undeploying two Network IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

NetworkIPv6 module

/api/v3/networkv6/

GET

Obtaining list of Network IPv6 objects

It is possible to specify in several ways fields desired to be retrieved in Network IPv6 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Network IPv6 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- block1
- block2
- block3
- block4
- block5
- block6
- block7
- block8
- prefix
- networkv6
- mask1
- mask2
- mask3
- mask4
- mask5
- mask6
- mask7
- mask8
- mask_formated
- *vlan*
- **network_type**
- *environmentvip*
- active
- dhcprelay
- cluster_unit

Obtaining list of Network IPv6 objects through id's URL:

```
/api/v3/networkv6/[networkv6_ids]/
```

where **networkv6_ids** are the identifiers of Network IPv6 objects desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv6/1/
```

Many IDs:

```
/api/v3/networkv6/1;3;8/
```

Obtaining list of Network IPv6 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/networkv6/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/networkv6/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "block1": "fefe",
        },
        {
            "block1": "fdbe",
        }
    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, networkv6 and mask_formated:

```
fields=id,networkv6,mask_formated
```

Using kind GET parameter

The Network IPv6 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "networks": [
        {
            "id": <integer>,
            "networkv6": <string>,
            "mask_formated": <string>,
            "vlan": {
                "id": <integer>,
                "name": <string>,
                "num_vlan": <integer>
            },
            "network_type": <integer>,
            "environmentvip": <integer>
        }
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "networks": [
        {
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "prefix": <integer>,
            "networkv6": <string>,
            "mask1": <string>,
            "mask2": <string>,
            "mask3": <string>,
            "mask4": <string>,
```

```
            "mask5": <string>,
            "mask6": <string>,
            "mask7": <string>,
            "mask8": <string>,
            "mask_formated": <string>,
            "vlan": {
                "id": <integer>,
                "name": <string>,
                "num_vlan": <integer>,
                "environment": <integer>,
                "description": <string>,
                "acl_file_name": <string>,
                "acl_valida": <boolean>,
                "acl_file_name_v6": <string>,
                "acl_valida_v6": <boolean>,
                "active": <boolean>,
                "vrf": <string>,
                "acl_draft": <string>,
                "acl_draft_v6": <string>
            },
            "network_type": {
                "id": <integer>,
                "tipo_rede": <string>
            },
            "environmentvip": {
                "id": <integer>,
                "finalidade_txt": <string>,
                "cliente_txt": <string>,
                "ambiente_p44_txt": <string>,
                "description": <string>
            },
            "active": <boolean>,
            "dhcprelay": [
                <string>, ...
            ],
            "cluster_unit": <string>
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "networks": [
```

```
        {
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "prefix": <integer>,
            "mask1": <string>,
            "mask2": <string>,
            "mask3": <string>,
            "mask4": <string>,
            "mask5": <string>,
            "mask6": <string>,
            "mask7": <string>,
            "mask8": <string>,
            "vlan": <integer>,
            "network_type": <integer>,
            "environmentvip": <integer>,
            "active": <boolean>,
            "cluster_unit": <string>
        }
    ]
}
```

POST

Creating list of IPv6 objects

URL:

```
/api/v3/networkv6/
```

Request body:

```
{
    "networks": [{
        "block1": <string>,
        "block2": <string>,
        "block3": <string>,
        "block4": <string>,
        "block5": <string>,
        "block6": <string>,
        "block7": <string>,
        "block8": <string>,
        "prefix": <integer>,
        "mask1": <string>,
        "mask2": <string>,
        "mask3": <string>,
        "mask4": <string>,
        "mask5": <string>,
        "mask6": <string>,
        "mask7": <string>,
        "mask8": <string>,
```

```
        "vlan": <integer>,
        "network_type": <integer>,
        "environmentvip": <integer>,
        "cluster_unit": <string>,
    },..]
}
```

Request Example with only required fields:

```
{
    "networks": [{
        "vlan": 10
    }]
}
```

Request Example with some more fields:

```
{
    "networks": [{
        "block1": "fdbe",
        "block2": "bebe",
        "block3": "bebe",
        "block4": "bebe",
        "block5": "0000",
        "block6": "0000",
        "block7": "0000",
        "block8": "0000",
        "prefix": 64,
        "network_type": 5,
        "environmentvip": 5,
        "vlan": 5
    }]
}
```

Through Network IPv6 POST route you can create one or more Network IPv6 objects. Only "vlan" field are required. You can specify other fields such as:

- **block1**, **block2**, **block3**, **block4**, **block5**, **block6**, **block7**, **block8** - Are the octets of Network IPv6. Given an Vlan, API can provide automatically a Network IPv6 range to you, but it's possible to assign a Network IPv6 range respecting limits defined in Vlan. If you specify some octet, you need to specify all the others.
- **mask1**, **mask2**, **mask3**, **mask4**, **mask5**, **mask6**, **mask7**, **mask8** and **prefix** - If you specify octets of Network IPv6, it' mandatory to specify the mask by octets or by prefix.
- **network_type** - Says if it's a valid/invalid network of Vip Requests, Equipments or NAT.
- **environmentvip** - Use it to associate a new Network IPv6 to an existent Environment Vip

At the end of POST request, it will be returned the identifiers of new Network IPv6 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Network IPv6 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/networkv6/
```

PUT

Updating list of Network IPv6 objects in database

URL:

```
/api/v3/networkv6/[networkv6_ids]/
```

where **networkv6_ids** are the identifiers of Network IPv6 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv6/1/
```

Many IDs:

```
/api/v3/networkv6/1;3;8/
```

Request body:

```
{
    "networks": [{
        "id": <integer>,
        "network_type": <integer>,
        "environmentvip": <integer>,
        "cluster-unit": <string>
    },..]
}
```

Request Example:

```
{
    "networks": [{
        "id": 1,
        "network_type": 2,
        "environmentvip": 2,
        "cluster-unit": ""
    }]
}
```

In Network IPv6 PUT request, you can only change cluster-unit, environmentvip and network_type. If you don't provide at your request some of attributes below, this attribute will be changed to Null in database.

- **id** - Identifier of Network IPv6 that will be changed. It's mandatory.

- **network_type** - Says if it's a valid/invalid network of Vip Requests, Equipments or NAT.
- **environmentvip** - Use it to associate Network IPv6 to an existent Environment Vip.

URL Example:

```
/api/v3/networkv6/1/
```

DELETE

Deleting a list of Network IPv6 objects in database

Deleting list of Network IPv6 objects and associated IPv6 addresses URL:

```
/api/v3/networkv6/[networkv6_ids]/
```

where **networkv6_ids** are the identifiers of Network IPv6 objects desired to delete. It can use multiple id's separated by semicolons. Doing this, all IP addresses of Network IPv6 desired to be deleted will be also deleted. Remember that you can't delete Network IPv6 in database if it is deployed or if it exists Vip Request using some IP address of this Network IPv6.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv6/1/
```

Many IDs:

```
/api/v3/networkv6/1;3;8/
```

/api/v3/networkv6/deploy/

POST

Deploying list of Network IPv6 in equipments

URL:

```
/api/v3/networkv6/deploy/[networkv6_ids]/
```

where **networkv6_ids** are the identifiers of Network IPv6 desired to be deployed. These selected Network IPv6 objects must exist in the database. **networkv6_ids** can also be assigned to multiple id's separated by semicolons.

Examples:

One ID:

```
/api/v3/networkv6/deploy/1/
```

Many IDs:

```
/api/v3/networkv6/deploy/1;3;8/
```

DELETE

Undeploying list of Network IPv6 objects from equipments

URL:

```
/api/v3/networkv6/deploy/[networkv6_ids]/
```

where **networkv6_ids** are the identifiers of Network IPv6 objects desired to be undeployed from equipments. It can use multiple id's separated by semicolons. The undeployed Network IPv6 will continue existing in database as inactive.

Example with Parameter IDs:

One ID:

```
/api/v3/networkv6/deploy/1/
```

Many IDs:

```
/api/v3/networkv6/deploy/1;3;8/
```

/api/v3/networkv6/async/

POST

Creating list of Network IPv6 asynchronously

URL:

```
/api/v3/networkv6/async/
```

You can also create Network IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Network IPv6 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv6 desired to be created in response, but for each Network IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv6 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
```

```
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of Network IPv6 asynchronously

URL:

```
/api/v3/networkv6/async/
```

You can also update Network IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous Network IPv6 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv6 desired to be updated in response, but for each Network IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of Network IPv6 asynchronously

URL:

```
/api/v3/networkv6/async/
```

You can also delete Network IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous Network IPv6 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in

the synchronous request, but for each Network IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/networkv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two Network IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

/api/v3/networkv6/deploy/async/

POST

Deploying list of Network IPv6 asynchronously

URL:

```
/api/v3/networkv6/deploy/async/[networkv6_ids]/
```

You can also deploy Network IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **networkv6_ids** are the identifiers of Network IPv6 objects desired to be deployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv6 desired to be deployed in response, but for each Network IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv6 objects be deployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/networkv6/deploy/async/
```

URL Example with one identifier:

```
/api/v3/networkv6/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
```

```
    },...
]
```

Response Example for Deploying two Network IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Undeploying list of Network IPv6 asynchronously

URL:

```
/api/v3/networkv6/deploy/async/[networkv6_ids]/
```

You can also undeploy Network IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request, where **networkv6_ids** are the identifiers of Network IPv6 objects desired to be undeployed separated by commas. In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each Network IPv6 desired to be undeployed in response, but for each Network IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that Network IPv6 objects be undeployed after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example with one identifier:

```
/api/v3/networkv6/deploy/async/
```

URL Example with one identifier:

```
/api/v3/networkv6/deploy/async/1;3;8/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for Undeploying two Network IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

IPv4 module

/api/v3/ipv4/

GET

Obtaining list of IPv4 objects

It is possible to specify in several ways fields desired to be retrieved in IPv4 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for IPv4 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- ip_formated
- oct1
- oct2
- oct3
- oct4
- *networkipv4*
- description
- *equipments*
- *vips*
- **server_pool_members**
  - id
  - *server_pool*
  - identifier
  - *ip*
  - *ipv6*
  - priority
  - weight
  - limit
  - port_real
  - member_status
  - last_status_update
  - last_status_update_formated
  - *equipments*
  - *equipment*

Obtaining list of IPv4 objects through id's URL:

```
/api/v3/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of IPv4 objects desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv4/1/
```

Many IDs:

```
/api/v3/ipv4/1;3;8/
```

Obtaining list of IPv4 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/ipv4/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/ipv4/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "oct1": 10,
        },
        {
            "oct1": 172,
        }

    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, ip_formated and networkipv4:

