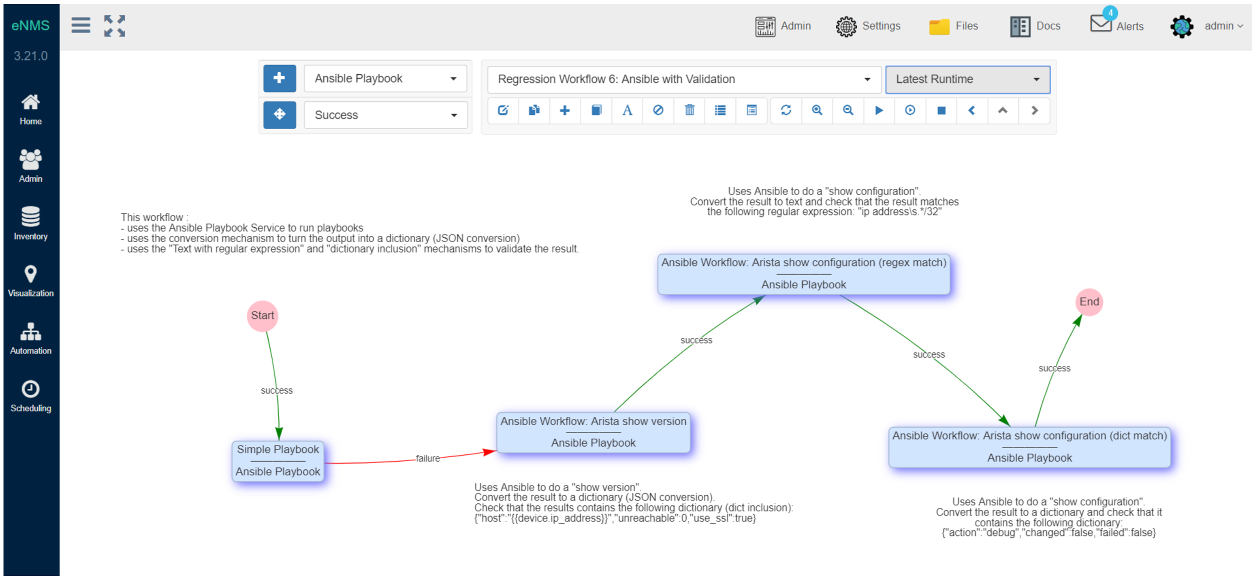
eNMS

Release 3.17.2

Sep 13, 2021

Getting started

eNMS is a vendor-agnostic NMS designed for building workflow-based network automation solutions.



This workflow :
- uses the Ansible Playbook Service to run playbooks
- uses the conversion mechanism to turn the output into a dictionary (JSON conversion)
- uses the "Text with regular expression" and "dictionary inclusion" mechanisms to validate the result.

Uses Ansible to do a "show configuration".
Convert the result to text and check that the result matches
the following regular expression: "ip address\s.*/32"

Uses Ansible to do a "show version".
Convert the result to a dictionary (JSON conversion).
Check that the results contains the following dictionary (dict inclusion):
{"host":"{{device.ip_address}}","unreachable":0,"use_ssl":true}

Uses Ansible to do a "show configuration".
Convert the result to a dictionary and check that it
contains the following dictionary:
{"action":"debug","changed":false,"failed":false}

It encompasses the following aspects of network automation:

- **Configuration Management Service**: Backup with Git, change and rollback of configurations.
- **Validation Services**: Validate data about the state of a device with Netmiko and NAPALM.
- **Ansible Service**: Store and run Ansible playbooks.
- **REST Service**: Send REST calls with variable URL and payload.
- **Python Script Service**: Any python script can be integrated into the web UI. eNMS will automatically generate a form in the UI for the script input parameters.
- **Workflows**: Services can be combined together graphically in a workflow.
- **Scheduling**: Services and workflows can be scheduled to start at a later time, or run periodically with CRON.
- **Event-driven automation**: Services and workflows can be triggered from the REST API.

CHAPTER 1

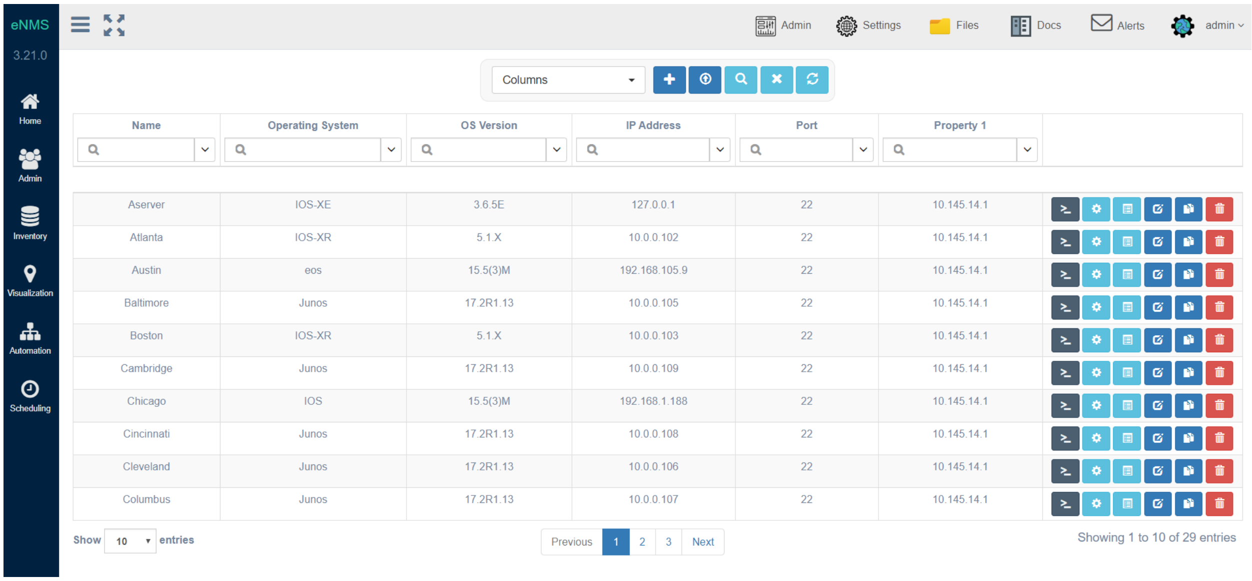
Application stack

Function	Component
HTTP Service	nginx
WSGI Service	gunicorn
Application	Flask/Python 3.6+
Database	SQLite, MySQL or PostgreSQL
Credentials storage	Hashicorp vault

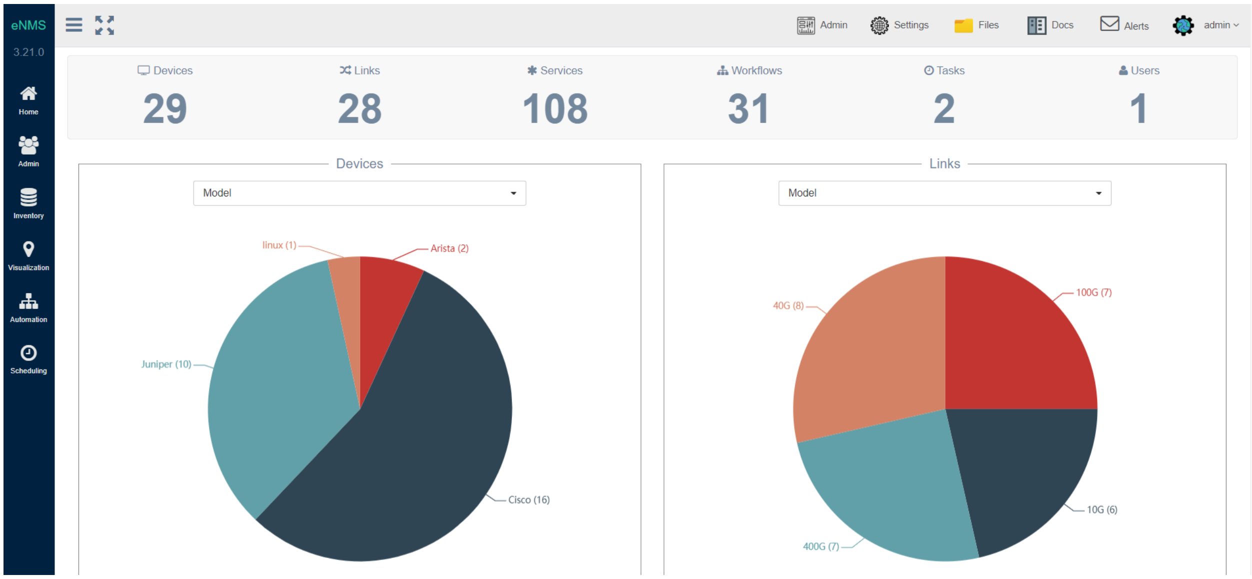
1.1 Features

1.1.1 Creation of the network

Your network topology can be created manually or imported from an external Source of Truth (OpenNMS, LibreNMS, or Netbox). Once created, it is displayed in a sortable and searchable table. A dashboard provides a graphical overview of your network with dynamic charts.

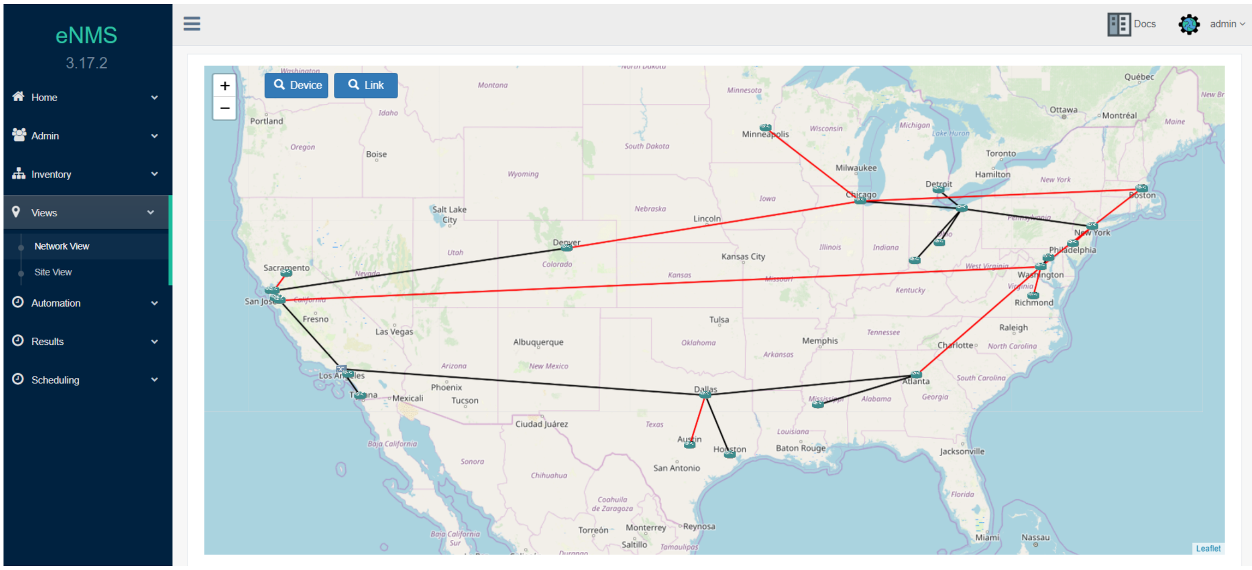


Name	Operating System	OS Version	IP Address	Port	Property 1
Aserver	IOS-XE	3.6.5E	127.0.0.1	22	10.145.14.1
Atlanta	IOS-XR	5.1.X	10.0.0.102	22	10.145.14.1
Austin	eos	15.5(3)M	192.168.105.9	22	10.145.14.1
Baltimore	Junos	17.2R1.13	10.0.0.105	22	10.145.14.1
Boston	IOS-XR	5.1.X	10.0.0.103	22	10.145.14.1
Cambridge	Junos	17.2R1.13	10.0.0.109	22	10.145.14.1
Chicago	IOS	15.5(3)M	192.168.1.188	22	10.145.14.1
Cincinnati	Junos	17.2R1.13	10.0.0.108	22	10.145.14.1
Cleveland	Junos	17.2R1.13	10.0.0.106	22	10.145.14.1
Columbus	Junos	17.2R1.13	10.0.0.107	22	10.145.14.1

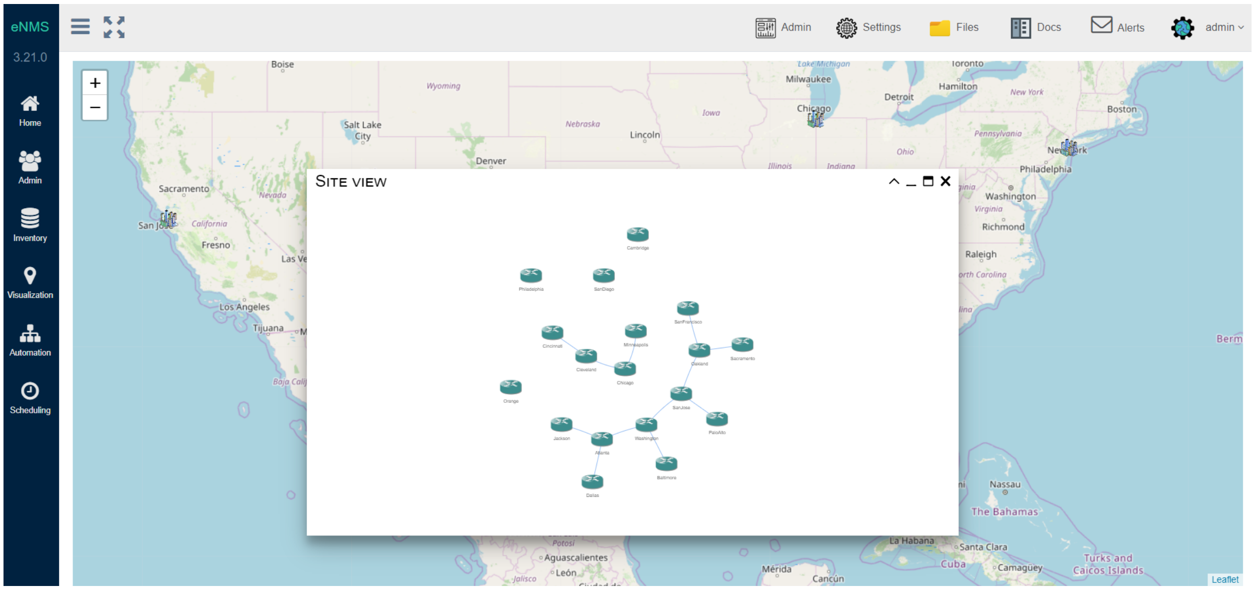


1.1.2 Network visualization

eNMS can display your network on a world map (Google Map or Open Street Map). Each device is displayed at its GPS coordinates. You can click on a device to display its properties, configuration, or start an SSH terminal session.