```
fields=id,ip_formated,networkipv4
```

Using kind GET parameter

The IPv4 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

{

“ips”: [

{ “id”: <integer>, “ip_formated”: <string>, “networkipv4”: {

“id”: <integer>, “networkv4”: <string>, “mask_formated”: <string>, “broadcast”: <string>, “vlan”: {

“id”: <integer>, “name”: <string>, “num_vlan”: <integer>

}, “network_type”: <integer>, “environmentvip”: <integer>

}, “description”: <string>

}

]

}

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": {
                "id": <integer>,
                "oct1": <integer>,
                "oct2": <integer>,
                "oct3": <integer>,
                "oct4": <integer>,
```

```
                "prefix": <integer>,
                "networkv4": <string>,
                "mask_oct1": <integer>,
                "mask_oct2": <integer>,
                "mask_oct3": <integer>,
                "mask_oct4": <integer>,
                "mask_formated": <string>,
                "broadcast": <string>,
                "vlan": {
                    "id": <integer>,
                    "name": <string>,
                    "num_vlan": <integer>,
                    "environment": <integer>,
                    "description": <string>,
                    "acl_file_name": <string>,
                    "acl_valida": <boolean>,
                    "acl_file_name_v6": <string>,
                    "acl_valida_v6": <boolean>,
                    "active": <boolean>,
                    "vrf": <string>,
                    "acl_draft": <string>,
                    "acl_draft_v6": <string>
                },
                "network_type": {
                    "id": <integer>,
                    "tipo_rede": <string>
                },
                "environmentvip": {
                    "id": <integer>,
                    "finalidade_txt": <string>,
                    "cliente_txt": <string>,
                    "ambiente_p44_txt": <string>,
                    "description": <string>
                },
                "active": <boolean>,
                "dhcprelay": [
                    <string>,...
                ],
                "cluster_unit": <string>
            },
            "description": <string>
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ips":[
        {
            "id": <integer>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": <integer>,
            "description": <string>
        }
    ]
}
```

POST

Creating list of IPv4 objects

URL:

```
/api/v3/ipv4/
```

Request body:

```
{
    "ips": [{
        "oct1": <integer>,
        "oct2": <integer>,
        "oct3": <integer>,
        "oct4": <integer>,
        "networkipv4": <integer>,
        "description": <string>,
        "equipments": [
            {
                "id": <integer>
            },...
        ]
    },..]
}
```

Request Example with only required fields:

```
{
    "ips": [{
        "networkipv4": 10
    }]
}
```

Request Example with some more fields:

```
{
    "ips": [{
        "oct1": 10,
        "oct2": 10,
        "oct3": 0,
        "oct4": 20,
```

```
        "networkipv4": 2,
        "equipments": [
            {
                "id": 3
            },
            {
                "id": 4
            }
        ]
    }]
}
```

Through IPv4 POST route you can create one or more IPv4 objects. Only "networkipv4" field are required. You can specify other fields such as:

- **oct1**, **oct2**, **oct3**, **oct4** - Are the octets of IPv4. Given a network, API can provide to you an IPv4 Address automatically, but you can assign a IPv4 Address in a manually way. If you specify some octet, you need to specify all the others.
- **description** - Description of new IPv4.
- **networkipv4** - This parameter is mandatory. It is the network to which new IP address will belong.
- **equipments** - You can associate new IP address to one or more equipments.

At the end of POST request, it will be returned the identifiers of new IPv4 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two IPv4 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/ipv4/
```

PUT

Updating list of IPv4 objects in database

URL:

```
/api/v3/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of IPv4 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv4/1/
```

Many IDs:

```
/api/v3/ipv4/1;3;8/
```

Request body:

```
{
    "ips": [{
        "id": <integer>,
        "description": <string>,
        "equipments": [
            {
                "id": <integer>
            },...
        ]
    },..]
}
```

Request Example:

```
{
    "ips": [{
        "id": 1,
        "description": "New description",
        "equipments": [
            {
                "id": 5
            },
            {
                "id": 6
            }
        ]
    }]
}
```

In IPv4 PUT request, you can only change description and associations with equipments.

- **id** - Identifier of IPv4 that will be changed. It's mandatory.
- **description** - Description of new IPv4.
- **equipments** - You can create new associations with equipments when updating IPv4. Old associations will be deleted even you don't specify new associations to other equipments.

URL Example:

```
/api/v3/ipv4/1/
```

DELETE

Deleting a list of IPv4 objects in database

Deleting list of IPv4 objects and associated Vip Requests and relationships with Equipments URL:

```
/api/v3/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of ipv4s desired to delete. It can use multiple id's separated by semicolons. Doing this, all Vip Request associated with IPv4 desired to be deleted will be deleted too. All associations made to equipments will also be deleted.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv4/1/
```

Many IDs:

```
/api/v3/ipv4/1;3;8/
```

/api/v3/ipv4/async/

POST

Creating list of IPv4 asynchronously

URL:

```
/api/v3/ipv4/async/
```

You can also create IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv4 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv4 desired to be created in response, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of IPv4 asynchronously

URL:

```
/api/v3/ipv4/async/
```

You can also update IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv4 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv4 desired to be updated in response, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of IPv4 asynchronously

URL:

```
/api/v3/ipv4/async/
```

You can also delete IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous IPv4 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in the synchronous request, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

IPv6 module

/api/v3/ipv6/

GET

Obtaining list of IPv6 objects

It is possible to specify in several ways fields desired to be retrieved in IPv6 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for IPv6 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- ip_formated
- block1
- block2
- block3
- block4
- block5
- block6
- block7
- block8
- *networkipv6*
- description
- *equipments*

- *vips*
- **server_pool_members**
  - id
  - *server_pool*
  - identifier
  - *ip*
  - *ipv6*
  - priority
  - weight
  - limit
  - port_real
  - member_status
  - last_status_update
  - last_status_update_formated
  - *equipments*
  - *equipment*

Obtaining list of IPv6 objects through id's URL:

```
/api/v3/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of IPv6 objects desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv6/1/
```

Many IDs:

```
/api/v3/ipv6/1;3;8/
```

Obtaining list of IPv6 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/ipv6/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/ipv6/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "block1": "fefe",
        },
        {
            "block1": "fdfd",
        }

    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, ip_formated and networkipv6:

```
fields=id,ip_formated,networkipv6
```

Using kind GET parameter

The IPv6 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

{

"ips": [

{ "id": <integer>, "ip_formated": <string>, "networkipv6": {

"id": <integer>, "networkv6": <string>, "mask_formated": <string>, "broadcast": <string>, "vlan": {

"id": <integer>, "name": <string>, "num_vlan": <integer>

}, “network_type”: <integer>, “environmentvip”: <integer>

}, “description”: <string>

}

]

}

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "networkipv6": {
                "id": <integer>,
                "block1": <string>,
                "block2": <string>,
                "block3": <string>,
                "block4": <string>,
                "block5": <string>,
                "block6": <string>,
                "block7": <string>,
                "block8": <string>,
                "prefix": <integer>,
                "networkv6": <string>,
                "mask1": <string>,
                "mask2": <string>,
                "mask3": <string>,
                "mask4": <string>,
                "mask5": <string>,
                "mask6": <string>,
                "mask7": <string>,
                "mask8": <string>,
                "mask_formated": <string>,
                "vlan": {
                    "id": <integer>,
                    "name": <string>,
                    "num_vlan": <integer>,
                    "environment": <integer>,
                    "description": <string>,
                    "acl_file_name": <string>,
                    "acl_valida": <boolean>,
                    "acl_file_name_v6": <string>,
                    "acl_valida_v6": <boolean>,
                    "active": <boolean>,
```

```
                    "vrf": <string>,
                    "acl_draft": <string>,
                    "acl_draft_v6": <string>
                },
                "network_type": {
                    "id": <integer>,
                    "tipo_rede": <string>
                },
                "environmentvip": {
                    "id": <integer>,
                    "finalidade_txt": <string>,
                    "cliente_txt": <string>,
                    "ambiente_p44_txt": <string>,
                    "description": <string>
                },
                "active": <boolean>,
                "dhcprelay": [
                    <string>,...
                ],
                "cluster_unit": <string>
            },
            "description": <string>
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ips":[
        {
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "networkipv6": <integer>,
            "description": <string>
        }
```

```
    ]
}
```

POST

Creating list of IPv6 objects

URL:

```
/api/v3/ipv6/
```

Request body:

```
{
    "ips": [{
        "block1": <string>,
        "block2": <string>,
        "block3": <string>,
        "block4": <string>,
        "block5": <string>,
        "block6": <string>,
        "block7": <string>,
        "block8": <string>,
        "networkipv6": <integer>,
        "description": <string>,
        "equipments": [
            {
                "id": <integer>
            },...
        ]
    },..]
}
```

Request Example with only required fields:

```
{
    "ips": [{
        "networkipv6": 10
    }]
}
```

Request Example with some more fields:

```
{
    "ips": [{
        "block1": "fdbe",
        "block2": "fdbe",
        "block3": "0000",
        "block4": "0000",
        "block5": "0000",
        "block6": "0000",
        "block7": "0000",
        "block8": "0000",
        "networkipv6": 2,
        "equipments": [
            {
                "id": 3
```

```
            },
            {
                "id": 4
            }
        ]
    }]
}
```

Through IPv6 POST route you can create one or more IPv6 objects. Only "networkipv6" field are required. You can specify other fields such as:

- **block1**, **block2**, **block3**, **block4**, **block5**, **block6**, **block7** and **block8** - Are the octets of IPv6. Given a network, API can provide to you an IPv6 Address automatically, but you can assign a IPv6 Address in a manually way. If you specify some octet, you need to specify all the others.
- **networkipv6** - This parameter is mandatory. It is the network to which new IP address will belong.
- **description** - Description of new IPv6.
- **equipments** - You can associate new IP address to one or more equipments.

At the end of POST request, it will be returned the identifiers of new IPv6 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two IPv6 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/ipv6/
```

PUT

Updating list of IPv6 objects in database

URL:

```
/api/v3/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of IPv6 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv6/1/
```

Many IDs:

```
/api/v3/ipv6/1;3;8/
```

Request body:

```
{
    "ips": [{
        "id": <integer>,
        "description": <string>,
        "equipments": [
            {
                "id": <integer>
            },...
        ]
    },..]
}
```

Request Example:

```
{
    "ips": [{
        "id": 1,
        "description": "New description",
        "equipments": [
            {
                "id": 5
            },
            {
                "id": 6
            }
        ]
    }]
}
```

In IPv6 PUT request, you can only change description and associations with equipments.

- **id** - Identifier of IPv6 that will be changed. It's mandatory.
- **description** - Description of new IPv6.
- **equipments** - You can create new associations with equipments when updating IPv6. Old associations will be deleted even you don't specify new associations to other equipments.

URL Example:

```
/api/v3/ipv6/1/
```

DELETE

Deleting a list of IPv6 objects in database

Deleting list of IPv6 objects and associated Vip Requests and relationships with Equipments URL:

```
/api/v3/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of ipv6s desired to delete. It can use multiple id's separated by semicolons. Doing this, all Vip Request associated with IPv6 desired to be deleted will be deleted too. All associations made to equipments will also be deleted.

Example with Parameter IDs:

One ID:

```
/api/v3/ipv6/1/
```

Many IDs:

```
/api/v3/ipv6/1;3;8/
```

/api/v3/ipv6/async/

POST

Creating list of IPv6 asynchronously

URL:

```
/api/v3/ipv6/async/
```

You can also create IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv6 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv6 desired to be created in response, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of IPv6 asynchronously

URL:

```
/api/v3/ipv6/async/
```

You can also update IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv6 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv6 desired to be updated in response, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of IPv6 asynchronously

URL:

```
/api/v3/ipv6/async/
```

You can also delete IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous IPv6 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in the synchronous request, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v3/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

Object Group Permissions module

/api/v3/object-group-perm/

GET

Obtaining list of Object Group Permissions

It is possible to specify in several ways fields desired to be retrieved in Object Group Permission module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Object Group Permission module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- user_group
- object_type
- object_value
- read
- write
- change_config
- delete

Obtaining list of Object Group Permissions through id's URL:

```
/api/v3/object-group-perm/[object_group_perm_ids]/
```

where **object_group_perm_ids** are the identifiers of Object Group Permissions desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/object-group-perm/1/
```

Many IDs:

```
/api/v3/object-group-perm/1;3;8/
```

Obtaining list of Object Group Permissions through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/object-group-perm/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/object-group-perm/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "read": true
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, object_type and read:

```
fields=id,object_type,read
```

Using kind GET parameter

The Object Group Permission module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "ogps": [
        {
            "user_group": <integer>,
            "object_type": <integer>,
            "object_value": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ogps": [
        {
            "user_group": <integer>,
            "object_type": <integer>,
            "object_value": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ogps": [
        {
            "user_group": <integer>,
```

```
            "object_type": <integer>,
            "object_value": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
    ]
}
```

POST

Creating list of Object Group Permissions objects

URL:

```
/api/v3/object-group-perm/
```

Request body:

```
{
    "ogps": [{
        "user_group": <integer>,
        "object_type": <integer>,
        "object_value": <integer>,
        "read": <boolean>,
        "write": <boolean>,
        "change_config": <boolean>,
        "delete": <boolean>
    },..]
}
```

Request Example:

```
{
    "ogps": [{
        "user_group": 5,
        "object_type": 3,
        "object_value": 10,
        "read": true,
        "write": false,
        "change_config": false,
        "delete": false
    }]
}
```

Through Object Group Permissions POST route you can assign permissions for individual objects to some user group. Remember that individual permissions always prevail over general if it exists. All fields are required:

- **user_group** - It receives the identifier of some user group.
- **object_type** - It receives the identifier of some object type.
- **object_value** - It receives the identifier of some object value.
- **read** - Tell if the users of group identified by **user_group** will have read rights about specific object identified by **object_value** and by its type identified by **object_type**.

- **write** - Tell if the users of group identified by **user_group** will have write rights about specific object identified by **object_value** and by its type identified by **object_type**.
- **change_config** - Tell if the users of group identified by **user_group** will have change config rights about specific object identified by **object_value** and by its type identified by **object_type**.
- **delete** - Tell if the users of group identified by **user_group** will have delete rights about specific object identified by **object_value** and by its type identified by **object_type**.

At the end of POST request, it will be returned the identifiers of new Object Group Permissions objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Object Group Permissions objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/object-group-perm/
```

PUT

Updating list of Object Group Permissions objects

URL:

```
/api/v3/object-group-perm/
```

Request body:

```
{
    "ogps": [{
        "id": <integer>,
        "read": <boolean>,
        "write": <boolean>,
        "change_config": <boolean>,
        "delete": <boolean>
    },..]
}
```

Request Example:

```
{
    "ogps": [{
        "id": 5,
        "read": true,
```

```
        "write": false,
        "change_config": false,
        "delete": false
    }]
}
```

Through Object Group Permissions PUT route you can change permissions assigned for individual objects to some user group. Remember that individual permissions always prevail over general if it exists. Only **id** is required:

- **id** - Its the identifier fo the individual permission.
- **read** - Tell if the users of group identified by **user_group** will have read rights about specific object identified by **object_value** and by its type identified by **object_type**.
- **write** - Tell if the users of group identified by **user_group** will have write rights about specific object identified by **object_value** and by its type identified by **object_type**.
- **change_config** - Tell if the users of group identified by **user_group** will have change config rights about specific object identified by **object_value** and by its type identified by **object_type**.
- **delete** - Tell if the users of group identified by **user_group** will have delete rights about specific object identified by **object_value** and by its type identified by **object_type**.

At the end of PUT request, it will be returned the identifiers of Object Group Permissions objects updated.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Object Group Permissions objects updated:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/object-group-perm/
```

DELETE

Deleting a list of Object Group Permissions objects in database

Deleting list of Object Group Permissions objects URL:

```
/api/v3/object-group-perm/[object_group_perm_ids]/
```

where **object_group_perm_ids** are the identifiers of Object Group Permissions desired to delete. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/object-group-perm/1/
```

Many IDs:

```
/api/v3/object-group-perm/1;3;8/
```

General Object Group Permissions module

/api/v3/object-group-perm-general/

GET

Obtaining list of General Object Group Permissions

It is possible to specify in several ways fields desired to be retrieved in General Object Group Permission module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for General Object Group Permission module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- user_group
- object_type
- read
- write
- change_config
- delete

Obtaining list of General Object Group Permissions through id's URL:

```
/api/v3/object-group-perm-general/[object_group_perm_general_ids]/
```

where **object_group_perm_general_ids** are the identifiers of General Object Group Permissions desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/object-group-perm-general/1/
```

Many IDs:

```
/api/v3/object-group-perm-general/1;3;8/
```

Obtaining list of General Object Group Permissions through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/object-group-perm-general/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/object-group-perm-general/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "read": true,
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, read and write:

```
fields=id,read,write
```

Using kind GET parameter

The General Object Group Permission module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "ogpgs": [
        {
```

```
            "id": <integer>,
            "user_group": <integer>,
            "object_type": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ogpgs": [
        {
            "id": <integer>,
            "user_group": <integer>,
            "object_type": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ogpgs": [
        {
            "id": <integer>,
            "user_group": <integer>,
            "object_type": <integer>,
            "read": <boolean>,
            "write": <boolean>,
            "change_config": <boolean>,
            "delete": <boolean>
        },...
```

```
    ]
}
```

POST

Creating list of General Object Group Permissions objects

URL:

```
/api/v3/object-group-perm-general/
```

Request body:

```
{
    "ogpgs": [{
        "user_group": <integer>,
        "object_type": <integer>,
        "read": <boolean>,
        "write": <boolean>,
        "change_config": <boolean>,
        "delete": <boolean>
    },..]
}
```

Request Example:

```
{
    "ogpgs": [{
        "user_group": 5,
        "object_type": 3
        "read": true,
        "write": false,
        "change_config": false,
        "delete": false
    }]
}
```

Through General Object Group Permissions POST route you can assign permissions for a class of objects to some user group. Remember that general permissions do not prevail over individual if it exists. All fields are required:

- **user_group** - It receives the identifier of some user group.
- **object_type** - It receives the identifier of some object type.
- **read** - Tell if the users of group identified by **user_group** will have read rights about objects of type identified by **object_type**.
- **write** - Tell if the users of group identified by **user_group** will have write rights about objects of type identified by **object_type**.
- **change_config** - Tell if the users of group identified by **user_group** will have change config rights about objects of type identified by **object_type**.
- **delete** - Tell if the users of group identified by **user_group** will have delete rights about objects of type identified by **object_type**.

At the end of POST request, it will be returned the identifiers of new General Object Group Permissions objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two General Object Group Permissions objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/object-group-perm-general/
```

PUT

Updating list of General Object Group Permissions objects

URL:

```
/api/v3/object-group-perm-general/
```

Request body:

```
{
    "ogpgs": [{
        "user_group": <integer>,
        "object_type": <integer>,
        "read": <boolean>,
        "write": <boolean>,
        "change_config": <boolean>,
        "delete": <boolean>
    },..]
}
```

Request Example:

```
{
    "ogpgs": [{
        "user_group": 5,
        "object_type": 3
        "read": true,
        "write": false,
        "change_config": false,
        "delete": false
    }]
}
```

Through General Object Group Permissions PUT route you can change permissions assigned for a class of objects to some user group. Remember that general permissions do not prevail over individual if it exists. Only **id** is required:

- **id** - Its the identifier fo the general permission.
- **read** - Tell if the users of group identified by **user_group** will have read rights about objects of type identified by **object_type**.
- **write** - Tell if the users of group identified by **user_group** will have write rights about objects of type identified by **object_type**.
- **change_config** - Tell if the users of group identified by **user_group** will have change config rights about objects of type identified by **object_type**.
- **delete** - Tell if the users of group identified by **user_group** will have delete rights about objects of type identified by **object_type**.

At the end of PUT request, it will be returned the identifiers of General Object Group Permissions objects updated.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two General Object Group Permissions objects updated:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/object-group-perm-general/
```

DELETE

Deleting a list of General Object Group Permissions objects in database

Deleting list of General Object Group Permissions objects URL:

```
/api/v3/object-group-perm-general/[object_group_perm_general_ids]/
```

where **object_group_perm_general_ids** are the identifiers of General Object Group Permissions desired to delete. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/object-group-perm-general/1/
```

Many IDs:

```
/api/v3/object-group-perm-general/1;3;8/
```

Object Type module

/api/v3/object-type/

GET

Obtaining list of Object Types

It is possible to specify in several ways fields desired to be retrieved in Object Type module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Object Type module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- name

Obtaining list of Object Types through id's URL:

```
/api/v3/object-type/[object_type_ids]/
```

where **object_type_ids** are the identifiers of Object Types desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/object-type/1/
```

Many IDs:

```
/api/v3/object-type/1;3;8/
```

Obtaining list of Object Types through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/object-type/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/object-type/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "name": "Vrf",
```

```
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id and name:

```
fields=id,name
```

Using kind GET parameter

The Object Type module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "ots": [
        {
            "id": <integer>,
            "name": <string>
        },...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ots": [
        {
            "id": <integer>,
            "name": <string>
        },...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ots": [
        {
            "id": <integer>,
            "name": <string>
        },...
    ]
}
```

Vrf module

/api/v3/vrf/

GET

Obtaining list of Vrfs

It is possible to specify in several ways fields desired to be retrieved in Vrf module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Vrf module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation):

- id
- internal_name
- vrf

Obtaining list of Vrfs through id's URL:

```
/api/v3/vrf/[vrf_ids]/
```

where **vrf_ids** are the identifiers of Vrfs desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v3/vrf/1/
```

Many IDs:

```
/api/v3/vrf/1;3;8/
```

Obtaining list of Vrfs through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v3/vrf/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v3/vrf/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "vrf__contains": "Default",
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id and internal_name:

```
fields=id,internal_name
```

Using kind GET parameter

The Vrf module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "vrfs": [
        {
            "id": <integer>,
            "internal_name": <string>,
            "vrf": <string>
        },...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "vrfs": [
        {
            "id": <integer>,
            "internal_name": <string>,
            "vrf": <string>
        },...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "vrfs": [
        {
            "id": <integer>,
            "internal_name": <string>,
            "vrf": <string>
        },...
    ]
}
```

POST

Creating list of Vrf objects

URL:

```
/api/v3/vrf/
```

Request body:

```
{
    "vrfs": [{
        "vrf": <string>,
        "internal_name": <string>
    },..]
}
```

Request Example:

```
{
    "vrfs": [{
        "vrf": "BEVrf",
        "internal_name": "BEVrf"
    }]
}
```

Through Vrf POST route you can create one or more Vrf objects. All fields are required:

- **vrf**, **internal_name** - Are the names that represent the Vrf.

At the end of POST request, it will be returned the identifiers of new Vrf objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Vrf objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/vrf/
```

PUT

Updating list of Vrf objects

URL:

```
/api/v3/vrf/
```

Request body:

```
{
    "vrfs": [{
        "id": <integer>,
        "vrf": <string>,
        "internal_name": <string>
    },..]
}
```

Request Example:

```
{
    "vrfs": [{
        "id": 1,
        "vrf": "BEVrf",
        "internal_name": "BEVrf"
    }]
}
```

Through Vrf PUT route you can update one or more Vrf objects. All fields are required:

- **id** - Identifier of Vrf desired to update.
- **vrf**, **internal_name** - Are the names that represent the Vrf.

At the end of PUT request, it will be returned the identifiers of Vrf objects update.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two Vrf objects updated:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v3/vrf/
```

DELETE

Deleting a list of Vrf objects in database

Deleting list of Vrf objects and relationships with Equipments URL:

```
/api/v3/vrf/[vrf_ids]/
```

where **vrf_ids** are the identifiers of Vrf's desired to delete. It can use multiple id's separated by semicolons. Doing this, all associations made to equipments will also be deleted. You can't delete Vrf if it's used at some Environment or have relationship with Vlan and Equipment at same time.