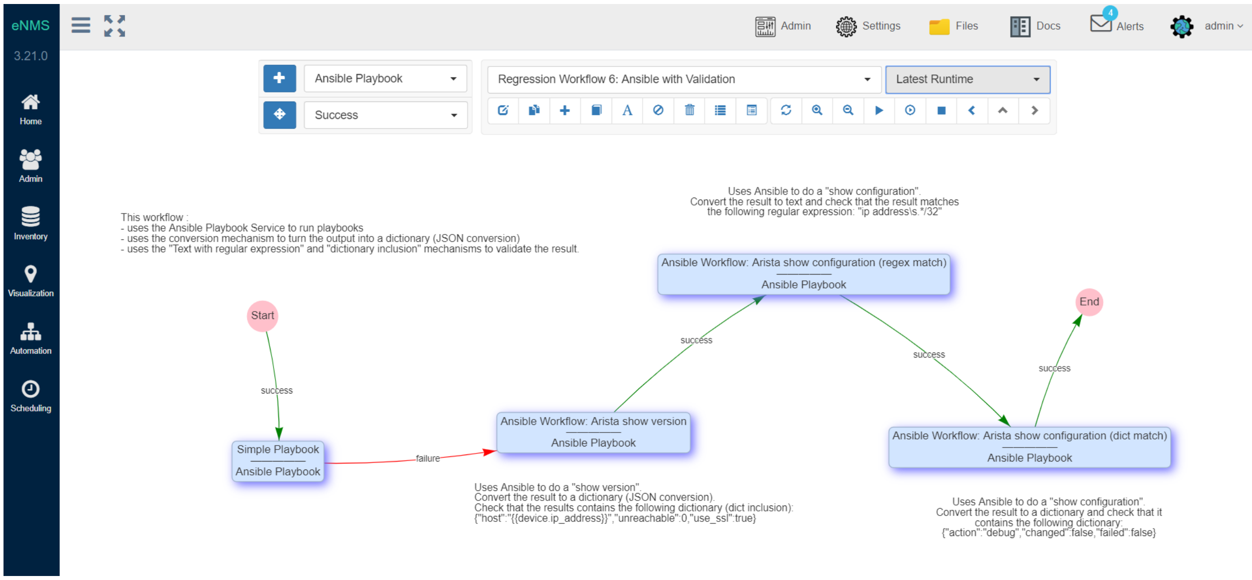


Colocated devices can be grouped into geographical sites (campus, datacenter, …), and displayed logically with a force-directed layout.



1.1.3 Workflows

Services can be combined into a workflow. When a workflow is executed, its status is updated in real-time on the web UI.



This workflow :
- uses the Ansible Playbook Service to run playbooks
- uses the conversion mechanism to turn the output into a dictionary (JSON conversion)
- uses the "Text with regular expression" and "dictionary inclusion" mechanisms to validate the result.

Uses Ansible to do a "show configuration".
Convert the result to text and check that the result matches
the following regular expression: "ip address\s.*/32"

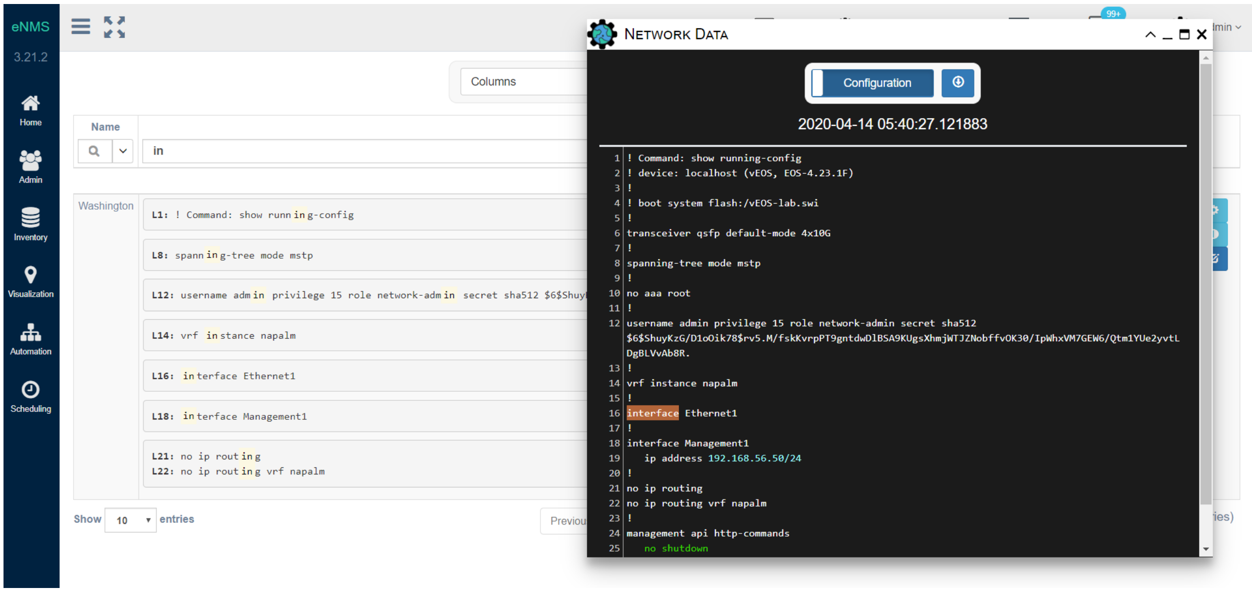
Uses Ansible to do a "show version".
Convert the result to a dictionary (JSON conversion).
Check that the results contains the following dictionary (dict inclusion):
{"host":"{{device.ip_address}}","unreachable":0,"use_ssl":true}

Uses Ansible to do a "show configuration".
Convert the result to a dictionary and check that it
contains the following dictionary:
{"action":"debug","changed":false,"failed":false}

1.1.4 Configuration Management

eNMS can work as a network device configuration backup tool and replace Oxidized/Rancid with the following features:

- Poll network elements and store the latest configuration in the database.
- Search for any text or regular-expression in all configurations.
- Download device configuration to a local text file.
- Use the REST API support to return a specified device's configuration.
- Export all configurations to a remote Git repository (e.g. Gitlab)
- View git-style differences between various revisions of a configuration



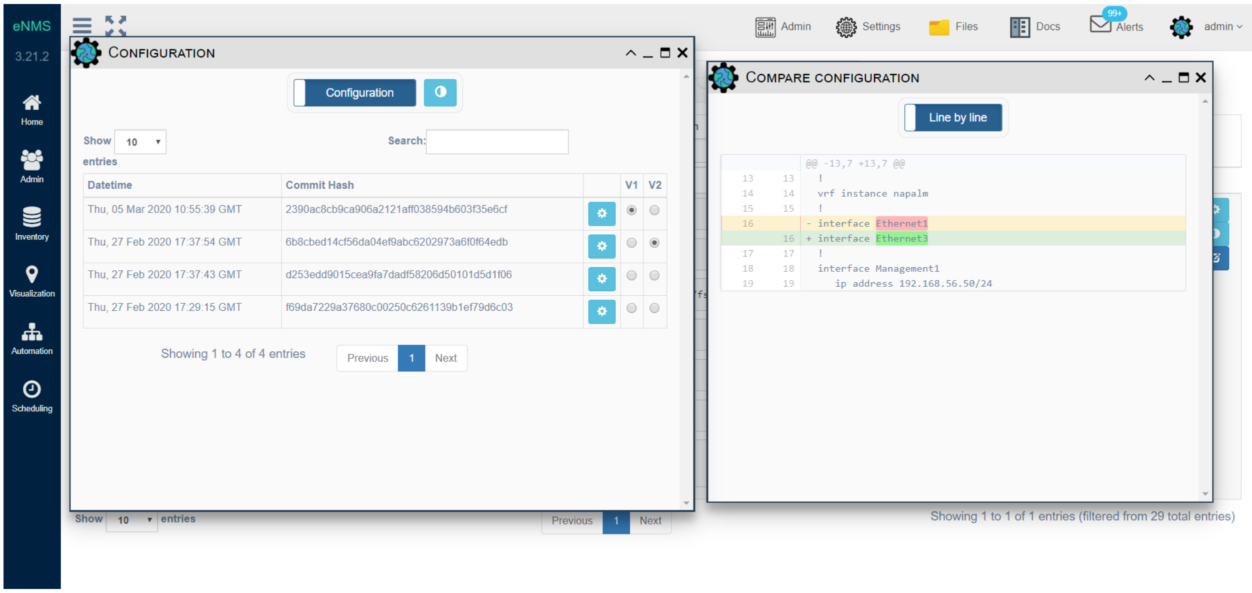
Name: in

Washington

```
L1: ! Command: show running-config
L8: spanning-tree mode mstp
L12: username admin privilege 15 role network-admin secret sha512 $6$Shuy
L14: vrf instance napalm
L16: interface Ethernet1
L18: interface Management1
L21: no ip routing
L22: no ip routing vrf napalm
```

2020-04-14 05:40:27.121883

```
! Command: show running-config
! device: localhost (vEOS, EOS-4.23.1F)
!
! boot system flash:/vEOS-lab.swi
!
transceiver qsfp default-mode 4x10G
!
spanning-tree mode mstp
!
no aaa root
!
username admin privilege 15 role network-admin secret sha512
$6$ShuyKzG/D1oOik78$rv5.M/fskKvrpPT9gntdwDlBSA9KUgsXhmjWTJZNobffvOK30/IpWhxVM7GEW6/Qtm1YUe2yvtL
DgBLVvAb8R.
!
vrf instance napalm
!
interface Ethernet1
!
interface Management1
   ip address 192.168.56.50/24
!
no ip routing
no ip routing vrf napalm
!
management api http-commands
   no shutdown
```



Show 10 entries Search:

Datetime	Commit Hash		V1	V2
Thu, 05 Mar 2020 10:55:39 GMT	2390ac8cb9ca906a2121aff038594b603f35e6cf	⚙	●	○
Thu, 27 Feb 2020 17:37:54 GMT	6b8cbed14cf56da04ef9abc6202973a6f0f64edb	⚙	○	●
Thu, 27 Feb 2020 17:37:43 GMT	d253edd9015cea9fa7dadf58206d50101d5d1f06	⚙	○	○
Thu, 27 Feb 2020 17:29:15 GMT	f69da7229a37680c00250c6261139b1ef79d6c03	⚙	○	○

Showing 1 to 4 of 4 entries Previous 1 Next

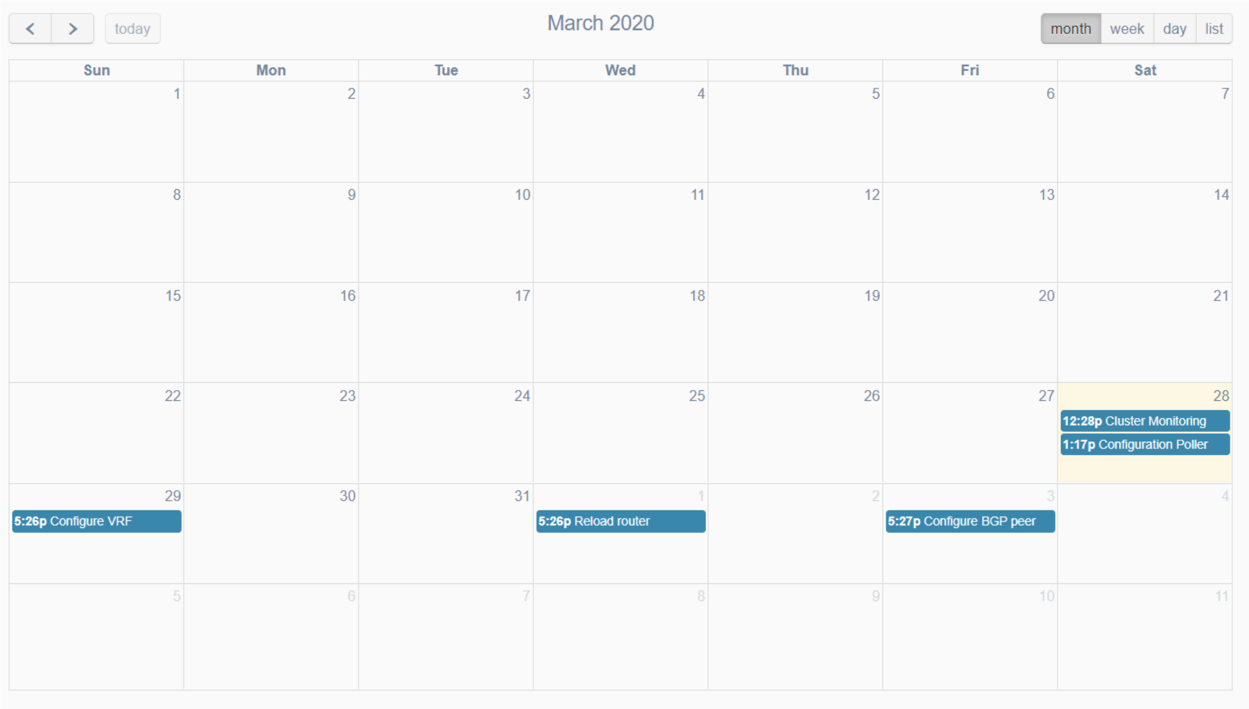
```
            @@ -13,7 +13,7 @@
13    13      !
14    14      vrf instance napalm
15    15      !
16          - interface Ethernet1
      16    + interface Ethernet3
17    17      !
18    18      interface Management1
19    19         ip address 192.168.56.50/24
```

Show 10 entries Previous 1 Next

Showing 1 to 1 of 1 entries (filtered from 29 total entries)

1.1.5 Event-driven automation

While services can be run directly and immediately from the UI, you can also schedule them to run at a later time, or periodically by defining a frequency or a CRON expression. All scheduled tasks are displayed in a calendar.