Example with Parameter IDs:

One ID:

```
/api/v3/vrf/1/
```

Many IDs:

```
/api/v3/vrf/1;3;8/
```

Task module

/api/v3/task/

GET

How can I know the state of an asynchronous request?

URL:

```
/api/v3/task/[task_id]/
```

where **task_id** is the generated identifier for some asynchronous task. This route only accepts one **task_id** at a time.

Example with Parameter ID:

```
/api/v3/task/f8bb9ecf-ff40-4070-b379-6dcad7c8488a/
```

A task can assume the five status listed below. One way to track progress of some task is pooling NetworkAPI through this route. Once the task reaches SUCCESS, FAILURE or REVOKED status, you can stop to pooling NetworkAPI because your task have finished:

- PENDING - The task not yet run or status is unknown.
- SUCCESS - The task finished successfully.
- PROGRESS - The task is currently running.
- FAILURE - The job have failed.
- REVOKED - The job was cancelled (e.g. For some unknown reason, the worker that was attending the task was killed in a non-graceful way and therefore task was interrupted at the middle).

When task reaches SUCCESS or FAILURE status, you can know the result for your task through the "result" key returned by Task Module.

Response body when PENDING status is returned:

```
{
    "status": [string],
    "task_id": [string],
}
```

Response body when SUCCESS, PROGRESS, FAILURE or REVOKED status is returned:

```
{
    "status": [string],
    "task_id": [string],
    "result": [dict]
}
```

CHAPTER 8

Using GloboNetworkAPI V4

As module

/api/v4/as/

GET

Obtaining list of AS's

It is possible to specify in several ways fields desired to be retrieved in AS module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for AS module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- name
- description
- **equipments**
  - *id_as*
  - *equipment*

Obtaining list of AS's through id's URL:

```
/api/v4/as/[as_ids]/
```

where **as_ids** are the identifiers of AS's desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/as/1/
```

Many IDs:

```
/api/v4/as/1;3;8/
```

Obtaining list of AS's through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/as/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/as/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "name": "AS_BGP"
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and description:

```
fields=id,name,description
```

Using kind GET parameter

The AS module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field equipment_type for *basic* will contain only the identifier and for *details* will contain also the description.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "asns": [
        {
            "id": <integer>,
            "name": <string>,
            "description": <string>,
            "equipments": [
                {
                    "equipment": {
                        "id": <integer>,
                        "name": <string>
                    }
                },...
            ]
        },...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "asns": [
        {
            "id": <integer>,
            "name": <string>,
            "description": <string>,
            "equipments": [
                {
                    "equipment": {
                        "id": <integer>,
                        "name": <string>,
                        "maintenance": <boolean>,
                        "equipment_type": {
                            "id": <integer>,
                            "equipment_type": <string>
                        },
                        "model": {
                            "id": <integer>,
                            "name": <string>
                        },
                        "ipsv4": [
                            {
                                "ip": {
                                    "id": <integer>,
                                    "oct4": <integer>,
                                    "oct3": <integer>,
                                    "oct2": <integer>,
                                    "oct1": <integer>,
                                    "networkipv4": <integer>,
                                    "description": <string>
                                },
                                "virtual_interface": {
                                    "id": <integer>,
                                    "name": <string>,
                                    "vrf": {
```

```
                "id": <integer>,
                "internal_name": <string>,
                "vrf": <string>
            }
        }
    },...
],
"ipsv6": [
    {
        "ip": {
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "networkipv6": <integer>,
            "description": <string>
        },
        "virtual_interface": {
            "id": <integer>,
            "name": <string>,
            "vrf": {
                "id": <integer>,
                "internal_name": <string>,
                "vrf": <string>
            }
        }
    },...
],
"environments": [
    {
        "is_router": <boolean>,
        "is_controller": <boolean>,
        "environment": {
            "id": <integer>,
            "name": <string>,
            "grupo_l3": <integer>,
            "ambiente_logico": <integer>,
            "divisao_dc": <integer>,
            "filter": <integer>,
            "acl_path": <string>,
            "ipv4_template": <string>,
            "ipv6_template": <string>,
            "link": <string>,
            "min_num_vlan_1": <integer>,
            "max_num_vlan_1": <integer>,
            "min_num_vlan_2": <integer>,
            "max_num_vlan_2": <integer>,
            "default_vrf": <integer>,
            "father_environment": <reference-to:environment>,
            "sdn_controllers": null
        }
    },...
],
```

```
                        "groups": [
                            {
                                "id": <integer>,
                                "name": <string>
                            },...
                        ],
                        "id_as": {
                            "id": <integer>,
                            "name": <string>,
                            "description": <string>
                        }
                    }
                }
            ]
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "asns": [
        {
            "id": <integer>,
            "name": <string>,
            "description": <string>
        },...
    ]
}
```

POST

Creating list of AS's

URL:

```
/api/v4/as/
```

Request body:

```
{
    "asns": [
        {
```

```
            "name": <string>,
            "description": <string>
        },...
    ]
}
```

- Both **name** and **description** fields are required.

URL Example:

```
/api/v4/as/
```

PUT

Updating list of AS's

URL:

```
/api/v4/as/[as_ids]/
```

where **as_ids** are the identifiers of AS's. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/as/1/
```

Many IDs:

```
/api/v4/as/1;3;8/
```

Request body:

```
{
    "asns": [
        {
            "id": <integer>,
            "name": <string>,
            "description": <string>
        },...
    ]
}
```

- **id** field is mandatory. The other fields are not mandatory, but if they don't provided, they will be replaced by null.

URL Example:

```
/api/v4/as/1/
```

DELETE

Deleting list of AS's in database

Deleting list of AS's URL:

```
/api/v4/as/[as_ids]/
```

where **as_ids** are the identifiers of AS's desired to delete. It can use multiple id's separated by semicolons. If AS is associated with some Equipment, it cannot be deleted until this relationship be removed.

Example with Parameter IDs:

One ID:

```
/api/v4/as/1/
```

Many IDs:

```
/api/v4/as/1;3;8/
```

Equipment module

/api/v4/equipment/

GET

Obtaining list of Equipments

It is possible to specify in several ways fields desired to be retrieved in Equipment module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Equipment module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- name
- maintenance
- **equipment_type**
- **model**
  - name
  - **brand**
    - id
    - name
- **ipsv4**
  - *ip*
  - *virtual-interface*
- **ipsv6**
  - *ip*
  - *virtual-interface*
- **environments**

    - *environment*
    - *equipment*
- **groups**
- *id_as*

Obtaining list of Equipments through some Optional GET Parameters URL:

```
/api/v4/equipment/
```

Optional GET Parameters:

```
rights_write=[string]
environment=[integer]
ipv4=[string]
ipv6=[string]
is_router=[integer]
name=[string]
```

Where:

- **rights_write** must receive 1 if desired to obtain the equipments where at least one group to which the user logged in is related has write access.
- **environment** is some environment identifier.
- **ipv4** and **ipv6** are IP's must receive some valid IP Adresss.
- **is_router** must receive 1 if only router equipments are desired, 0 if only equipments that is not routers are desired.
- **name** is a unique string that only one equipment has.

Example:

With environment and ipv4 GET Parameter:

```
/api/v4/equipment/?ipv4=192.168.0.1&environment=5
```

Obtaining list of Equipments through id's URL:

```
/api/v4/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of Equipments desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/equipment/1/
```

Many IDs:

```
/api/v4/equipment/1;3;8/
```

Obtaining list of Equipments through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/equipment/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/equipment/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "maintenance": false,
        "tipo_equipamento": 1
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, name and maintenance:

```
fields=id,name,maintenance
```

Using kind GET parameter

The Equipment module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field equipment_type for *basic* will contain only the identifier and for *details* will contain also the description.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "equipments": [
        {
            "id": <integer>,
            "name": <string>
        }, ...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "equipments": [
        {
            "id": <integer>,
            "name": <string>,
            "maintenance": <boolean>,
            "equipment_type": {
                "id": <integer>,
                "equipment_type": <string>
            },
            "model": {
                "id": <integer>,
                "name": <string>
            },
            "ipsv4": [
                {
                    "ip": {
                        "id": <integer>,
                        "oct4": <integer>,
                        "oct3": <integer>,
                        "oct2": <integer>,
                        "oct1": <integer>,
                        "networkipv4": {
                            "id": <integer>,
                            "oct1": <integer>,
                            "oct2": <integer>,
                            "oct3": <integer>,
                            "oct4": <integer>,
                            "prefix": <integer>,
                            "networkv4": <string>,
                            "mask_oct1": <integer>,
                            "mask_oct2": <integer>,
                            "mask_oct3": <integer>,
                            "mask_oct4": <integer>,
                            "mask_formated": <string>,
                            "broadcast": <string>,
                            "vlan": {
                                "id": <integer>,
                                "name": <string>,
                                "num_vlan": <integer>,
                                "environment": <integer>,
                                "description": <string>,
                                "acl_file_name": <string>,
                                "acl_valida": <boolean>,
```

```
                        "acl_file_name_v6": <string>,
                        "acl_valida_v6": <boolean>,
                        "active": <boolean>,
                        "vrf": <string>,
                        "acl_draft": <string>,
                        "acl_draft_v6": <string>
                    },
                    "network_type": {
                        "id": <integer>,
                        "tipo_rede": <string>
                    },
                    "environmentvip": {
                        "id": <integer>,
                        "finalidade_txt": <string>,
                        "cliente_txt": <string>,
                        "ambiente_p44_txt": <string>,
                        "description": <string>
                    },
                    "active": <boolean>,
                    "dhcprelay": [
                        {
                            "id": <integer>,
                            "ipv4": <integer>,
                            "networkipv4": <integer>
                        }, ...
                    ],
                    "cluster_unit": <string>
                },
                "description": <string>
            },
            "virtual_interface": {
                "id": <integer>,
                "name": <string>,
                "vrf": {
                    "id": <integer>,
                    "internal_name": <string>,
                    "vrf": <string>
                }
            }
        }, ...
    ],
    "ipsv6": [
        {
            "ip": {
                "id": 1,
                "block1": <string>,
                "block2": <string>,
                "block3": <string>,
                "block4": <string>,
                "block5": <string>,
                "block6": <string>,
                "block7": <string>,
                "block8": <string>,
                "networkipv6": {
                    "id": <integer>,
                    "block1": <string>,
                    "block2": <string>,
                    "block3": <string>,
```

```
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "prefix": <integer>,
            "networkv6": <string>,
            "mask1": <string>,
            "mask2": <string>,
            "mask3": <string>,
            "mask4": <string>,
            "mask5": <string>,
            "mask6": <string>,
            "mask7": <string>,
            "mask8": <string>,
            "mask_formated": <string>,
            "vlan": {
                "id": <integer>,
                "name": <string>,
                "num_vlan": <integer>,
                "environment": <integer>,
                "description": <string>,
                "acl_file_name": <string>,
                "acl_valida": <boolean>,
                "acl_file_name_v6": <string>,
                "acl_valida_v6": <boolean>,
                "active": <boolean>,
                "vrf": <integer>,
                "acl_draft": <string>,
                "acl_draft_v6": <string>
            },
            "network_type": {
                "id": <integer>,
                "tipo_rede": <string>
            },
            "environmentvip": {
                "id": <integer>,
                "finalidade_txt": <string>,
                "cliente_txt": <string>,
                "ambiente_p44_txt": <string>,
                "description": <string>
            },
            "active": <boolean>,
            "dhcprelay": [
                {
                    "id": <integer>,
                    "ipv6": <integer>,
                    "networkipv6": <integer>
                }, ...
            ],
            "cluster_unit": <string>
        },
        "description": <string>
    },
    "virtual_interface": {
        "id": <integer>,
        "name": <string>,
        "vrf": {
```

```
                            "id": <integer>,
                            "internal_name": <string>,
                            "vrf": <string>
                        }
                    }
                }, ...
            ],
            "environments": [
                {
                    "is_router": <boolean>,
                    "is_controller": <boolean>,
                    "environment": {
                        "id": <integer>,
                        "name": <string>,
                        "grupo_l3": <integer>,
                        "ambiente_logico": <integer>,
                        "divisao_dc": <integer>,
                        "filter": <integer>,
                        "acl_path": <string>,
                        "ipv4_template": <string>,
                        "ipv6_template": <string>,
                        "link": <string>,
                        "min_num_vlan_1": <integer>,
                        "max_num_vlan_1": <integer>,
                        "min_num_vlan_2": <integer>,
                        "max_num_vlan_2": <integer>,
                        "default_vrf": <integer>,
                        "father_environment": <recurrence-to:environment>,
                        "sdn_controllers": null
                    }
                }, ...
            ],
            "groups": [
                {
                    "id": <integer>,
                    "name": <string>
                }, ...
            ],
            "id_as": {
                "id": <integer>,
                "name": <string>,
                "description": <string>
            }
        }, ...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "equipments": [
        {
            "id": <integer>,
            "name": <string>,
            "maintenance": <boolean>,
            "equipment_type": <integer>,
            "model": <integer>
        }, ...
    ]
}
```