March 2020

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28 **12:28p** Cluster Monitoring **1:17p** Configuration Poller
29 **5:26p** Configure VRF	30	31	1 **5:26p** Reload router	2	3 **5:27p** Configure BGP peer	4
5	6	7	8	9	10	11

Services can also be executed programmatically: eNMS has a REST API and a CLI interface that can be used to create, update and delete any type of objects, but also to trigger the execution of a service.

1.2 Installation

eNMS is a Flask web application designed to run on a **Unix server** with Python **3.6+**.

1.2.1 First steps

The first step is to download the application. You can download the latest release of eNMS directly from your browser, by going to the Release section of eNMS github repository.

The other option is to clone the master branch of the git repository from github:

```
# download the code from github:
git clone https://github.com/afourmy/eNMS.git
cd eNMS
```

Once the application is installed, you must go to the *eNMS* folder and install eNMS python depedencies:

```
# install the requirements:
pip install -r build/requirements/requirements.txt
```

Once the requirements have been installed, you can run the application with Flask built-in development server.

```
# set the FLASK_APP environment variable
export FLASK_APP=app.py

# start the application with Flask
flask run --host=0.0.0.0
```

1.2.2 Production mode

Database

By default, eNMS will use an SQLite database (*sqlite:///database.db*). You can configure a different database with the *DATABASE_URL* environment variable (see SQL Alchemy database URL)

For example, for a MySQL database, the variable could be:

```
export DATABASE_URL="mysql://root:password@localhost/enms"
```

Secret key

You need to configure the secret key used by Flask to sign sessions

```
# set the SECRET_KEY environment variable
export SECRET_KEY=value-of-your-secret-key
```

WSGI server

You must use a WSGI HTTP server like gunicorn to run eNMS instead of Flask development server.

You can find a configuration file for gunicorn in the main folder (*gunicorn.py*), and run the application with the following command:

```
# start the application with gunicorn
gunicorn --config gunicorn.py app:app
```

Hashicorp Vault

All credentials should be stored in a Hashicorp Vault: the settings variable `active` under the `vault` section of the settings tells eNMS that a Vault has been setup and can be used. Follow the manufacturer's instructions and options for how to setup a Hashicorp Vault.

You must tell eNMS how to connect to the Vault with

- the `VAULT_ADDRESS` environment variable
- the `VAULT_TOKEN` environment variable

```
# set the VAULT_ADDRESS environment variable
export VAULT_ADDRESS=url of the vault
# set the VAULT_TOKEN environment variable
export VAULT_TOKEN=vault-token
```

eNMS can also unseal the Vault automatically at start time. This mechanism is disabled by default. To activate it, you need to: - set the `unseal` settings variable to `true` - set the UNSEAL_VAULT_KEYx (`x` in [1, 5]) environment variables :

```
export UNSEAL_VAULT_KEY1=key1
export UNSEAL_VAULT_KEY2=key2
etc
```

1.2.3 Settings

The `/setup/settings.json` file includes:

- Environment variables for all sensitive data (passwords, tokens, keys). Environment variables are exported from Unix with the `export` keyword: `export VARIABLE_NAME=value`. Environment variables include:

```
- SECRET_KEY = secret_key
- MAIL_PASSWORD = mail_password
- TACACS_PASSWORD = tacacs_password
- SLACK_TOKEN = slack_token
```

- Public variables defined in the `settings.json` file, and later modifiable from the administration panel. Note that changing settings from the administration panel do not currently cause the settings.json file to be rewritten.

Settings `app` section

- `address` (default: `""`) The address is needed when eNMS needs to provide a link back to the application, which is the case with GoTTY and mail notifications. When left empty, eNMS will try to guess the URL. This might not work consistently depending on your environment (nginx configuration, proxy, . . .)
- `config_mode` (default: `"debug"`) Must be set to “debug” or “production”.
- `startup_migration` (default: `"examples"`) Name of the migration to load when eNMS starts for the first time.
  - By default, when eNMS loads for the first time, it will create a network topology and a number of services and workflows as examples of what you can do.
  - You can set the migration to `"default"` instead, in which case eNMS will only load what is required for the application to function properly.
- `documentation_url` (default: `"https://enms.readthedocs.io/en/latest/"`) Can be changed if you want to host your own version of the documentation locally. Points to the online documentation by default.
- `git_repository` (default: `""`) Git is used as a version control system for device configurations: this variable is the address of the remote git repository where eNMS will push all device configurations.

Settings `cluster` section

- `active` (default: `false`)
- `id` (default: `true`)
- `scan_subnet` (default: `"192.168.105.0/24"`)
- `scan_protocol` (default: `"http"`)
- `scan_timeout` (default: `0.05`)

Settings `database` section

- `pool_size` (default: `1000`) Number of connections kept persistently in SQL Alchemy pool.
- `max_overflow` (default: `10`) Maximum overflow size of the connection pool.
- `tiny_string_length` (default: `64`) Length of a tiny string in the database.

- `small_string_length` (default: `255`) Length of a small string in the database.
- `small_string_length` (default: `32768`) Length of a large string in the database.

Settings `ldap` section

If LDAP/Active Directory is enabled and the user doesn't exist in the database yet, eNMS tries to authenticate against LDAP/AD using the *ldap3* library, and if successful, that user gets added to eNMS locally.

- `active` (default: `false`) Enable LDAP authentication.
- `server` (default: `"ldap://domain.ad.company.com"`) LDAP Server URL (also called LDAP Provider URL)
- `userdn` (default: `"domain.ad.company.com"`) LDAP Distinguished Name (DN) for the user
- `basedn` (default: `"DC=domain,DC=ad,DC=company,DC=com"`) LDAP base distinguished name subtree that is used when searching for user entries on the LDAP server. Use LDAP Data Interchange Format (LDIF) syntax for the entries.
- `admin_group` (default: `"eNMS.Users,network.Admins"`) string to match against 'memberOf' attributes of the matched user to determine if the user is allowed to log in.

Note: Failure to match memberOf attribute output against `admin_group` results in a valid ldap user within the `basedn` being denied access on login. If a memberOf attribute matches the `admin_group`, they will be given Admin permissions.

Note: eNMS does not store the credentials of LDAP and TACACS users; however, those users are listed in the Admin / Users panel.

Settings `mail` section

- `server` (default: `"smtp.googlemail.com"`)
- `port` (default: `587`)
- `use_tls` (default: `true`)
- `username` (default: `"eNMS-user"`)
- `sender` (default: `"eNMS@company.com"`)

Settings `mattermost` section

- `url` (default: `"https://mattermost.company.com/hooks/i1phfh6fxjfwpy586bwqq5sk8w"`)
- `channel` (default: `""`)
- `verify_certificate` (default: `true`)

Settings `paths` section

- `files` (default:`""`) Path to eNMS managed files needed by services and workflows. For example, files to upload to devices.
- `custom_code` (default: `""`) Path to custom libraries that can be utilized within services and workflows
- `custom_services` (default: `""`) Path to a folder that contains *Custom Services*. These services are added to the list of existing services in the Automation Panel when building services and workflows.
- `playbooks` (default: `""`) Path to where Ansible playbooks are stored so that they are choosable in the Ansible Playbook service.

Settings `requests` section

Allows for tuning of the Python Requests library internal structures for connection pooling. Tuning these might be necessary depending on the load on eNMS.

- Pool
  - `pool_maxsize` (default: `10`)
  - `pool_connections` (default: `100`)
  - `pool_block` (default: `false`)
- Retries
  - `total` (default: `2`)
  - `read` (default: `2`)
  - `connect` (default: `2`)
  - `backoff_factor` (default: `0.5`)

Settings `security` section

- `hash_user_passwords` (default: `true`) All user passwords are automatically hashed by default.
- `forbidden_python_libraries` (default: `["eNMS","os","subprocess","sys"]`) There are a number of places in the UI where the user is allowed to run custom python scripts. You can configure which python libraries cannot be imported for security reasons.

Settings `slack` section

- `channel` (default: `""`)

Settings `ssh` section

- `port_redirection` (default: `false`)
- `bypass_key_prompt` (default: `true`)
- `port` (default: `-1`)
- `start_port` (default: `9000`)
- `end_port` (default: `91000`)

Settings `tacacs` section

- `active` (default: `false`)
- `address` (default: `""`)

Settings `vault` section

For eNMS to use a Vault to store all sensitive data (user and network credentials), you must set the `active` variable to `true`, provide an address and export

Public variables

- `active` (default: `false`)
- `unseal` (default: `false`) Automatically unseal the Vault. You must export the keys as environment variables.

Environment variables

- `VAULT_ADDRESS`
- `VAULT_TOKEN`
- `UNSEAL_VAULT_KEY1`
- `UNSEAL_VAULT_KEY2`
- `UNSEAL_VAULT_KEY3`
- `UNSEAL_VAULT_KEY4`
- `UNSEAL_VAULT_KEY5`

Settings `view` section

Controls the default view for where the map is initially displayed in the Visualization panels

- `longitude` (default: `-96.0`)
- `latitude` (default: `33.0`)
- `zoom_level` (default: `5`)
- `tile_layer` (default: `"osm"`)
- `marker` (default: `"Image"`)

1.2.4 Logging file

Logging settings exist in separate file: `/setup/logging.json`. This file is directly passed into the Python Logging library, so it uses the Python3 logger file configuration syntax for your version of Python3. Using this file, the administrator can configure additional loggers and logger destinations as needed for workflows.

By default, the two loggers are configured:

- The default logger has handlers for sending logs to the stdout console as well as a rotating log file `logs/enms.log`
- A security logger captures logs for: User A ran Service/Workflow B on Devices [C,D,E. . .] to log file `logs/security.log`

And these can be reconfigured here to forward through syslog to remote collection if desired.

Additionally, the `external loggers` section allows for changing the log levels for the various libraries used by eNMS.

With multiple gunicorn workers, please consider:

- Using `Python WatchedFileHandler` instead of the `RotatingFileHandler`
- Configuring the LINUX `logrotate` utility to perform the desired log rotation

```
{
"handlers": {
  "rotation": {
    "level": "DEBUG",
    "formatter": "standard",
    "filename": "logs/enms.log",
    "class": "logging.handlers.WatchedFileHandler"
  }
}
```

1.2.5 Properties file

The `/setup/properties.json` file includes:

1. Allowing for additional custom properties to be defined in eNMS for devices. In this way, eNMS device inventory can be extended to include additional columns/fields
2. Allowing for additional custom parameters to be added to services and workflows
3. Controlling which parameters and widgets can be seen from the Dashboard
4. Controlling which column/field properties are visible in the tables for device and link inventory, configuration, pools, as well as the service, results, and task browsers

properties.json custom device addition example:

- **Keys under `{"custom": { "device": {`**
    - name the custom attribute being added.
    - **Keys/Value pairs under the newly added custom device attribute device_status.**
        - “pretty_name”:”Default Username”, *device attribute name to be displayed in UI*
        - “type”:”string”, *data type of attribute*
        - “default”:”None”, *default value of attribute*
        - “private”: true *optional - is attribute hidden from user*
        - “configuration”: true *optional* - creates a custom ‘Inventory/Configurations’ attribute
        - “log_change” false *optional* - disables logging when a changes is made to attribute
        - “form”: false *optional* - disables option to edit attribute in Device User Interface
        - “migrate”: fasle *optional* - choose if attribute should be consdered for migration
        - “serialize”: false *optional* - whether it is passed to the front-end when the object itself is

- **Keys under `"tables" : { "device" : [ { & "tables" : { "configuration" : [ {`**
    - Details which attributes to display in these table, add custom attributes here
    - **Keys/Value pairs for tables**
        - "data":"device_status", *attribute created in custom device above*
        - "title":"Device Status", *name to display in table*
        - "search":"text", *search type*
        - "width":"80%", *optional - text alignment, other example "width":"130px",*
        - "visible":false, *default display option*
        - "orderable": false *allow user to order by this attribute*
- **Values under `"filtering" : { "device" : [`**
    - details which attributes to use for filtering
    - you will need to add any custom device attributes name to this list for filtering

1.2.6 RBAC file

The `/setup/rbac.json` file allows configuration of:

- Which user roles have access to each of the controls in the UI
- Which user roles have access to each of the REST API endpoints

1.2.7 Environment variables

- SECRET_KEY=secret_key
- MAIL_PASSWORD=mail_password
- TACACS_PASSWORD=tacacs_password
- SLACK_TOKEN=slack_token

1.2.8 Scheduler

The scheduler used for running tasks at a later time is a web application that is distinct from eNMS. It can be installed on the same server as eNMS, or a remote server.

Before running the scheduler, you must configure the following environment variables so it knows where eNMS is located and what credentials to authenticate with:

- `ENMS_ADDR`: URL of the remote server (example: `"http://192.168.56.102"`)
- `ENMS_USER`: eNMS login
- `ENMS_PASSWORD`: eNMS password

The scheduler is a asynchronous application that must be deployed with uvicorn :

```
cd scheduler
uvicorn scheduler:scheduler --host 0.0.0.0
```

1.3 Release Notes

1.3.1 Version 4.1.0

- 3D Logical visualization
- Remove Event Model and Syslog server
- Refactor of the run mechanism. When running a service, a single run is created and saved to the

database. - Remove "operation" (any / all) property from pool - Change the way pool objects are computed: via SQL query instead of pure python: better performances expected for large pools. - Add regex support for SQLite - Add new "Invert" option for table filtering - Move terminal application for web SSH feature inside the application: the terminal application was previously moved outside the application because websockets requires sticky sessions which is incompatible with having multiple gunicorn workers. Moving to a deployment where eNMS is started multiple times with 1 gunicorn worker via the backend stream configuration, it is now possible for the terminal to be inside the main application. - Refactoring of the REST API * all requests are handled by the same "monitor requests" function * remove dependency to flask_restful and flask_httpauth - Fix submenu bug when the menu is minimized (gentelella bug) - Replace prerequisite edge with priority mechanism - Allow making non-shared service shared and vice-versa (if the shared service doesn't have more than one workflow). - Separate progress for main devices & iteration devices in workflow builder - Fix bug where subworkflow device counters not displayed in results when device iteration is used Bug report mail: "No status for services in subworkflow with device iteration" - HTTP requests logging: all requests are now logged by eNMS and not by werkzeug like before. => fine grained controlled for what is logged for each request. The log now contains the username. - Add duplicate button in service table - Refactor the geographical and Logical View to behave like the workflow builder: * List of all pools that contain at least one device or link, stored in user browser local storage * Remove default pool mechanism. Remove "visualization_default" property in pool model. By design, the default pool becomes the first pool in alphabetical order * Add backward / forward control like the workflow builder - Rename "monitor_requests" function to "process_requests": impact on plugins - Add global "factory" and "delete" functions in the workflow builder to create and delete new objects from a workflow. - When refreshing a pool, rbac is now ignored so that the pool "refresh" action result does not depend on the user triggering it. - If a workflow is defined to run on a set of devices, and the user lacks access to one or more devices, execute for all accessible devices and fail for the inaccessible devices instead of failing the entire workflow. - app.service_db was renamed to "service_run_count" and it no longer has an inner "runs" key: the gunicorn auto safe restart code that uses it must be updated accordingly. - Store and commit web SSH session content in backend instead of relying on send beacon mechanism and onbeforeunload callback so that the saving of a session does not depend on user behavior - Refactoring of the forms: all forms are now in eNMS.forms.py. Impact on form import: eNMS.forms.automation -> eNMS.forms - Refactoring of the setup file: replace "from eNMS.setup" with "from eNMS.variables" - Change model_properties in model from list of properties to dict of property with associated type - Custom properties defined in properties.json: change type from "boolean" to "bool" and "string" to "str" for consistency with rest of codebase - Add "parent_service" property to retrieve all results from a workflow, including subworkflow service results (see "Re: [E] Re: Retrieving results via REST"). The parent service is the service corresponding to the "parent runtime property". Example of /search payload: {

> "type": "result", "columns": ["result", "parent_runtime", "service_name"], "maximum_return_records": 1000, "search_criteria": {"parent_runtime": "2021-04-19 04:09:05.424206", "parent_service": "A"}

} - Add new "Empty" option in table filters and pool definition to filter based on whether the property value is empty or not. - Add table display with property value constraint when clicking on the charts in the dashboard. - Add scrapli netconf service - Move LDAP and TACACS+ server init to environment file instead of custom file. Impact on authentication ldap / tacacs functions. - Add Token-based authentication via REST API. New GET endpoint "/rest/token" to generate a token. - Separate controller (handling HTTP POST requests) from main application (gluing everything together) Impact: * In plugins, * the "custom" file that contains pre_init, post_init, and the authentication custom code no longer inherits from the controller - Add new "ip_address" field in settings.json > app section - Add paging for REST API search endpoint: new integer parameter "start" to request results from "start" - Add server time at the bottom of the menu (e.g for scheduling tasks / ease of use) - Add button in service table to export services in bulk (export all displayed services as .tgz) - Ability to paste device list (comma or space separated) into a multiple instance field (e.g service device and pool targets) - Re-add current Run counter to 'Service' and 'Workflow' on the dashboard banner + Active tasks - Ability to download result as json file + new copy result path to clipboard button in result json editor panel - Ability to download logs as text file - When importing existing workflows via service import, remove all existing services and edges from the workflow - Upload service from laptop instead of checking for file on the VM - Add Initial Form mechanism to update run properties and payload. - Add new "full results" button to results tree

MIGRATION:

- In all services,

def job(self, run, payload, device): -> def job(self, run, device): - Check that all "operator" property in pool.yaml are set to "all" - In all plugins, "monitor_requests" should be renamed to "process_requests" and "register_endpoint" should be renamed "_register_endpoint" - "export_topology" endpoint should be renamed "topology_export" in migration files - Replace all "from eNMS.forms.automation" with "from eNMS.forms" - Rename "run_db" to "run_states" - Plugin: add env and vs argument - Change model_properties in model from list of properties to dict of property with associated type - Custom properties defined in properties.json: change type from "boolean" to "bool" and "string" to "str" for consistency with rest of codebase. In properties.json: * change "type": "boolean" to "type": "bool" * change "type": "string" to "type": "str" - Update authentication functions in custom.py

To be tested: - Refactoring of the run mechanism: * log level mechanism * workflow run, progress display in workflow builder * parameterized run * service value and device iteration * service / workflow performances with large number of devices * workflow: all run modes / SxS & DxD * results display: automation / results page, workflow tree, result table, device result table * get_var / set_var and get_result mechanism, including get_result along with restart mechanism - Pool refactoring * dynamic pool content, inclusion / equality / regex * pool individual and global refresh mechanism in pool table - Conversion shared - non shared for a service. Duplication of shared / non-shared service. - New global "factory" and "delete" functions

1.3.2 Version 4.0.1

- Don't update pool during migration import
- Add scalability migration files
- Remove "All", "None" and "Unrelated" options in relationship filtering
- Use join instead of subqueries to improve relationship filtering scalability
- Add form endpoints in rbac files when instantiating custom services
- Fix changelog like pool update not logged bug
- Fix workflow tree mechanism from workflow with superworkflow bug
- Change of all GET endpoints to no longer contain backslash:
- renaming /table/{type} to {type}_table
- renaming of /form/{form_type} to "{form_type}_form

Everything that comes after backslash is considered to be an argument (*args) - Change of rbac.json structure: list becomes dict, each line can have one of three values: * "admin" (not part of RBAC, only admin have access, e.g admin panel, migration etc) * "all" (not part of RBAC, everyone has access, e.g dashboard, login, logout etc) * "access" (access restricted by RBAC, used to populate access form) Impact on plugins: the settings.json "rbac" section has to be updated accordingly. - Add RBAC support for nested submenus

Impact of RBAC on plugins: - plugins must be mounted at "/", custom "url_prefix" are no longer working… changes:

  - server.register_blueprint(blueprint, url_prefix=kwargs["url_prefix"])
  - server.register_blueprint(blueprint)

- **need for new argument in settings.json "blueprint" section: "static_url_path". changes:**

    "blueprint": { "template_folder": "templates", "static_folder": "static"

  - "static_url_path": "/template-static"

    },

- plugins endpoints cannot contain a slash.

1.3.3 Version 4.0.0

- Extend pool for users and services.
- Add relation mechanism in table for scalability * For each table, add link to relation table * Replaces the old "Pool Objects" window in the pool table. * New mechanism to add instances to a "relation table", both by individual selection and in bulk by copy pasting a list of names. * New mechanism to remove selection from a relation table.
- Add "run service on targets mechanism" * run service on a single device and in bulk from service page * run service on a single device and in bulk from visualization pages
- Add bulk deletion and bulk edit mechanism for tables * Bulk edit (edit all instances filtered in tables) * Bulk deletion (delete all instances filtered in tables)
- Add "copy to clipboard" mechanism to get comma-separated list of names of all filtered instances.
- Add 3D network view and 3D Logical View. * Add right click menu for property, configuration, run service * Add default pools mechanism for large networks. * Add run service in bulk on all currently displayed devices mechanism
- Move all visualization settings from settings.json > "visualization" to dedicated visualization.json
- Make the error page colors confiurable per theme (move css colors to theme specific CSS file)
- Use the log level of the parameterized run instead of always using the service log level
- Change field syntax for context help to be 'help="path"' instead of using render_kw={"help": . . . }
- Don't update the "creator" field when an existing object is edited
- Add new function "get_neighbors" to retrieve neighboring devices or links of a device
- Refactor the migration import mechanism to better handle class relationships
- Web / Desktop connection to a device is now restrictable to make the users provide their own credentials

=> e.g to prevent inventory device credentials from being used to connect to devices - Configuration git diff: indicate which is V1 and which is V2. Option to display more context lines, including all of it. - Improve display of Json property in form (make them collapsed by default) - Update to new version of Vis.Js (potential workflow builder impact) - Add mechanism to save only failed results (e.g for config collection workflow) - New database.json to define engine parameters, import / export properties, many to many relationship, etc. - Fork based on string value instead of just True / False: new discard mode for the skip mechanism. When using discard, devices do not follow any edge after the skipped service. - Refactor skip property so that it is no longer a property of the service to avoid side effect of skipping shared services. - Add new option in pool to invert logic for each property. - New Option "Update pools after running" for workflow like the configuration management workflow. - Refactor skip mechanism to work with run once mode service. - Don't reset run status when running a CLI command with CLI plugins - Refactor log mechanism to send log to client bit by bit, instead of all run logs at each refresh request - "No validation" in the service panel is now an option of the "validation condition" instead of the "validation method". Migration impact. - The timestamps like "last runtime", "last failure", etc are now per configuration property. The timestamps are all stored per device in a json.file called "timestamps.json". These timestamps properties have been added to the configuration table. - Add ability to hard-code logic to mask password hashes when config is displayed in custom controller. - Add workflow tree in the workflow builder to visualize workflow and subworkflows as a tree with buttons: edit / new mechanism: highlight to teleport to any service. Makes it easier to work with large multi-level workflows. - Replace gotty with pure python implementation. Save session output with webssh. Need to set ENMS_USER and ENMS_PASSWORD like with the scheduler to save the session via REST API. For this to work, admin credentials must be defined via two new environment variables: ENMS_USER and ENMS_PASSWORD (same as scheduler) - Fix bug connection not cached when using iteration values with a standalone service - Fix bug when exporting table to .csv - column shift if comma in property value - When scheduling a task, the creator of the service run is not properly set to the user who scheduled the task instead of the admin user. - Add a cap for number of threads when running a service with multiprocessing enabled. Maximum number of threads configurable from settings.json > automation > max process. - Add runtimes select list in service results window, so you can visualize service results in workflow builder. - Include private properties (custom password, …) when exporting a service, or migration files. - New color property for workflow edges. - Export service now exports to user browser besides exporting the tgz to the VM. - Remove Create Pool endpoint in the rest API - Add python snippet mechanism to troubleshooting (ctrl + alt + click on upper left logo) - Refactor REST service in case status code is not in (200, 300) to fix validation bug - Refactoring of the rbac system:

  - Use pools extension to user and services to define user access.
  - Add new "default access" property to choose between creator, admin, and public
  - Remove "group" table (a group is a pool of users)
  - Add "groups" property to user and add "creator" property for pools, devices and links. By defining pools of services

  based on the group property, and autofilling the groups property of an object when it is created with the user "groups", objects can be automatically added to the pool of services of the appropriate groups.

- New Credentials mechanism: * Credentials can be either username / password or SSH key. Both passwords and SSH key are stored in the Vault (no key file stored on the unix server). * Credentials also have an "Enable Password" field to go to enable mode after logging in. * Credentials have a priority field; credential object with higher priority is used if multiple available credentials. * Credentials have two pools: user pool to define which users can use the credentials, and device pools to define which devices the credential can be used for. * User "groups" property is now a field. This field can be used to define user pools. Services have the same "groups" property. When creating a new service, the groups field will be automatically set to the user groups. This allows services to be automatically added to the appriopriate pool of services, if the pool of services is defined based on that group property. * Credentials can be either "Read - Write" (default) or "Read only". In a top-level service, new "credential type" field to choose between "Any", "Read-only" and "Read-write" in order to define which credentials should be used when running the service.
- The skip values were renamed from "True" / "False" to "Success" / "Failure".