POST

Creating list of equipments

URL:

```
/api/v4/equipment/
```

Request body:

```
{
    "equipments": [
        {
            "environments": [
                {
                    "id": <integer>,
                    "is_router": <boolean>,
                    "is_controller": <boolean>
                }, ...
            ],
            "equipment_type": <integer>,
            "groups": [
                {
                    "id": <integer>
                }, ...
            ],
            "ipsv4": [
                {
                    "ipv4": {
                        "id": <integer>
                    },
                    "virtual_interface": {
                        "id": <integer>
                    }
                }, ...
            ],
            "ipsv6": [
                {
                    "ipv6": {
```

```
                    "id": <integer>
                },
                "virtual_interface": {
                    "id": <integer>
                }
            }, ...
        ],
        "maintenance": <boolean>,
        "model": <integer>,
        "name": <string>,
        "id_as": <integer>
    }, ...
]
}
```

- **environments** - You can associate environments to new Equipment and specify if your equipment in each association will act as a router for specific environment.
- **equipment_type** - You must specify if your Equipment is a Switch, a Router, a Load Balancer...
- **groups** - You can associate the new Equipment to one or more groups of Equipments.
- **ipv4** - You can assign to the new Equipment how many IPv4 addresses is needed.
- **ipv6** - You can assign to the new Equipment how many IPv6 addresses is needed.
- **maintenance** - You must assign to the new Equipment a flag saying if the Equipment is or not in maintenance mode.
- **model** - You must assign to the Equipment some model (Cisco, Dell, HP, F5, ...).
- **name** - You must assign to the Equipment any name.

URL Example:

```
/api/v4/equipment/
```

PUT

Updating list of equipments in database

URL:

```
/api/v4/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of equipments. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/equipment/1/
```

Many IDs:

```
/api/v4/equipment/1;3;8/
```

Request body:

```
{
    "equipments": [
        {
            "id": <integer>,
            "environments": [
                {
                    "id": <integer>,
                    "is_router": <boolean>,
                    "is_controller": <boolean>
                }, ...
            ],
            "equipment_type": <integer>,
            "groups": [
                {
                    "id": <integer>
                }, ...
            ],
            "ipsv4": [
                {
                    "ipv4": {
                        "id": <integer>
                    },
                    "virtual_interface": {
                        "id": <integer>
                    }
                }, ...
            ],
            "ipsv6": [
                {
                    "ipv6": {
                        "id": <integer>
                    },
                    "virtual_interface": {
                        "id": <integer>
                    }
                }, ...
            ],
            "maintenance": <boolean>,
            "model": <integer>,
            "name": <string>,
            "id_as": <integer>
        }, ...
    ]
}
```

- **id** - Specify what Equipment you want to change.
- **environments** - You can associate environments to new Equipment and specify if your equipment in each association will act as a router for specific environment and if it will act as a SDN controller in this particular environment.
- **equipment_type** - You must specify if your Equipment is a Switch, a Router, a Load Balancer...
- **groups** - You can associate the new Equipment to one or more groups of Equipments.
- **ipsv4** - You can assign to the new Equipment how many IPv4 addresses are needed and for each association between IPv4 and Equipment you can set a Virtual Interface.
- **ipsv6** - You can assign to the new Equipment how many IPv6 addresses are needed and for each association between IPv6 and Equipment you can set a Virtual Interface.

- **maintenance** - You must assign to the new Equipment a flag saying if the Equipment is or not in maintenance mode.
- **model** - You must assign to the Equipment some model (Cisco, Dell, HP, F5, ...).
- **name** - You must assign to the Equipment any name.
- **id_as** - You can associate the Equipment with one ASN.

Remember that if you don't provide the not mandatory fields, actual information (e.g. associations between Equipment and Environments) will be deleted. The effect of PUT Request is always to replace actual data by what you provide into fields in this type of request.

URL Example:

```
/api/v4/equipment/1/
```

DELETE

Deleting a list of equipments in database

Deleting list of equipments and all relationships URL:

```
/api/v4/equipment/[equipment_ids]/
```

where **equipment_ids** are the identifiers of equipments desired to delete. It can use multiple id's separated by semicolons. Doing this, all associations between Equipments and IP addresses, Access, Script (Roteiro), Interface, Environment and Group will be deleted. Equipments that have a relationship at same time between IPv4 and Virtual Interface objects or IPv6 and Virtual Interface objects can't be deleted.

Example with Parameter IDs:

One ID:

```
/api/v4/equipment/1/
```

Many IDs:

```
/api/v4/equipment/1;3;8/
```

IPv4 module

/api/v3/ipv4/

GET

Obtaining list of IPv4 objects

It is possible to specify in several ways fields desired to be retrieved in IPv4 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for IPv4 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id

- ip_formated
- oct1
- oct2
- oct3
- oct4
- *networkipv4*
- description
- **equipments**
  - *equipment*
  - *virtual-interface*
- *vips*
- **server_pool_members**
  - id
  - *server_pool*
  - identifier
  - *ip*
  - *ipv6*
  - priority
  - weight
  - limit
  - port_real
  - member_status
  - last_status_update
  - last_status_update_formated
  - *equipments*
  - *equipment*

Obtaining list of IPv4 objects through id's URL:

```
/api/v4/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of IPv4 objects desired to be retrieved. It can use multiple id's separated by semi-colons.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv4/1/
```

Many IDs:

```
/api/v4/ipv4/1;3;8/
```

Obtaining list of IPv4 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/ipv4/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/ipv4/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "oct1": 10,
        },
        {
            "oct1": 172,
        }

    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, ip_formated and networkipv4:

```
fields=id,ip_formated,networkipv4
```

Using kind GET parameter

The IPv4 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "networkipv4": {
                "id": <integer>,
                "networkv4": <string>,
                "mask_formated": <string>,
                "broadcast": <string>,
                "vlan": {
                    "id": <integer>,
                    "name": <string>,
                    "num_vlan": <integer>
                },
                "network_type": <integer>,
                "environmentvip": <integer>
            },
            "description": <string>
        }
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": {
                "id": <integer>,
                "oct1": <integer>,
                "oct2": <integer>,
                "oct3": <integer>,
                "oct4": <integer>,
                "prefix": <integer>,
                "networkv4": <string>,
                "mask_oct1": <integer>,
                "mask_oct2": <integer>,
                "mask_oct3": <integer>,
                "mask_oct4": <integer>,
                "mask_formated": <string>,
                "broadcast": <string>,
                "vlan": {
                    "id": <integer>,
```

```
            "name": <string>,
            "num_vlan": <integer>,
            "environment": <integer>,
            "description": <string>,
            "acl_file_name": <string>,
            "acl_valida": <boolean>,
            "acl_file_name_v6": <string>,
            "acl_valida_v6": <boolean>,
            "active": <boolean>,
            "vrf": <string>,
            "acl_draft": <string>,
            "acl_draft_v6": <string>
        },
        "network_type": {
            "id": <integer>,
            "tipo_rede": <string>
        },
        "environmentvip": {
            "id": <integer>,
            "finalidade_txt": <string>,
            "cliente_txt": <string>,
            "ambiente_p44_txt": <string>,
            "description": <string>
        },
        "active": <boolean>,
        "dhcprelay": [
            <string>,...
        ],
        "cluster_unit": <string>
    },
    "description": <string>,
    "equipments": [
        {
            "equipment": {
                "id": <integer>,
                "name": <string>,
                "maintenance": <boolean>,
                "equipment_type": {
                    "id": <integer>,
                    "equipment_type": <string>
                },
                "model": {
                    "id": <integer>,
                    "name": <string>
                },
                "environments": [
                    {
                        "is_router": <boolean>,
                        "is_controller": <boolean>,
                        "environment": {
                            "id": <integer>,
                            "name": <string>,
                            "grupo_l3": <integer>,
                            "ambiente_logico": <integer>,
                            "divisao_dc": <integer>,
                            "filter": <integer>,
                            "acl_path": <string>,
                            "ipv4_template": <string>,
```

```
                                    "ipv6_template": <string>,
                                    "link": <string>,
                                    "min_num_vlan_1": <integer>,
                                    "max_num_vlan_1": <integer>,
                                    "min_num_vlan_2": <integer>,
                                    "max_num_vlan_2": <integer>,
                                    "default_vrf": <integer>,
                                    "father_environment": <recurrence-to:environment>,
                                    "sdn_controllers": null
                                }
                            }
                        ],
                        "groups": [
                            {
                                "id": <integer>,
                                "name": <string>
                            }
                        ],
                        "id_as": {
                            "id": <integer>,
                            "name": <string>,
                            "description": <string>
                        }
                    },
                    "virtual_interface": {
                        "id": <integer>,
                        "name": <string>,
                        "vrf": {
                            "id": <integer>,
                            "internal_name": <string>,
                            "vrf": <string>
                        }
                    }
                }
            ]
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ips":[
        {
```

```
            "id": <integer>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": <integer>,
            "description": <string>
        }
    ]
}
```

POST

Creating list of IPv4 objects

URL:

```
/api/v4/ipv4/
```

Request body:

```
{
    "ips": [{
        "oct1": <integer>,
        "oct2": <integer>,
        "oct3": <integer>,
        "oct4": <integer>,
        "networkipv4": <integer>,
        "description": <string>,
        "equipments": [
            {
                "equipment": {
                    "id": <integer>
                },
                "virtual_interface": {
                    "id": <integer>
                }
            }, ...
        ]
    },..]
}
```

Request Example with only required fields:

```
{
    "ips": [{
        "networkipv4": 10
    }]
}
```

Request Example with some more fields:

```
{
    "ips": [{
        "oct1": 10,
        "oct2": 10,
        "oct3": 0,
```

```
        "oct4": 20,
        "networkipv4": 2,
        "equipments": [
            {
                "equipment": {
                    "id": 1
                },
                "virtual_interface": {
                    "id": 1
                }
            },
            {
                "equipment": {
                    "id": 2
                }
            }
        ]
    }]
}
```

Through IPv4 POST route you can create one or more IPv4 objects. Only "networkipv4" field are required. You can specify other fields such as:

- **oct1**, **oct2**, **oct3**, **oct4** - Are the octets of IPv4. Given a network, API can provide to you an IPv4 Address automatically, but you can assign a IPv4 Address in a manually way. If you specify some octet, you need to specify all the others.
- **description** - Description of new IPv4.
- **networkipv4** - This parameter is mandatory. It is the network to which new IP address will belong.
- **equipments** - You can associate new IPv4 address to one or more equipments and with Virtual Interfaces together. In the association to Equipment it's not mandatory to specify Virtual Interface.

At the end of POST request, it will be returned the identifiers of new IPv4 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two IPv4 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v4/ipv4/
```

PUT

Updating list of IPv4 objects in database

URL:

```
/api/v4/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of IPv4 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv4/1/
```

Many IDs:

```
/api/v4/ipv4/1;3;8/
```

Request body:

```
{
    "ips": [{
        "id": <integer>,
        "description": <string>,
        "equipments": [
            {
                "equipment": {
                    "id": <integer>
                },
                "virtual_interface": {
                    "id": <integer>
                }
            }, ...
        ]
    },..]
}
```

Request Example:

```
{
    "ips": [{
        "id": 1,
        "description": "New description",
        "equipments": [
            {
                "equipment": {
                    "id": 1
                },
                "virtual_interface": {
                    "id": 1
                }
            },
            {
                "equipment": {
                    "id": 2
                }
            }
        ]
```

```
    }]
}
```

In IPv4 PUT request, you can only change description and associations with equipments.

- **id** - Identifier of IPv4 that will be changed. It's mandatory.
- **description** - Description of new IPv4.
- **equipments** - You can create new associations with equipments and Virtual Interfaces when updating IPv4. Old associations will be deleted even you don't specify new associations to other equipments if all of them not contains a Virtual Interface. If some Virtual Interface appears at least one relationship between IPv4 and Equipment, it can't be deleted and the IPv4 will not be updated.

URL Example:

```
/api/v4/ipv4/1/
```

DELETE

Deleting a list of IPv4 objects in database

Deleting list of IPv4 objects and associated Vip Requests and relationships with Equipments URL:

```
/api/v4/ipv4/[ipv4_ids]/
```

where **ipv4_ids** are the identifiers of ipv4s desired to delete. It can use multiple id's separated by semicolons. Doing this, all Vip Request associated with IPv4 desired to be deleted will be deleted too. All associations made to equipments will also be deleted. If Virtual Interface is present in some association of IPv4 desired to be deleted to some equipment, the association will not be deleted and therefore the IPv4 will also not be deleted.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv4/1/
```

Many IDs:

```
/api/v4/ipv4/1;3;8/
```

/api/v3/ipv4/async/

POST

Creating list of IPv4 asynchronously

URL:

```
/api/v4/ipv4/async/
```

You can also create IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv4 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv4 desired to be created in response, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of IPv4 asynchronously

URL:

```
/api/v4/ipv4/async/
```

You can also update IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv4 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv4 desired to be updated in response, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
```

```
    }
]
```

DELETE

Deleting list of IPv4 asynchronously

URL:

```
/api/v4/ipv4/async/
```

You can also delete IPv4 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous IPv4 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in the synchronous request, but for each IPv4 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv4 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv4/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    }, ...
]
```

Response Example for update of two IPv4 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

IPv6 module

/api/v4/ipv6/

GET

Obtaining list of IPv6 objects

It is possible to specify in several ways fields desired to be retrieved in IPv6 module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for IPv6 module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using

fields, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- ip_formated
- block1
- block2
- block3
- block4
- block5
- block6
- block7
- block8
- *networkipv6*
- description
- **equipments**
  - *equipment*
  - *virtual-interface*
- *vips*
- **server_pool_members**
  - id
  - *server_pool*
  - identifier
  - *ip*
  - *ipv6*
  - priority
  - weight
  - limit
  - port_real
  - member_status
  - last_status_update
  - last_status_update_formated
  - *equipments*
  - *equipment*

Obtaining list of IPv6 objects through id's URL:

```
/api/v4/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of IPv6 objects desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv6/1/
```

Many IDs:

```
/api/v4/ipv6/1;3;8/
```

Obtaining list of IPv6 objects through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/ipv6/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/ipv6/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [
        {
            "block1": "fefe",
        },
        {
            "block1": "fdfd",
        }

    ],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, ip_formated and networkipv6:

```
fields=id,ip_formated,networkipv6
```

Using kind GET parameter

The IPv6 module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "networkipv6": {
                "id": <integer>,
                "networkv6": <string>,
                "mask_formated": <string>,
                "vlan": {
                    "id": <integer>,
                    "name": <string>,
                    "num_vlan": <integer>
                },
                "network_type": <integer>,
                "environmentvip": <integer>
            },
            "description": <string>
        }
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "ips": [
        {
            "id": <integer>,
            "ip_formated": <string>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
```

```
"networkipv6": {
    "id": <integer>,
    "block1": <string>,
    "block2": <string>,
    "block3": <string>,
    "block4": <string>,
    "block5": <string>,
    "block6": <string>,
    "block7": <string>,
    "block8": <string>,
    "prefix": <integer>,
    "networkv6": <string>,
    "mask1": <string>,
    "mask2": <string>,
    "mask3": <string>,
    "mask4": <string>,
    "mask5": <string>,
    "mask6": <string>,
    "mask7": <string>,
    "mask8": <string>,
    "mask_formated": <string>,
    "vlan": {
        "id": <integer>,
        "name": <string>,
        "num_vlan": <integer>,
        "environment": <integer>,
        "description": <string>,
        "acl_file_name": <string>,
        "acl_valida": <boolean>,
        "acl_file_name_v6": <string>,
        "acl_valida_v6": <boolean>,
        "active": <boolean>,
        "vrf": <string>,
        "acl_draft": <string>,
        "acl_draft_v6": <string>
    },
    "network_type": {
        "id": <integer>,
        "tipo_rede": <string>
    },
    "environmentvip": {
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>
    },
    "active": <boolean>,
    "dhcprelay": [
        <string>,...
    ],
    "cluster_unit": <string>
},
"description": <string>,
"equipments": [
    {
        "equipment": {
            "id": <integer>,
```

```
                "name": <string>,
                "maintenance": <boolean>,
                "equipment_type": {
                    "id": <integer>,
                    "equipment_type": <string>
                },
                "model": {
                    "id": <integer>,
                    "name": <string>
                },
                "environments": [
                    {
                        "is_router": <boolean>,
                        "is_controller": <boolean>,
                        "environment": {
                            "id": <integer>,
                            "name": <string>,
                            "grupo_l3": <integer>,
                            "ambiente_logico": <integer>,
                            "divisao_dc": <integer>,
                            "filter": <integer>,
                            "acl_path": <string>,
                            "ipv4_template": <string>,
                            "ipv6_template": <string>,
                            "link": <string>,
                            "min_num_vlan_1": <integer>,
                            "max_num_vlan_1": <integer>,
                            "min_num_vlan_2": <integer>,
                            "max_num_vlan_2": <integer>,
                            "default_vrf": <integer>,
                            "father_environment": <recurrence-to:environment>,
                            "sdn_controllers": null
                        }
                    }
                ],
                "groups": [
                    {
                        "id": <integer>,
                        "name": <string>
                    }
                ],
                "id_as": {
                    "id": <integer>,
                    "name": <string>,
                    "description": <string>
                }
            },
            "virtual_interface": {
                "id": <integer>,
                "name": <string>,
                "vrf": {
                    "id": <integer>,
                    "internal_name": <string>,
                    "vrf": <string>
                }
            }
        }
    ]
```

```
        }
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "ips":[
        {
            "id": <integer>,
            "block1": <string>,
            "block2": <string>,
            "block3": <string>,
            "block4": <string>,
            "block5": <string>,
            "block6": <string>,
            "block7": <string>,
            "block8": <string>,
            "networkipv6": <integer>,
            "description": <string>
        }
    ]
}
```

POST

Creating list of IPv6 objects

URL:

```
/api/v4/ipv6/
```

Request body:

```
{
    "ips": [{
        "block1": <string>,
        "block2": <string>,
        "block3": <string>,
        "block4": <string>,
        "block5": <string>,
        "block6": <string>,
        "block7": <string>,
```

```
        "block8": <string>,
        "networkipv6": <integer>,
        "description": <string>,
        "equipments": [
            {
                "equipment": {
                    "id": <integer>
                },
                "virtual_interface": {
                    "id": <integer>
                }
            }, ...
        ]
    },..]
}
```

Request Example with only required fields:

```
{
    "ips": [{
        "networkipv6": 10
    }]
}
```

Request Example with some more fields:

```
{
    "ips": [{
        "block1": "fdbe",
        "block2": "fdbe",
        "block3": "0000",
        "block4": "0000",
        "block5": "0000",
        "block6": "0000",
        "block7": "0000",
        "block8": "0000",
        "networkipv6": 2,
        "equipments": [
            {
                "equipment": {
                    "id": 1
                },
                "virtual_interface": {
                    "id": 1
                }
            },
            {
                "equipment": {
                    "id": 2
                }
            }
        ]
    }]
}
```

Through IPv6 POST route you can create one or more IPv6 objects. Only "networkipv6" field are required. You can specify other fields such as:

- **block1**, **block2**, **block3**, **block4**, **block5**, **block6**, **block7** and **block8** - Are the octets of IPv6. Given a network,

API can provide to you an IPv6 Address automatically, but you can assign a IPv6 Address in a manually way. If you specify some octet, you need to specify all the others.

- **networkipv6** - This parameter is mandatory. It is the network to which new IP address will belong.
- **description** - Description of new IPv6.
- **equipments** - You can associate new IPv6 address to one or more equipments and with Virtual Interfaces together. In the association to Equipment it's not mandatory to specify Virtual Interface.

At the end of POST request, it will be returned the identifiers of new IPv6 objects created.

Response Body:

```
[
    {
        "id": <integer>
    },...
]
```

Response Example for two IPv6 objects created:

```
[
    {
        "id": 10
    },
    {
        "id": 11
    }
]
```

URL Example:

```
/api/v4/ipv6/
```

PUT

Updating list of IPv6 objects in database

URL:

```
/api/v4/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of IPv6 objects. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv6/1/
```

Many IDs:

```
/api/v4/ipv6/1;3;8/
```

Request body:

```
{
    "ips": [{
        "id": <integer>,
        "description": <string>,
```

```
        "equipments": [
            {
                "equipment": {
                    "id": <integer>
                },
                "virtual_interface": {
                    "id": <integer>
                }
            }, ...
        ]
    },..]
}
```

Request Example:

```
{
    "ips": [{
        "id": 1,
        "description": "New description",
        "equipments": [
            {
                "equipment": {
                    "id": 1
                },
                "virtual_interface": {
                    "id": 1
                }
            },
            {
                "equipment": {
                    "id": 2
                }
            }
        ]
    }]
}
```

In IPv6 PUT request, you can only change description and associations with equipments.