Test: - test new bulk edit, bulk delete, copy clipboard mechanism - test new relation table mechanism with add to relation (individual and bulk selection) and remove from relation. - test new logical and geographical views (right-click menu, scalability with 10K+ devices, default pools mechanism, network filtering mechanism, run service mechanism, etc) - test new get_neighbors function, including using get_neighbors output for service iteration - test that notification mechanism still works - test that the new web SSH mechanism works, make sure that the session saving mechanism works as intended. - test that the workflow mechanism in both DxD and SxS still works: the workflow algorithm was refactored and

  DxD / SxS now uses the same function.

- test the skip mechanism: * test skip of shared service only affects workflow from which service is skipped * test new discard option * test that skip works fine with services in "run once" mode.
- test the iteration mechanism (both iteration on value and iteration on devices). Tests that the connection

is cached and reused for iteration values. - test the device query mechanism. - user rbac (access to UI + access to models) is properly updated when one of its associated pool OR access is modified. - test new credentials mechanism - test new option in pool to invert logic - test new "update pools after running mechanism" - test that service logs works properly (was refactored from scratch) - test new "per configuration property timestamp" mechanism for configuration management mechanism. - test new mechanism to mask passwords when displaying configuration via custom controller function - test export table to csv mechanism - when a service is renamed, the custom password still works. - test that connections are cached when using iteration values on standalone service. - test that when scheduling task, run creator is set to user who scheduled task. - test new "maximum number of thread" mechanism - test new troubleshooting snippet mechanism - test performances and scalability compared to last version (no improvements to be expected as no work as made on performances, but we have to make sure it's not worse). - test rest call services as the rest service was refactored.

Migration: - Update endpoint: view/network and view/site no longer exists, to be replaced with geographical_view and view_builder - Configure the new visualization.json file, remove visualization settings from settings.json - In the service.yaml file, the "devices" and "pools" relationship with services have to be renamed "target_devices" and "target_pools". Besides, "update_pools" must be renamed to "update_target_pools". - In service.yaml, remove the skip property: it will not be migrated (refactoring of skip mechanism so that skip is per workflow and not a property of the service itself) - In service.yaml, "No Validation" is now part of the "Validation Condition" section. This means that all services where "validation_method" is set to "none", it must be replaced with "text" and "validation_condition" must be set to "none" instead. - Add ENMS_USER and ENMS_PASSWORD (admin credentials) to environment variables. - The create_pool endpoint has been removed, make sure the /instance/pool endpoint is used instead. - The Rest service has been refactored in case the response is not in range 200 - 300: the "response_code" key is now "status_code", and "response" key becomes "result" (consistent with the case where the rest call is successful). Need to check these keys in the migration files, i.e for services that use these keys as part of the post-processing or as part of the workflow later one. - Whenever the "Use host keys" option is used, need to create a credential object instead with the key. The "Use host key" option in all connection services no longer exists. - In service.yaml, the "skip_value" property is "success" / "failure" instead of "True" / "False" (skip_value: 'True' -> skip_value: 'success' / skip_value: 'False' -> skip_value: 'failure') - In service.yaml, all references to devices via "self.devices" must use "self.target_devices" instead as the row was renamed in the Service table.

1.3.4 Version 3.22.4

- Catch exception in log function when fetching log level from database
- Fix object numbers not updated for manually defined pool
- Catch exception in query rest endpoint when no results found to avoid stacktrace in server logs
- Add "fetch" and "fetch_all" function to workflow global space. Set rbac to "edit" and username to current user

for both these functions. - Add "encrypt" function to workflow global space to encrypt password and use substitution in custom passwords. - Return json object in get result REST endpoint when no results found for consistency. - Reset service status to "Idle" when reloading the app along with the run status.

1.3.5 Version 3.22.3

- Add regression workflow for file transfer
- Fix RBAC service run and task scheduling REST API bug
- Fix payload extraction workflow __setitem__ bug
- Add regression workflow with lots of service for scalability testing
- Add regression workflow for skipped service in workflow targets SxS run mode
- Fix rest call service local() scope bug
- Fix get var / set var "devices" keyword bug
- Add jump on connect parameters for netmiko backup service
- Fix skipped query with device in service by service with workflow targets mode bug

1.3.6 Version 3.22.2

- Fix iteration device factory commit bug
- Fix workflow in service by service with workflow targets skipped service bug
- Add missing rbac endpoints in full + read only access
- Fix device creation empty driver due to Scrapli
- Fix workflow iteration mechanism bug
- Fix workflow skip query bug

1.3.7 Version 3.22.1

- Add user authentication method in user forms
- Fix settings saving mechanism
- Fix gunicorn multiple workers sqlalchemy post fork session conflict bug
- Dont prevent wrong device GPS coordinates from displaying links in network view
- Fix RBAC bugs
- Add new Scrapli service to send commands / configuration to network device

1.3.8 Version 3.22

- Remove database url from settings. Configured via env variable DATABASE_URL
- Remote scheduler
- Remove TACACS+ parameters from settings, use env variable instead: TACACS_ADDR, TACACS_PASSWORD
- Make REST API accept Tacacs and LDAP credentials (in the last version, if you were using TACACS+ or LDAP, you could authenticate

in the UI but couldn't make calls to the REST API) - Remove LDAP parameters from settings. The LDAP authentication is in the custom controller, there is a default function that works with a standard LDAP installation, but you can customize however you want. The LDAP server is now configured with the env variable LDAP_SERVER. The settings contain a new section "database" to enable ldap, database or tacacs authentication. - Add replier option in send mail mechanism - Rename "app_log" option to "changelog" in log function for services - Add new entry in workflow RC menu "Workflow Results Table": contains all results for a given runtime, allowing for comparison of results same device / different service, same service / different device, etc. - Refactor logging mechanism. In settings.json, add new logging sections to configure whether the log for a given logger should also be logged as changelog or service log by default. - RBAC - Fix authentication bug flask_login and add session timeout mechanism - Make plugins separate from eNMS in their own folder, add bash script to install/update/uninstall them - Make the CLI interface a plugins - Remove summary from service state to improve workflow refresh performances - Add Dark mode and theme mechanism - Make search endpoint work with result to retrieve device results - Allow dictionary and json as custom properties. For json properties, use jsoneditor to let the user edit them. - Add placeholder as a global variable in a workflow (e.g to be used in the superworkflow) - Add mechanism for creating custom configuration property - Refactor data backup services with custom configuration properties. Implement "Operational Data" as an example custom property. - Add new Git service. Replace "git_push_configurations" swiss army knife service with instance of git service. - Add database fetch/commit retry mechanism to handle deadlocks & other SQL operational errors - Add validation condition for validation section.

MIGRATION: - Remove RBAC in rbac.json - Update migration files (user.yaml): group: Admin -> groups: [Admin Users] - app_log -> changelog in the service migration files (python snippet services) - set_var: add export keyword set to True in service.yaml for backward compatibility - rename DataBackupService / NetmikoBackupService, data_backup_service -> netmiko_backup_service

1.3.9 Version 3.21.3

- Add new plugins mechanism
- Fix bug help panel open when clicking a field or label
- Add error message in the logs when a service is run in per device mode but no devices have been selected.
- Add default port of 22 for TCP ping in ping service
- Disable edit panel on double-click for start/end services of a workflow
- Fix invalid request bug when pressing enter after searching the “add services to workflow” panel
- Forbid “Start”, “End” and “Placeholder” for service names
- Fix Result in mail notification for run once mode
- Make Netmiko prompt command service a substitution string in the UI
- Fix wrong jump password when using a Vault
- Fix workflow results recursive display no path in results bug
- Improve “Get Result” REST endpoint: returns 404 error if no run found, run status if a run is found but there are

no results (e.g job still running), and the results if the job is done. - Remove wtforms email validator in example service following wtforms 2.3 release

1.3.10 Version 3.21.2

- Fix rest api update endpoint bug
- Add device results to rest api get_result endpoint
- Rename subservice -> placeholder
- Fix rendering of custom boolean properties
- Fix custom properties accordion in service panel
- Fix service cascade deletion bug with service logs and placeholder
- Fix front-end alert deleting services and make it a success alert
- Fix historical config / oper data comparison mechanism
- Fix bug where superworkflow cannot be cleared from list after selection
- Fix bug placeholder service deletion from workflow
- Make superworkflow a workflow property only. Remove superworkflow targets option
- Display only workflows in the superworkflow drop-down list
- Save alert when displaying python error as an alert
- When using a custom logger, only the actual user content is logged

- Update docs rest API
- Improve log function (custom logger behavior / creator)
- Fix superworkflow bug for standalone services
- Dont display private properties in parameterized run results
- Add Ansible playbook service log to security logger
- Update superworkflow initial payload with placeholder service initial payload
- Dont update netmiko and napalm configuration / oper data backup if empty result / no commands

1.3.11 Version 3.21.1

- Upgrade JS Panel to v4.10
- Fix jspanel position on long pages with a scrollbar
- Fix placeholder double-click bug
- Fix table display bug
- Fix operational data display bug

1.3.12 Version 3.21

- When entering a subworkflow, the selected runtime is now preserved.
- When running a workflow, the runtime is added to the runtime list in workflow builder and selected.
- Workflow Refresh button now updates the list of runtimes in the workflow builder dropdown of runtimes.
- Duplicating a shared service from the workflow builder now creates a NON SHARED deep copy in the current workflow only.
- Created dedicated category for shared services in “Add services to workflow” tree.
- Implemented “Clear all filters” mechanism for all tables
- When displaying workflow services in service table, all search input resetted (otherwise nothing was displayed)
- Add download buttons for configuration and operational data
- Add button in tables to export search result as CSV file.
- When duplicating top-level workflow, display edit panel
- Fix progress display for service in run once mode in workflow builder
- Multiline field for skip / device query
- Add “Maximum number of retries” property to prevent infinite loop (hardcoded before)
- Add “All” option in relationship filtering (filter object with relation to All)
- Rename “never_update” with “manually_defined”
- Set focus on name field when creating a new instance
- New property in service panel (targets section): Update pools before running.
- Extend the custom properties to all classes including services (displayed in an accordion in first tab).
- Add new search mechanism in the “Add services to workflow” panel

- Add new “Trigger” property for runs to know if they were started from the UI or REST API
- Add time-stamp of when the configuration / oper data displayed was collected
- Ability to display config older config from GIT
- Ability to compare currently displayed config/data to any point in time in the past.
- Syntax highlight option: ability to highlight certain keywords based on regular expression match, defined in eNMS/static/lib/codemirror/logsMode. Can be customized.
- New logging property to configure log level for a service or disable logging.
- Fix bug when typing invalid regex for table search (eg “(”)
- Dont display Start / End services in service table
- Make configuration search case-insensitive for inclusion (“Search” REST endpoint + UI)
- Use log level of top-level workflow for all services.
- Add context sensitive help mechanism
- Add keyword so that the “log” function in a service can log to the application log (+ create log object)
- Add timestamp for session logs
- Add device result counter in result tree window
- Move to optional_requirements file and catch import error of all optional libraries: ansible, hvac, ldap3, pyats, pynetbox, slackclient>=1.3,<2, tacacs_plus
- Fix Napalm BGP example service
- Fix 404 custom passwords logs from Vault
- Encrypt and decrypt all data going in and out of the vault (b64 / Fernet)
- No longer store user password when external authentication is used (LDAP/TACACS+)
- No longer create / import duplicated edges of the same subtype.
- Add preprocessing code area for all services
- all post processing mode: “run on success” / “run on failure” / “run all the time” selector
- Support functions and classes with set_var / get_var
- Fix front end bug when displaying the results if they contain a python SET (invalid JSON): all non-JSON compliant types are now automatically converted to a string when saving the results in the database, and a warning is issue in the service logs.
- Add superworkflow mechanism
- Add jump on connect support
- Add log deletion support from CLI interface
- Forbid import of “os”, “subprocess” and “sys” in a python code area in service panel (snippet, pre/postprocessing, etc)
- Refactor logging configuration: all the logging are now configured from a file in setup: logging.json Besides, the log function in a workflow takes a new parameter “logger” where you can specify a logger name. This means you can first add your own loggers in logging.json, then log to them from a workflow.
- Remove CLI fetch, update and delete endpoint (curl to be used instead if you need it from the VM)

- Improve workflow stop mechanism: now hitting stop will try to stop ASAP, not just after the on-going service but also after the on-going device, or after the on-going retry (e.g many retries…). Besides stop should now work from subworkflow too.

MIGRATION: In services, "result_postprocessing" -> "postprocessing" In pools, "never_update" -> "manually_defined" use_jumpserver -> jump_on_connect In settings.json, the log level is no longer in the "section" but in a dedicated "logging" section. In settings.json, configure Syslog Handler (Security logs).