- **id** - Identifier of IPv6 that will be changed. It's mandatory.
- **description** - Description of new IPv6.
- **equipments** - You can create new associations with equipments and Virtual Interfaces when updating IPv6. Old associations will be deleted even you don't specify new associations to other equipments if all of them not contains a Virtual Interface. If some Virtual Interface appears at least one relationship between IPv6 and Equipment, it can't be deleted and the IPv6 will not be updated.

URL Example:

```
/api/v4/ipv6/1/
```

DELETE

Deleting a list of IPv6 objects in database

Deleting list of IPv6 objects and associated Vip Requests and relationships with Equipments and Virtual Interfaces URL:

```
/api/v4/ipv6/[ipv6_ids]/
```

where **ipv6_ids** are the identifiers of ipv6s desired to delete. It can use multiple id's separated by semicolons. Doing this, all Vip Request associated with IPv6 desired to be deleted will be deleted too. All associations made to equipments will also be deleted. If Virtual Interface is present in some association of IPv6 desired to be deleted to some equipment, the association will not be deleted and therefore the IPv6 will also not be deleted.

Example with Parameter IDs:

One ID:

```
/api/v4/ipv6/1/
```

Many IDs:

```
/api/v4/ipv6/1;3;8/
```

/api/v4/ipv6/async/

POST

Creating list of IPv6 asynchronously

URL:

```
/api/v4/ipv6/async/
```

You can also create IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv6 Creating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv6 desired to be created in response, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been created after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

PUT

Updating list of IPv6 asynchronously

URL:

```
/api/v4/ipv6/async/
```

You can also update IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information about request body please check *Synchronous IPv6 Updating*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive the identifier of each IPv6 desired to be updated in response, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

DELETE

Deleting list of IPv6 asynchronously

URL:

```
/api/v4/ipv6/async/
```

You can also delete IPv6 objects asynchronously. It is only necessary to provide the same as in the respective synchronous request (For more information please check *Synchronous IPv6 Deleting*). In this case, when you make request NetworkAPI will create a task to fullfil it. You will not receive an empty dict in response as occurs in the synchronous request, but for each IPv6 you will receive an identifier for the created task. Since this is an asynchronous request, it may be that IPv6 objects have been updated after you receive the response. It is your task, therefore, to consult the API through the available means to verify that your request have been met.

URL Example:

```
/api/v4/ipv6/async/
```

Response body:

```
[
    {
        "task_id": [string with 36 characters]
    },...
]
```

Response Example for update of two IPv6 objects:

```
[
    {
        "task_id": "36dc887e-48bf-4c83-b6f5-281b70976a8f"
    },
    {
        "task_id": "17ebd466-0231-4bd0-8f78-54ed20238fa3"
    }
]
```

Neighbor module

/api/v4/neighbor/

GET

Obtaining list of Neighbors

It is possible to specify in several ways fields desired to be retrieved in Neighbor module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Neighbor module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- remote_as
- remote_ip
- password
- maximum_hops
- timer_keepalive
- timer_timeout
- description
- soft_reconfiguration
- community
- remove_private_as
- next_hop_self
- kind

- created
- *virtual-interface*

Obtaining list of Equipments through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/neighbor/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/neighbor/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "community": false,
    }],
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id, password and community:

```
fields=id,password,community
```

Using kind GET parameter

The Neighbor module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field virtual_interface for *basic* will contain only the identifier and for *details* will contain a bunch of information.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "neighbors": [{
        "id": <integer>,
        "remote_as": <string>,
        "remote_ip": <string>,
        "password": <string>,
        "maximum_hops": <string>,
        "timer_keepalive": <string>,
        "timer_timeout": <string>,
        "description": <string>,
        "soft_reconfiguration": <boolean>,
        "community": <boolean>,
        "remove_private_as": <boolean>,
        "next_hop_self": <boolean>,
        "kind": <string>,
        "created": <boolean>,
        "virtual_interface": {
            "id": <integer>,
            "name": <string>,
            "vrf": <integer>
        }
    }]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "neighbors": [
        {
            "id": <integer>,
            "remote_as": <string>,
            "remote_ip": <string>,
            "password": <string>,
            "maximum_hops": <string>,
            "timer_keepalive": <string>,
            "timer_timeout": <string>,
            "description": <string>,
            "soft_reconfiguration": <boolean>,
            "community": <boolean>,
            "remove_private_as": <boolean>,
            "next_hop_self": <boolean>,
            "kind": <string>,
            "created": <boolean>,
            "virtual_interface": {
                "id": <integer>,
                "name": <string>,
                "vrf": {
                    "id": <integer>,
                    "internal_name": <string>,
                    "vrf": <string>
                },
```

```
"ipv4_equipment": [
    {
        "ip": {
            "id": <integer>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": <integer>,
            "description": <string>
        },
        "equipment": {
            "id": <integer>,
            "name": <string>,
            "maintenance": <boolean>,
            "equipment_type": {
                "id": <integer>,
                "equipment_type": <string>
            },
            "model": {
                "id": <integer>,
                "name": <string>
            },
            "environments": [
                {
                    "is_router": <boolean>,
                    "is_controller": <boolean>,
                    "environment": {
                        "id": <integer>,
                        "name": <string>,
                        "grupo_l3": <integer>,
                        "ambiente_logico": <integer>,
                        "divisao_dc": <integer>,
                        "filter": <integer>,
                        "acl_path": <string>,
                        "ipv4_template": <string>,
                        "ipv6_template": <string>,
                        "link": <string>,
                        "min_num_vlan_1": <integer>,
                        "max_num_vlan_1": <integer>,
                        "min_num_vlan_2": <integer>,
                        "max_num_vlan_2": <integer>,
                        "default_vrf": <integer>,
                        "father_environment": <recursion-to:environment>,
                        "sdn_controllers": null
                    }
                }, ...
            ],
            "groups": [
                {
                    "id": <integer>,
                    "name": <string>
                }, ...
            ],
            "id_as": {
                "id": <integer>,
                "name": <string>,
                "description": <string>
```

```
                }
            }
        }, ...
    ],
    "ipv6_equipment": [
        {
            "ip": {
                "id": <integer>,
                "block1": <string>,
                "block2": <string>,
                "block3": <string>,
                "block4": <string>,
                "block5": <string>,
                "block6": <string>,
                "block7": <string>,
                "block8": <string>,
                "networkipv6": <integer>,
                "description": <string>
            },
            "equipment": {
                "id": <integer>,
                "name": <string>,
                "maintenance": <boolean>,
                "equipment_type": {
                    "id": <integer>,
                    "equipment_type": <string>
                },
                "model": {
                    "id": <integer>,
                    "name": <string>
                },
                "environments": [
                    {
                        "is_router": <boolean>,
                        "is_controller": <boolean>,
                        "environment": {
                            "id": <integer>,
                            "name": <string>,
                            "grupo_l3": <integer>,
                            "ambiente_logico": <integer>,
                            "divisao_dc": <integer>,
                            "filter": <integer>,
                            "acl_path": <string>,
                            "ipv4_template": <string>,
                            "ipv6_template": <string>,
                            "link": <string>,
                            "min_num_vlan_1": <integer>,
                            "max_num_vlan_1": <integer>,
                            "min_num_vlan_2": <integer>,
                            "max_num_vlan_2": <integer>,
                            "default_vrf": <integer>,
                            "father_environment": <recursion-to:environment>,
                            "sdn_controllers": null
                        }
                    }, ...
                ],
                "groups": [
                    {
```

```
                                    "id": <integer>,
                                    "name": <string>
                                }, ...
                            ],
                            "id_as": {
                                "id": <integer>,
                                "name": <string>,
                                "description": <string>
                            }
                        }
                    }, ...
                ]
            }
        }, ...
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "neighbors": [{
        "id": 1,
        "remote_as": <string>,
        "remote_ip": <string>,
        "password": <string>,
        "maximum_hops": <string>,
        "timer_keepalive": <string>,
        "timer_timeout": <string>,
        "description": <string>,
        "soft_reconfiguration": <boolean>,
        "community": <boolean>,
        "remove_private_as": <boolean>,
        "next_hop_self": <boolean>,
        "kind": <string>,
        "created": <boolean>,
        "virtual_interface": 1
    }, ...]
}
```

POST

Creating list of Neighbors

URL:

```
/api/v4/neighbor/
```

Request body:

```
{
    "neighbors": [
        {
            "id": <integer>,
            "remote_as": <string>,
            "remote_ip": <string>,
            "password": <string>,
            "maximum_hops": <string>,
            "timer_keepalive": <string>,
            "timer_timeout": <string>,
            "description": <string>,
            "soft_reconfiguration": <boolean>,
            "community": <boolean>,
            "remove_private_as": <boolean>,
            "next_hop_self": <boolean>,
            "kind": <string>,
            "virtual_interface": <integer:virtual_interface_fk>
        }, ...
    ]
}
```

- **virtual_interface** - You can associate a virtual interface to new Neighbor passing its identifier in this field.

URL Example:

```
/api/v4/neighbor/
```

PUT

Updating list of neighbors in database

URL:

```
/api/v4/neighbor/[neighbor_ids]/
```

where **neighbor_ids** are the identifiers of neighbors. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/neighbor/1/
```

Many IDs:

```
/api/v4/neighbor/1;3;8/
```

Request body:

```
{
    "neighbors": [
        {
            "id": <integer>,
            "remote_as": <string>,
            "remote_ip": <string>,
            "password": <string>,
            "maximum_hops": <string>,
            "timer_keepalive": <string>,
            "timer_timeout": <string>,
            "description": <string>,
            "soft_reconfiguration": <boolean>,
            "community": <boolean>,
            "remove_private_as": <boolean>,
            "next_hop_self": <boolean>,
            "kind": <string>,
            "virtual_interface": <integer:virtual_interface_fk>
        }, ...
    ]
}
```

- **virtual_interface** - You can associate a virtual interface to new Neighbor passing its identifier in this field.

Remember that if you don't provide the not mandatory fields, actual information (e.g. association to Virtual Interface) will be deleted. The effect of PUT Request is always to replace actual data by what you provide into fields in this type of request.

URL Example:

```
/api/v4/neighbor/1/
```

DELETE

Deleting a list of neighbors in database

Deleting list of neighbors URL:

```
/api/v4/neighbor/[neighbor_ids]/
```

where **neighbor_ids** are the identifiers of neighbors desired to delete. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/neighbor/1/
```

Many IDs:

```
/api/v4/neighbor/1;3;8/
```

Virtual Interface module

/api/v4/virtual-interface/

GET

Obtaining list of Virtual Interfaces

It is possible to specify in several ways fields desired to be retrieved in Virtual Interface module through the use of some GET parameters. You are not required to use these parameters, but depending on your needs it can make your requests faster if you are dealing with many objects and you need few fields. The following fields are available for Virtual Interface module (hyperlinked or bold marked fields acts as foreign keys and can be expanded using __basic or __details when using **fields**, **include** or **exclude** GET Parameters. Hyperlinked fields points to its documentation. Some expandable fields that do not have documentation have its childs described here too because some of these childs are also expandable.):

- id
- name
- *vrf*

Obtaining list of Virtual Interfaces through id's URL:

```
/api/v4/virtual-interface/[equipment_ids]/
```

where **equipment_ids** are the identifiers of Virtual Interfaces desired to be retrieved. It can use multiple id's separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/virtual-interface/1/
```

Many IDs:

```
/api/v4/virtual-interface/1;3;8/
```

Obtaining list of Virtual Interfaces through extended search More information about Django QuerySet API, please see:

```
:ref:`Django QuerySet API reference <https://docs.djangoproject.com/el/1.10/ref/models/querysets/>`_
```

URL:

```
/api/v4/virtual-interface/
```

GET Parameter:

```
search=[encoded dict]
```

Example:

```
/api/v4/virtual-interface/?search=[encoded dict]
```

Request body example:

```
{
    "extends_search": [{
        "vrf__id": 1,
        "name__contains": "abc"
    }],
```

```
    "start_record": 0,
    "custom_search": "",
    "end_record": 25,
    "asorting_cols": [],
    "searchable_columns": []
}
```

- When **"search"** is used, "total" property is also retrieved.

Using fields GET parameter

Through **fields**, you can specify desired fields.

Example with field id:

```
fields=id
```

Example with fields id and vrf:

```
fields=id,vrf
```

Using kind GET parameter

The Virtual Interface module also accepts the **kind** GET parameter. Only two values are accepted by **kind**: *basic* or *details*. For each value it has a set of default fields. The difference between them is that in general *details* contains more fields than *basic*, and the common fields between them are more detailed for *details*. For example, the field equipment_type for *basic* will contain only the identifier and for *details* will contain also the description.

Example with basic option:

```
kind=basic
```

Response body with *basic* kind:

```
{
    "virtual_interfaces": [
        {
            "id": <integer>,
            "name": <string>,
            "vrf": {
                "id": <integer>,
                "internal_name": <string>,
                "vrf": <string>
            }
        }, ...
    ]
}
```

Example with details option:

```
kind=details
```

Response body with *details* kind:

```
{
    "virtual_interfaces": [
        {
            "id": <integer>,
```

```
"name": <string>,
"vrf": {
    "id": <integer>,
    "internal_name": <string>,
    "vrf": <string>
},
"ipv4_equipment": [
    {
        "ip": {
            "id": <integer>,
            "oct4": <integer>,
            "oct3": <integer>,
            "oct2": <integer>,
            "oct1": <integer>,
            "networkipv4": {
                "id": <integer>,
                "oct1": <integer>,
                "oct2": <integer>,
                "oct3": <integer>,
                "oct4": <integer>,
                "prefix": <integer>,
                "networkv4": <string>,
                "mask_oct1": <integer>,
                "mask_oct2": <integer>,
                "mask_oct3": <integer>,
                "mask_oct4": <integer>,
                "mask_formated": <string>,
                "broadcast": <string>,
                "vlan": {
                    "id": <integer>,
                    "name": <string>,
                    "num_vlan": <integer>,
                    "environment": <integer>,
                    "description": <string>,
                    "acl_file_name": <string>,
                    "acl_valida": <boolean>,
                    "acl_file_name_v6": <string>,
                    "acl_valida_v6": <boolean>,
                    "active": <boolean>,
                    "vrf": <string>,
                    "acl_draft": <string>,
                    "acl_draft_v6": <string>
                },
                "network_type": {
                    "id": <integer>,
                    "tipo_rede": <string>
                },
                "environmentvip": {
                    "id": <integer>,
                    "finalidade_txt": <string>,
                    "cliente_txt": <string>,
                    "ambiente_p44_txt": <string>,
                    "description": <string>
                },
                "active": <boolean>,
                "dhcprelay": [
                    <string>, ...
                ],
```

```
                "cluster_unit": <string>
            },
            "description": <string>
        },
        "equipment": {
            "id": <integer>,
            "name": <string>,
            "maintenance": <boolean>,
            "equipment_type": {
                "id": <integer>,
                "equipment_type": <string>
            },
            "model": {
                "id": <integer>,
                "name": <string>
            },
            "environments": [
                {
                    "is_router": <boolean>,
                    "is_controller": <boolean>,
                    "environment": {
                        "id": <integer>,
                        "name": <string>,
                        "grupo_l3": <integer>,
                        "ambiente_logico": <integer>,
                        "divisao_dc": <integer>,
                        "filter": <integer>,
                        "acl_path": <string>,
                        "ipv4_template": <string>,
                        "ipv6_template": <string>,
                        "link": <string>,
                        "min_num_vlan_1": <integer>,
                        "max_num_vlan_1": <integer>,
                        "min_num_vlan_2": <integer>,
                        "max_num_vlan_2": <integer>,
                        "default_vrf": <integer>,
                        "father_environment": <recurrence-to:environment>,
                        "sdn_controllers": null
                    }
                }, ...
            ],
            "groups": [
                {
                    "id": <integer>,
                    "name": <string>
                }, ...
            ],
            "id_as": {
                "id": <integer>,
                "name": <string>,
                "description": <string>
            }
        }
    }, ...
],
"ipv6_equipment": [
    {
        "ip": {
```

```
"id": <integer>,
"block1": <string>,
"block2": <string>,
"block3": <string>,
"block4": <string>,
"block5": <string>,
"block6": <string>,
"block7": <string>,
"block8": <string>,
"networkipv6": {
    "id": <integer>,
    "block1": <string>,
    "block2": <string>,
    "block3": <string>,
    "block4": <string>,
    "block5": <string>,
    "block6": <string>,
    "block7": <string>,
    "block8": <string>,
    "prefix": <integer>,
    "networkv6": <string>,
    "mask1": <string>,
    "mask2": <string>,
    "mask3": <string>,
    "mask4": <string>,
    "mask5": <string>,
    "mask6": <string>,
    "mask7": <string>,
    "mask8": <string>,
    "mask_formated": <string>,
    "vlan": {
        "id": <integer>,
        "name": <string>,
        "num_vlan": <integer>,
        "environment": <integer>,
        "description": <string>,
        "acl_file_name": <string>,
        "acl_valida": <boolean>,
        "acl_file_name_v6": <string>,
        "acl_valida_v6": <boolean>,
        "active": <boolean>,
        "vrf": <string>,
        "acl_draft": <string>,
        "acl_draft_v6": <string>
    },
    "network_type": {
        "id": <integer>,
        "tipo_rede": <string>
    },
    "environmentvip": {
        "id": <integer>,
        "finalidade_txt": <string>,
        "cliente_txt": <string>,
        "ambiente_p44_txt": <string>,
        "description": <string>
    },
    "active": <boolean>,
    "dhcprelay": [
```

```
                        <string>, ...
                    ],
                    "cluster_unit": <string>
                },
                "description": <string>
            },
            "equipment": {
                "id": <integer>,
                "name": <string>,
                "maintenance": <boolean>,
                "equipment_type": {
                    "id": <integer>,
                    "equipment_type": <string>
                },
                "model": {
                    "id": <integer>,
                    "name": <string>
                },
                "environments": [
                    {
                        "is_router": <boolean>,
                        "is_controller": <boolean>,
                        "environment": {
                            "id": <integer>,
                            "name": <string>,
                            "grupo_l3": <integer>,
                            "ambiente_logico": <integer>,
                            "divisao_dc": <integer>,
                            "filter": <integer>,
                            "acl_path": <string>,
                            "ipv4_template": <string>,
                            "ipv6_template": <string>,
                            "link": <string>,
                            "min_num_vlan_1": <integer>,
                            "max_num_vlan_1": <integer>,
                            "min_num_vlan_2": <integer>,
                            "max_num_vlan_2": <integer>,
                            "default_vrf": <integer>,
                            "father_environment": <recurrence-to:environment>,
                            "sdn_controllers": null
                        }
                    }, ...
                ],
                "groups": [
                    {
                        "id": <integer>,
                        "name": <string>
                    }, ...
                ],
                "id_as": {
                    "id": <integer>,
                    "name": <string>,
                    "description": <string>
                }
            }
        }, ...
    ]
}, ...
```

```
    ]
}
```

Using fields and kind together

If **fields** is being used together **kind**, only the required fields will be retrieved instead of default.

Example with details kind and id field:

```
kind=details&fields=id
```

Default behavior without kind and fields

If neither **kind** nor **fields** are used in request, the response body will look like this:

Response body:

```
{
    "virtual_interfaces": [
        {
            "id": <integer>,
            "name": <string>,
            "vrf": <integer>
        }, ...
    ]
}
```

POST

Creating list of Virtual Interfaces

URL:

```
/api/v4/virtual-interface/
```

Request body:

```
{
    "virtual_interfaces": [
        {
            "vrf": <integer>,
            "name": <string>
        }, ...
    ]
}
```

- **vrf** - You must associate one Vrf to each new Virtual Interface.
- **name** - You must assign a name to the new Virtual Interface.

URL Example:

```
/api/v4/virtual-interface/
```

PUT

Updating list of Virtual Interfaces in database

URL:

```
/api/v4/virtual-interface/[virtual_interface_ids]/
```

where **virtual_interface_ids** are the identifiers of Virtual Interfaces. It can use multiple ids separated by semicolons.

Example with Parameter IDs:

One ID:

```
/api/v4/virtual-interface/1/
```

Many IDs:

```
/api/v4/virtual-interface/1;3;8/
```

Request body:

```
{
    "virtual_interfaces": [
        {
            "id": <integer>,
            "vrf": <integer>,
            "name": <string>
        }, ...
    ]
}
```

- **id** - It's the identifier of Virtual Interface you want to edit.
- **vrf** - You must set the Vrf field maintaining actual relationship or setting another Vrf.
- **name** - You must give new name (or the same) to existing Virtual Interface.

Remember that if you don't provide the not mandatory fields, actual information (e.g. association between Virtual Interface and Vrf) will be deleted. The effect of PUT Request is always to replace actual data by what you provide into fields in this type of request.

URL Example:

```
/api/v4/virtual-interface/1/
```

DELETE

Deleting a list of Virtual Interfaces in database

Deleting list of Virtual Interfaces and all relationships URL:

```
/api/v4/equipment/[virtual_interface_ids]/
```

where **virtual_interface_ids** are the identifiers of Virtual Interfaces desired to delete. It can use multiple id's separated by semicolons. Doing this, all Neighbors that are related to this particular Virtual Interface will be deleted as well as the relationships between Equipments and IPv4's or relationships between Equipments and IPv6's containing this particular Virtual Interface.

Example with Parameter IDs:

One ID:

```
/api/v4/equipment/1/
```

Many IDs:

```
/api/v4/equipment/1;3;8/
```

Software Defined Networks

Contents

Software Defined Networks is an emerging concept. The OpenFlow protocol did the necessary work to decouple Control Plane from Data Plane.

Globo Network API takes advantage of these concepts to enable SDN based solutions. The following features are enabled through SDN:

Access Control Lists (ACLs)

Globo Network API enables deployment of *Access Control lists* on OpenVSwitch through the OpenFlow controller OpenDaylight.

To do it, Globo Network API exports HTTP urls to manage *flows* of ACLs. Using the abstraction of a *environment*, we segment the ACLs.

When a controller is inserted as a new equipment in the API we must inform for which Environment that controller belongs. This way we segment which Environment will use ACLs based on SDN.

If you run a set of OpenVSwitches and control them with the controller you should use the following HTTP Urls to manage ACLs flows inside the virtual switch:

GET

POST

PUT

DELETE

Architecture

The SDN architecture used by Network API depends on OpenDaylight controller and OpenFlow protocol.

CHAPTER 9

E-mail lists (Forums)

Users e-mail list (soon)

Developers e-mail list (soon)

CHAPTER 10

Indices and tables

- *genindex*
- *modindex*
- *search*

n

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X