CUSTOM SERVICES FILE MIGRATION: Fields are no longer imported from wtforms. All of them are now imported from eNMS.forms.fields Some of them have been removed: - substitution and python query are now a keyword - no validation is a keyword too

Imported via db: MutableList -> db.List MutableDict -> db.Dict Column -> db.Column SmallString -> db.SmallString LargeString -> db.LargeString

1.3.13 Version 3.20.1

- Update Generic File Transfer Service
- Fix runtime display bug in results window
- Fix file download and parameterized run bugs.
- Refactor LDAP authentication
- LDAP as first option if the LDAP authentication is active in settings
- Fix timing issue in SSH Desktop session mechanism
- Remove unique constraint for link names.
- Hash user passwords with argon2 by default. Add option to not hash user passwords in settings.
- Move linting and requirements in dedicated /build folder.
- Renamed key "pool" with "filtering" in properties.json
- Fix Service table filtering
- Fix object filtering from the network visualization page
- Fix Ansible service safe command bug and add regression test
- Remove column ordering for association proxy and all columns where ordering isn't useful
- Fixed workflow builder display when the path stored in local storage no longer exists
- Add service column in device results table
- Add result log deletion endpoint in RBAC
- Fix bug dictionary displayed in the UI in the results
- Add all service reference in submenu in workflow builder
- Add entry to copy service name as reference.
- Add new feature to accept a dictionary in iteration values. When a dictionary is used, the keys are used as the name of the iteration step in the results.
- Iteration variable are now referred to as global variable,
- Catch all exceptions in rest api to return proper error 500 (device not found for get configuration, etc)
- Fix bug position of shared services resetted after renaming workflow
- Fix refresh issue in configuration / operational data panel

- Fix upload of files from file management panel
- Forbid sets in the initial payload
- Fix user authentication when running a service
- Fix filtering tooltip in result table (no target found)
- Fix filtering per result type (success / failure) in result table
- Fix retry numbering
- Add Search REST endpoint

MIGRATION: All iteration variable became GLOBAL VARIABLE, which means that you need to use {{variable}} instead of {{get_var("variable")}} previously All services that use iteration variables must be updated in the migration files.

1.3.14 Version 3.20

- Add configuration management mechanism
- New Table properties mechanism: all table properties are displayed in a JSON file: you can configure which ones appear in each table by default, whether they are searchable or not, etc, their label in the UI, etc. You will need to add your CUSTOM properties to that file if you want them to appear in the table.
- Same with dashboard properties and pool properties
- New Column visibility feature
- New Configuration Management Mechanism
- RBAC
- Refactoring of the search system: next to the input, old "Advanced Search" button now dedicated to relationship. Everything is now persisted in the DOM.

MIGRATION: - In netmiko configuration backup service, rename:

- "configuration" -> "configuration_command"
- "operational_data" -> "operational_data_command"
- Moved ansible, pyats to a dedicated file called "requirements_optional.txt":

1.3.15 Version 3.19

- Add new File Management mechanism: browse, download, upload, delete and rename local files. Mechanism to use local files as part of the automation services.
- Add new color code for the logs window.
- Add New Copy to clipboard mechanism:
  - copy from RC on a service in Workflow builder
  - copy from icon in result tables
  - copy dict path to result in the json window.
- Full screen workflow builder
- Remember menu size PER USER
- Refactoring of all the tables

- Refactoring of the top-level menu
- Alerts are saved and displayed in the UI, top menubar.
- Remove recipients from settings.json. Recipients is now a mandatory field if mail notification is ticked.
- Add support for netmiko genie / pyATS (*use_genie*) option.
- New "Desktop session" mechanism to SSH to a device using teraterm / putty / etc.

MIGRATION: - Renaming "config" -> "settings". All services that use the "config" global variable must change it to "settings". - Session change log: some traceback previously returned as "result" key of service "results" now returned as "error": can create backward-compatibility issue when a workflow relies on the content of the traceback.

1.3.16 Version 3.18.2

- Fix subworkflow iteration bug
- Fix workflow display with same shared services in multiple subworkflows
- Fix task / run cascade deletion bug on MySQL
- Add "devices" keyword for result postprocessing
- Allow restart from top-level workflow when restarting from a subworkflow service
- New "Skip value" property to decide whether skip means success or failure
- Fix the workflow builder progress display when devices are skipped. Now eNMS shows how many devices are skipped, and it no longer shows anything when it's 0 ("0 failed", "0 passed" etc are no longer displayed)
- Netmiko session log code improvement for netmiko validation / prompt service

1.3.17 Version 3.18.1

- Display scoped name in hierarchial display mode
- Fix bug "Invalid post request" editing edge
- Improve display of filtering forms
- Reduce size of the service and workflow edit panel for low-resolution screens
- Add "success" key before result postprocessing
- Remove "Enter subworfklow" button in toolbar and add the same button in right-click menu
- Add button to switch to parent workflow

1.3.18 Version 3.18

- Add Operational Data mechanism
- Removed Clusterized and 3D View
- Changed configuration to be a .json file instead of env variables
- Removed Custom config and PATH_CUSTOM_CONFIG
- Remove Configuration comparison mechanism
- Display the results of a workflow as a tree

- Change the mechanism to add a service to a workflow to be a tree
- Add the forward and backward control to the service managemet table.
- Duplicate button at workflow level to duplicate any workflow as top-level workflow
- Update to the operational data backup service to include rancid-like prefixes
- Add new "run method" property to define how a service is running (once per device, or once for all devices), and the equivalent property for workflow: run device by device, or service by service.
- Replace endtime with "duration" in the results and run table
- Fix bug infinite loop when adding a workflow to itself
- New "run method" option for services: : - once per device - once for all devices
- New "run method" option for workflow - run device by device - service by service with workflow targets - service by service with service targets

1.3.19 Version 3.17.2

- Add Operational Data mechanism
- Removed Clusterized and 3D View
- Changed configuration to be a .json file instead of env variables
- Removed Custom config and PATH_CUSTOM_CONFIG
- Remove Configuration comparison mechanism

1.3.20 Version 3.17.1

- Performance optimization

1.3.21 Version 3.17

- Performance improvements
- Refactoring of the result window
- Refactoring of the search system
- Forbid single and double-quotes in names.
- Moved the validation mechanism to the base "Service" class. Validation is now available for all services.
- New "Close connection" option for a service. Closes cached connection.
- In the "Advanced search", new "None" entry for filtering relationship.
- Removed mypy from both the codebase and CI/CD test (travis).
- Refactoring of the configuration management system.
- Refactoring of the workflow system
- Ability to specify the alignment for workflow labels
- Upon creating the admin user, check if there is a password in the Vault. If there isn't, create it ("admin").
- Remove beginning and trailing white space Names (service name ends with space breaks get_results)

- Add config mode and honor it when retrieving a cached connection.
- Netmiko Validation Service: allow several commands

1.3.22 Version 3.16.3

- If the admin password is not set (db or Vault) when creating the admin user, set it regardless of the config mode.
- Move skip / unskip button to right-click menu.

1.3.23 Version 3.16.2

- Always delete a workflow when it is imported via import job
- New "Maximum number of runs" property for a job in a workflow: defines how many times the same job is allowed to run in the workflow.
- New "Result postprocessing" feature: allows for postprocessing the results of a service (per device if there are devices), including changing the success value.
- Add new version of Unix Shell Script service
- Enable multiple selection in the workflow builder + mass skip / unskip buttons

1.3.24 Version 3.16.1

- New feature to stop a workflow while it's running

1.3.25 Version 3.16

- Add "Workflow Restartability" window when clicking on a job.
- Cascade deletion of runs and results when jobs / devices are deleted.
- Forbid empty names and names with slash front-end
- Fix event issue after adding jobs to the workflow builder.
- Create and delete iteration loopback edge upon editing the service.
- Fix change of name in workflow builder upon editing the service.
- Make iteration variable name configurable
- Ansible add exit status:
- Workflow notes Desc: Support textboxes added to a workflow that are displayed in the workflow builder.
- New mechanism: success as a python query kind of thingAdd success query mechanism
- New Mechanism to switch back and forth in the workflow builder.
- New "Latest runtime" option in workflow builder.
- When displaying a workflow, automatically jump to the latest runtime.
- In Workflow builder, add the name of the user who ran the runtime in the runtime list.
- Display number of runs in parallel in the Service Management / Workflow Management page, next to the Status (Running / Idle)

- Job now displayed in grey if skip job is activated.
- Edge labels are now editable
- Results display: in text mode, multiline strings are now displayed without any transformation.
- User inactivity monitoring

1.3.26 Version 3.15.3

- "Use Workflow Targets" is now "Device Targets Run Mode"
- Service mode: run a workflow service by service, using the workflow targets Device mode: run a workflow device by device, using the workflow targets Use Service targets: ignore workflow targets and use service targets instead

1.3.27 Version 3.15.2

- New "Iteration Targets" feature to replace the iteration service
- Front-end validation of all fields accepting a python query
- check for substitution brackets ({{ }}) that the expression is valid with ast.parse
- Add new regression test for the payload extraction and validation services
- Payload extration refactoring
  - Store variables in the payload global variable namespace
  - Add optional operation parameter for each variable: set / append / extend / update
- New conversion option: "none" in case no conversion is necessary
- No longer retrieve device configuration when querying REST API.
- Remove web assets
- Refactor SQL Alchemy column declaration for MySQL compatibility
- Hide password in Ansible service results.
- Private properties are no longer considered for pools.

1.3.28 Version 3.15.1

- Waiting time is now skipped when the job is skipped.
- Change result to mediumblob pickletype
- remove Configurations from ansible command
- remove table filtering N/A
- Add more regression tests (including skip job feature)

1.3.29 Version 3.15

- New env variable: CUSTOM_CODE_PATH to define a path to a folder that contains custom code that you can use in your custom services.
- Advanced search: per relationship system
- eNMS version now displayed in the UI. The version number is read from the package.json file.
- Real-time log mechanism with multiprocessing enabled.
- Workflow restartability improvement:
- Fixed bug in tables: jump to bottom after page 1 when table is refreshed.
- Fixed panel repaint bug when pulling it down.
- Relationship are now displayed in the edit window: you can edit which service/workflow a device/task is a target of, etc…
- Spinning GIF when AJAX requests
- Add new services in a workflow: services are spread in a stairsteps in the workflow builder.
- Workflow Builder: edit the service when it's double clicked
- Copy to clipboard for device configuration
- Fix bug subworkflow edit panel
- Export Jobs needs to automatically delete devices and pools
- Service should fail if a python query produces a device target that does not match inventory/database
- timeout and other parameters getting updated for all services using cached Netmiko connections.
- Ability to close a cached connection and re-originate the connection in a service.
- Start time of each Service within a Workflow displayed,
- User can now track the progress of a workflow even if the workflow was started with a REST call
- New GET Result Endpoint for the REST API to get the result of a job run asynchronously: if async run_job was invoked, you can use the runtime returned in the REST response to collect the results after completion via a GET request to /result/name/runtime
- New Run Management window:
- Slashes are now forbidden from services and worklfow names (conflict with Unix path)
- The command sent to a device is now displayed in the results
- Credentials are now hidden when using gotty.
- Job Parametrization.
- Service type now displayed in the workflow builder.
- New service parameter: Skip (boolean)
- New parameter: Skip query (string) Same as skip, except that it takes a python query.
- Added number of successful / failed devices on workflow edges.
- Run status automatically switched from "Running" to "Aborted" upon reloading the app.
- napalm getter service: default dict match mode becomes inclusion.
- Replaced pyyaml with ruamel

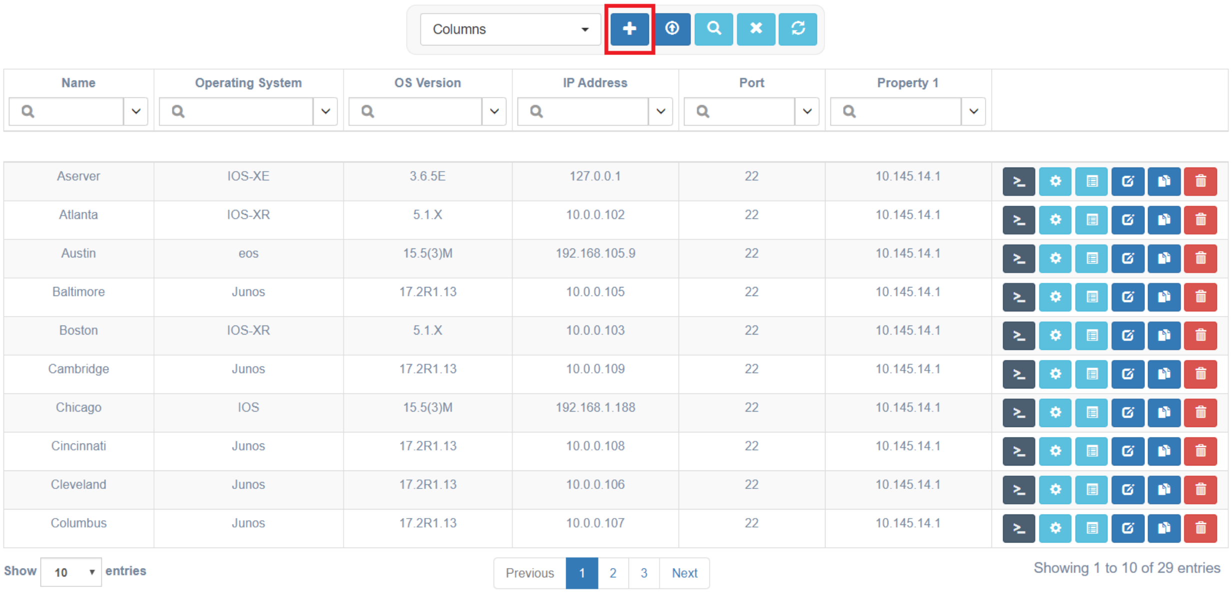
- Both true and True are now accepted when saving a dictionary field.
- Set stdout_callback = json in ansible config to get a json output by default.
- Change in the LDAP authentication: LDAP users that are not admin should now longer be allowed to log in (403 error).
- The "dictionary match" mechanism now supports lists.
- New "Logs" window to see the different logs of a service/workflow for each runtime.
- Show the user that initiated the job, along with the runtime when selecting a run

1.4 Network Creation

The network topology can be created in two different ways:

1.4.1 From the UI

By filling a form in *Inventory / Devices* and *Inventory / Links* ("+" button)



Name	Operating System	OS Version	IP Address	Port	Property 1
Aserver	IOS-XE	3.6.5E	127.0.0.1	22	10.145.14.1
Atlanta	IOS-XR	5.1.X	10.0.0.102	22	10.145.14.1
Austin	eos	15.5(3)M	192.168.105.9	22	10.145.14.1
Baltimore	Junos	17.2R1.13	10.0.0.105	22	10.145.14.1
Boston	IOS-XR	5.1.X	10.0.0.103	22	10.145.14.1
Cambridge	Junos	17.2R1.13	10.0.0.109	22	10.145.14.1
Chicago	IOS	15.5(3)M	192.168.1.188	22	10.145.14.1
Cincinnati	Junos	17.2R1.13	10.0.0.108	22	10.145.14.1
Cleveland	Junos	17.2R1.13	10.0.0.106	22	10.145.14.1
Columbus	Junos	17.2R1.13	10.0.0.107	22	10.145.14.1

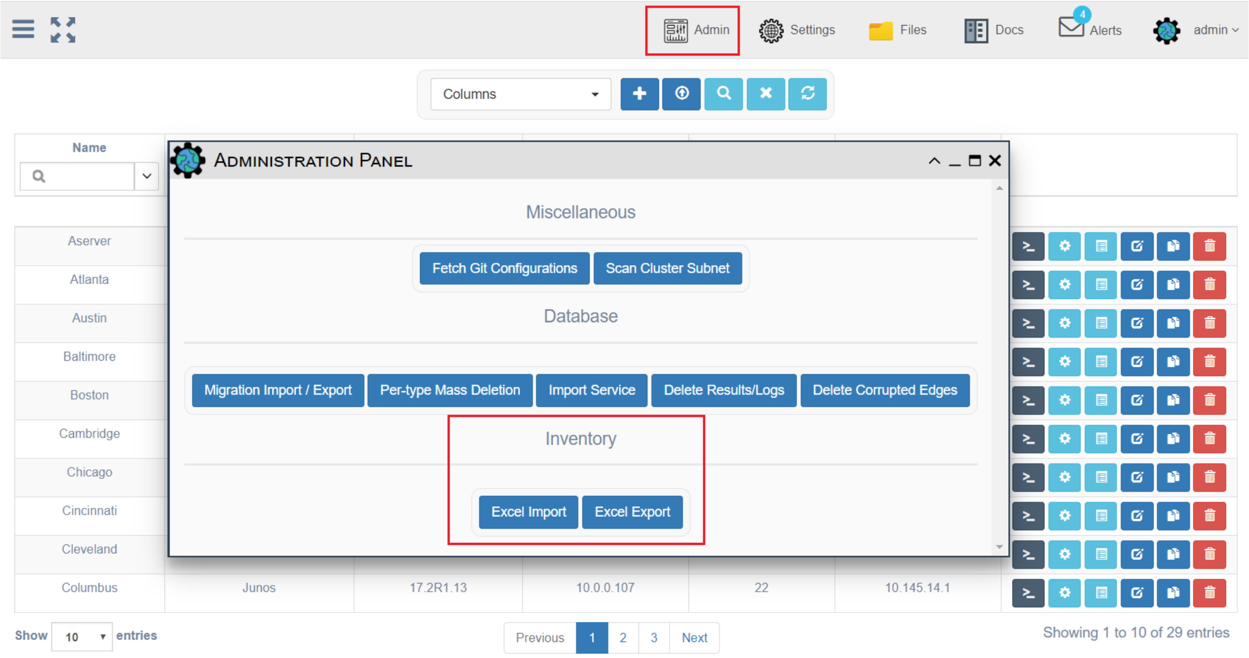
Note: Some properties are mandatory:

- Name: objects are uniquely defined by their name.
- Source and destination: a link needs a source and a destination to be created.

Note: In order to visualize the network topology on a map, devices must have geographical coordinates (longitude and latitude).

1.4.2 From an Excel spreadsheet

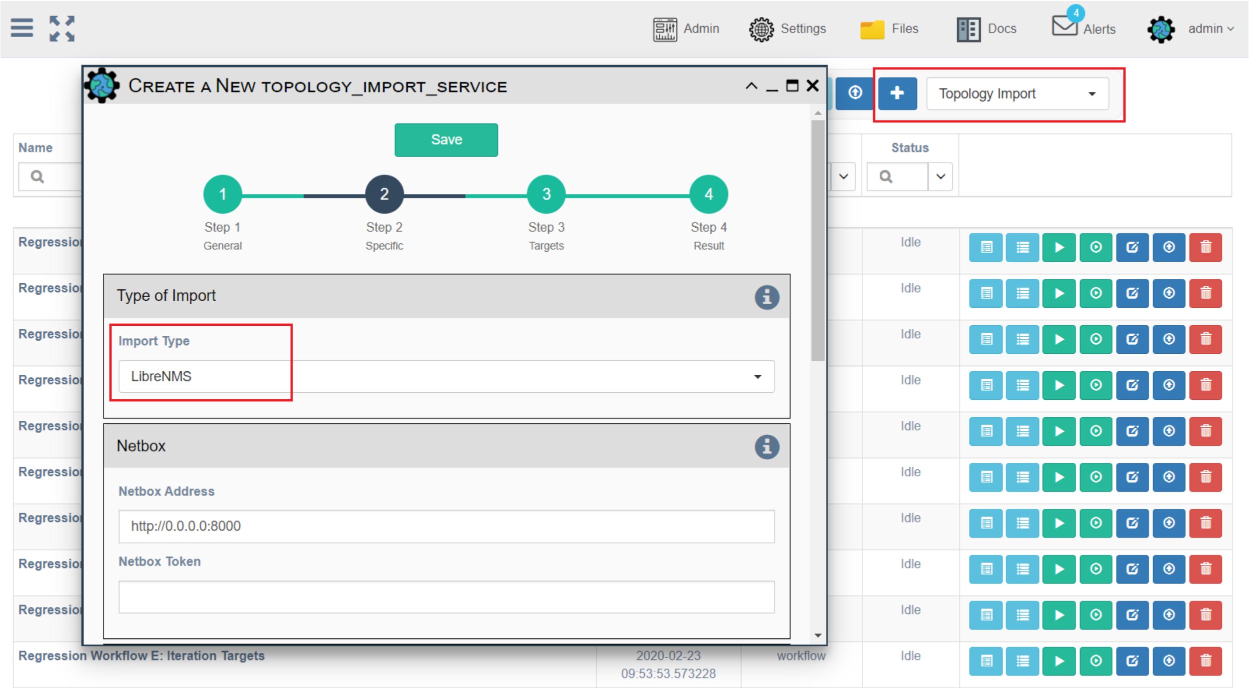
The inventory can be imported from / exported to an Excel spreadsheet in the admin panel (see screenshot below), section `Inventory`. You can find examples of such spreadsheets in `files` / `spreadsheets`.



Note: If you import an object that has already been created, its properties will be updated.

1.4.3 Querying an external API

Another way to create your network is to query an external API: OpenNMS, Netbox, or LibreNMS. You can do that by creating a "Topology Import" service from the `Services` page.



Import Type: LibreNMS

Netbox Address: http://0.0.0.0:8000

Netbox Token:

You can select an "Import Type" and fill the corresponding section of the form.

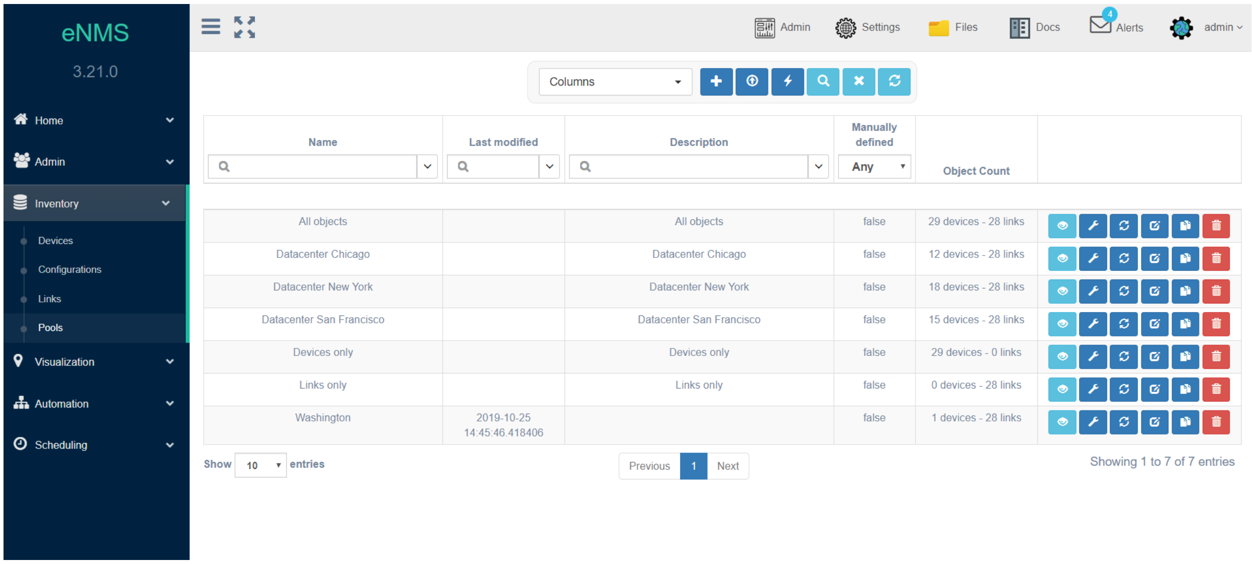
1.5 Pools

The Pools feature allow the user to create groups of Device or Link objects. They can be used to filter the Pool Inventory table and in the *Visualization -> Network* view screen. Pools are also used to target an automation task to a specific, defined, group ("pool") of devices. A pool is defined as a combination of properties values of device or link object. When defining a Pool, for each property value, decide, whether the match is based on inclusion, equality, or regular expression.

If the properties of a device or link object matches the pool properties, that object will be automatically added to the pool.

1.5.1 Pool Management

Pools can be created, updated, duplicated and deleted from *Inventory -> Pools*. They can be edited to manually select devices and links instead of using criteria based on properties.



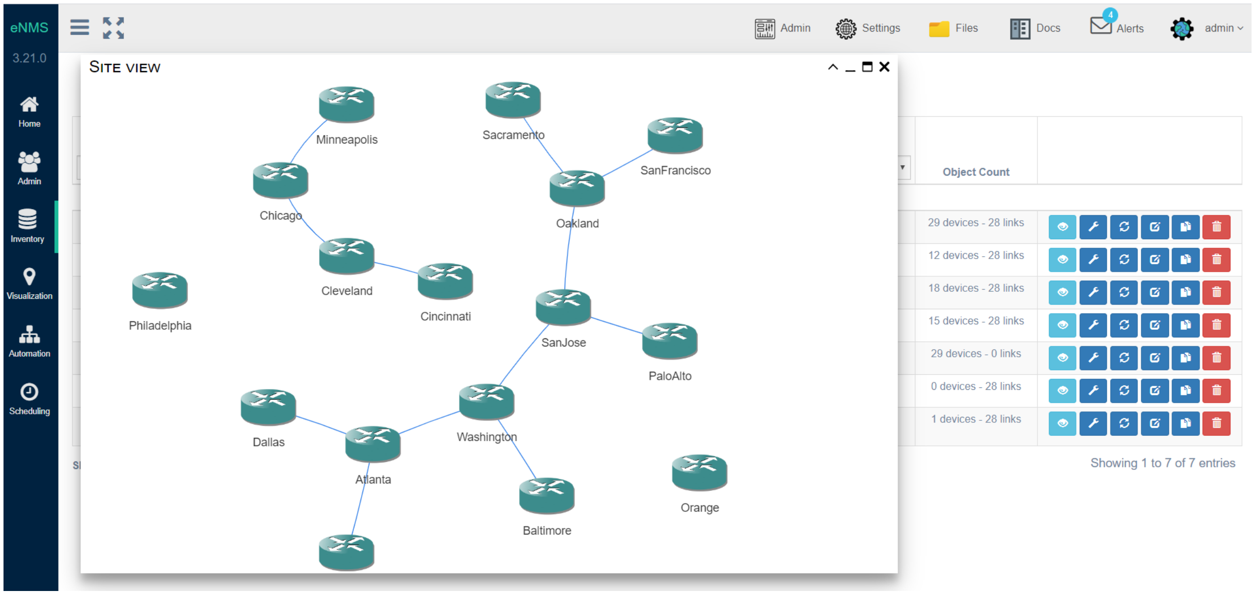
Name	Last modified	Description	Manually defined	Object Count
			Any	
All objects		All objects	false	29 devices - 28 links
Datacenter Chicago		Datacenter Chicago	false	12 devices - 28 links
Datacenter New York		Datacenter New York	false	18 devices - 28 links
Datacenter San Francisco		Datacenter San Francisco	false	15 devices - 28 links
Devices only		Devices only	false	29 devices - 0 links
Links only		Links only	false	0 devices - 28 links
Washington	2019-10-25 14:45:46.418406		false	1 devices - 28 links

Show 10 entries

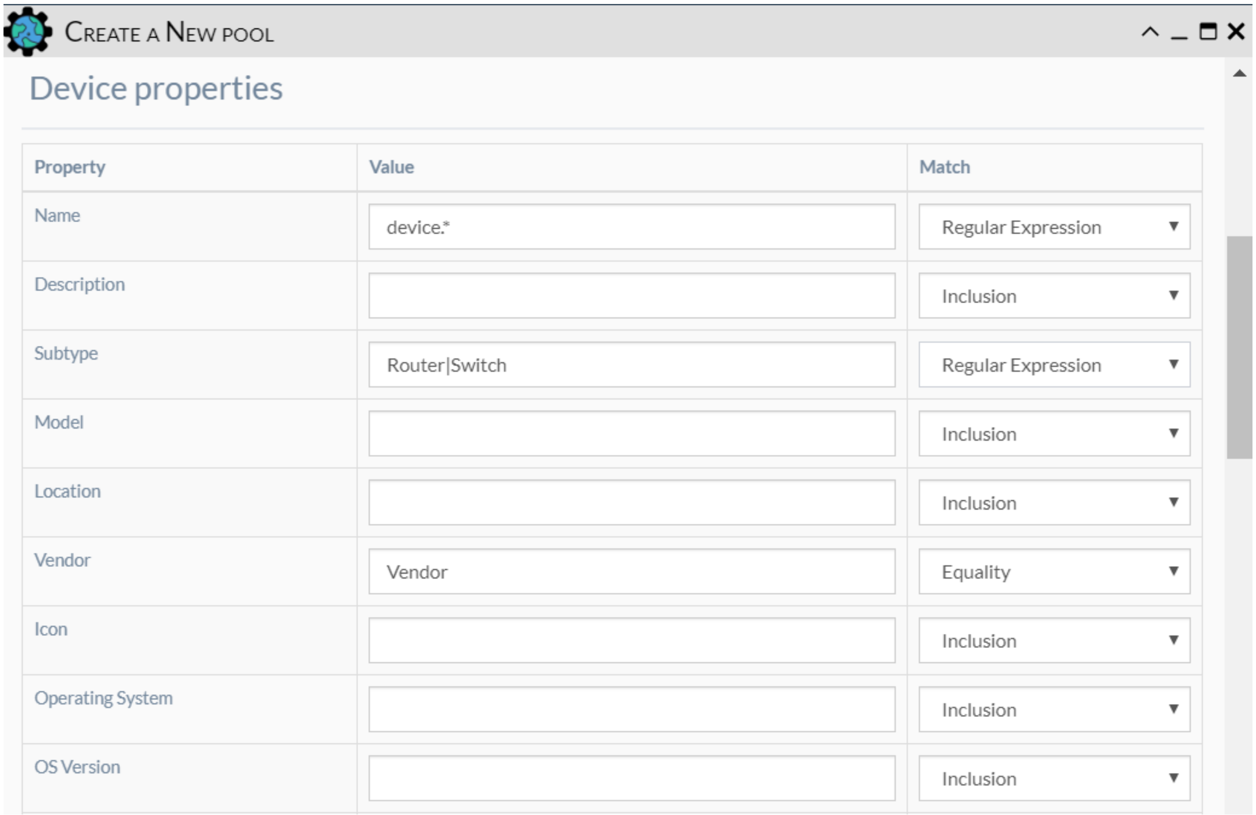
Previous 1 Next

Showing 1 to 7 of 7 entries

A Logical View of a pool can be displayed using the `Internal View` button.



1.5.2 Device Pool creation example



CREATE A NEW POOL

Device properties

Property	Value	Match
Name	device.*	Regular Expression
Description		Inclusion
Subtype	Router|Switch	Regular Expression
Model		Inclusion
Location		Inclusion
Vendor	Vendor	Equality
Icon		Inclusion
Operating System		Inclusion
OS Version		Inclusion

This pool enforces the Union of the following conditions:

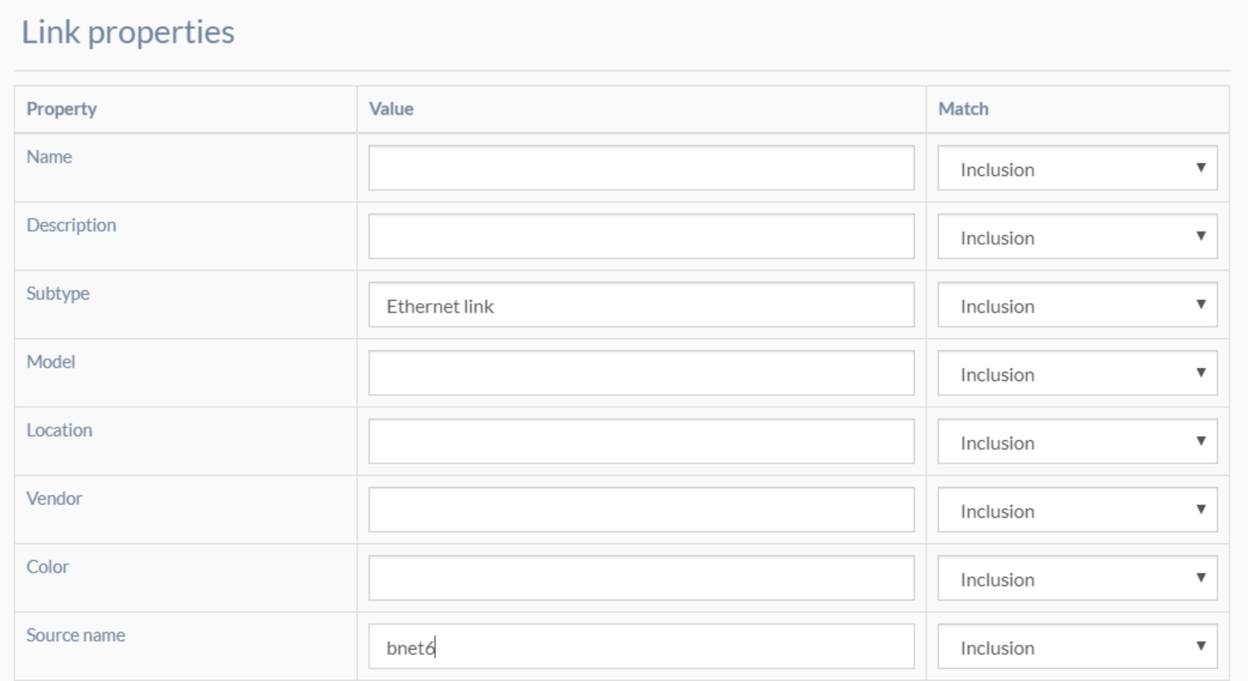
- name: `Washington.*` — Match is Inclusion; all devices whose name include `Washington` will be selected
- subtype: `GATEWAY|PEER` — Match is Regular Expression; all devices having subtype `GATEWAY` and `PEER` will be selected
- vendor: `Cisco` — Match is Equality; all devices whose vendor is `Cisco` will be selected

In summary, all `Cisco` `GATEWAY and PEER` whose name begins with `Washington` will match these conditions, and they will be members of the pool.

Note: All properties left with empty fields are simply ignored.

Note: Along with all properties of a device, you can use the device **Configuration** and **Operational Data** as a constraint for the pool. Refer to the Configuration Management page for more information.

1.5.3 Links Pool creation example



Link properties

Property	Value	Match
Name		Inclusion
Description		Inclusion
Subtype	Ethernet link	Inclusion
Model		Inclusion
Location		Inclusion
Vendor		Inclusion
Color		Inclusion
Source name	bnet6	Inclusion

This pool enforces the union of the following conditions:

- subtype: `Ethernet link` — Match is Equality; all Ethernet links will be selected
- source name: `Washington.*` — Match is Inclusion; all links whose source is the device name include `Washington` will be selected

In summary, all `Ethernet` links starting from devices with name that include `Washington` will be members of the pool.

1.5.4 Default Pools

Three pools are created by default in eNMS:

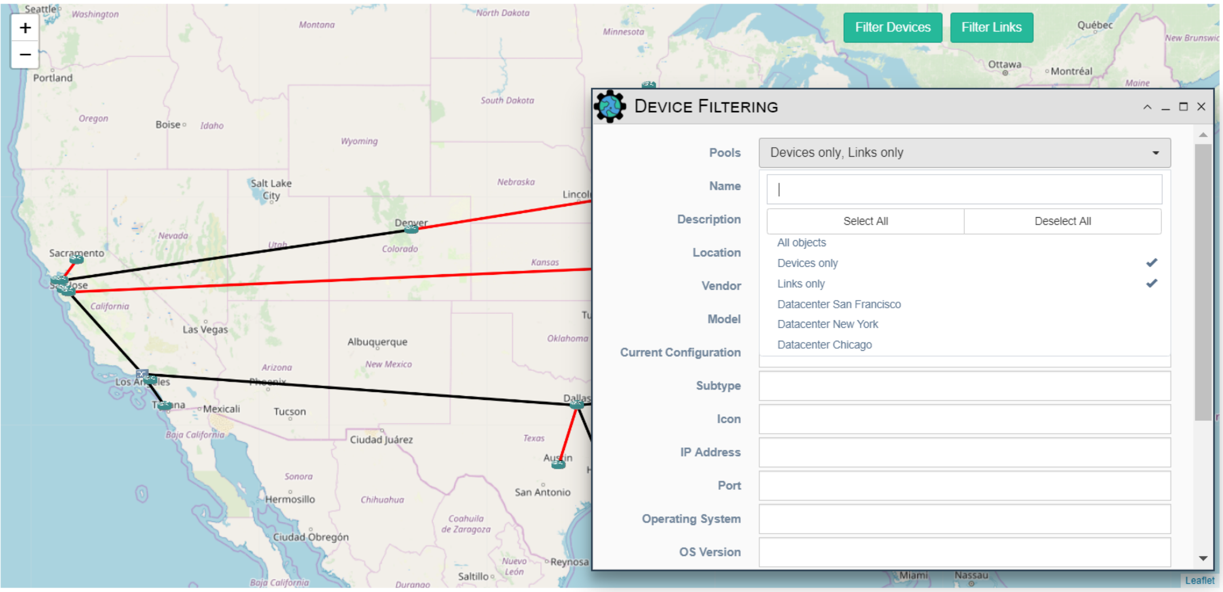
- "All objects": A pool that matches all Devices and Links.
- "Devices only": A pool that matches all Devices, no Links.
- "Links only": A pool that matches all Links, no Device.

1.5.5 Pools based on Configuration

Pools can be created by searching the configurations data collected from all of the devices, rather than just the Inventory parameters for each device. Configuration collection must be, first, configured and then allowed to run at least once before the configurations can be searched upon for the Pool.

1.5.6 Filter the view with a Pool

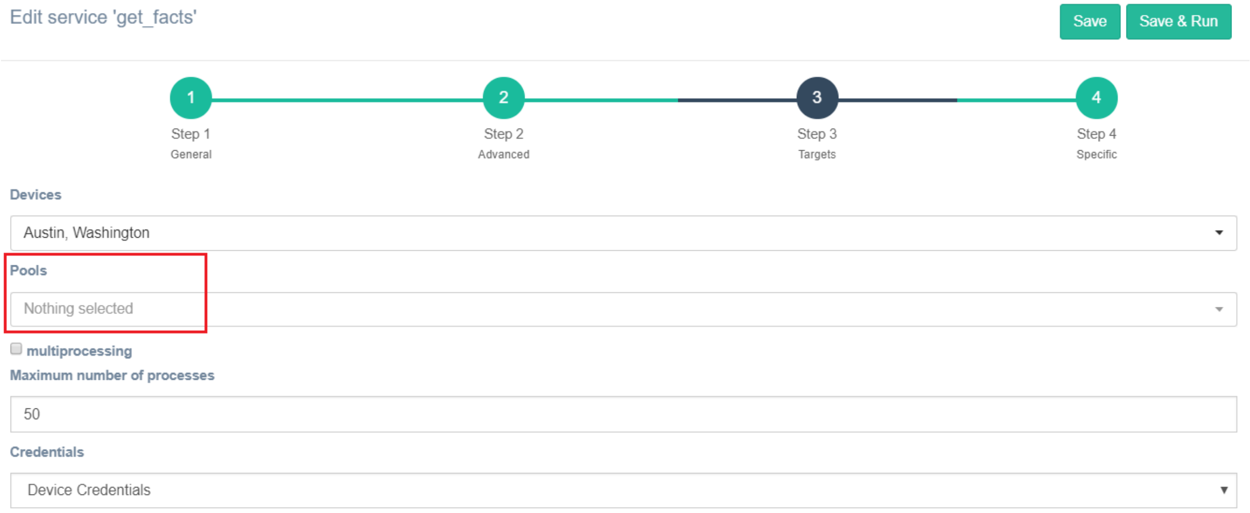
Pools can be used as filters for Devices and Links tables on the geographical views *Visualization* -> *Network*. At the top of the screen, click on the filter button `Devices` or `Link` to open the "Filtering" panel. Both of these panels contain a `Pools` drop-down list (multiple selection) to filter objects in the view. Click the refresh button after selecting filter criteria.



Pools: Devices only, Links only

1.5.7 Use a Pool as target of a Service or a Workflow

In “Step 3”, select Device(s) and/or Pool(s) as target(s).



Devices: Austin, Washington

Pools: Nothing selected

☐ multiprocessing

Maximum number of processes: 50

Credentials: Device Credentials

1.5.8 Use a Pool to restrict a user to a subset of objects

From the *Admin / User Management* panel, you can select a pool used as a database filtering mechanism for a particular user. All mechanisms and all pages in eNMS will be restricted to the objects of that pool for that particular user. The exception is Service and Workflows that have been already configured to run against a particular set of devices and links. If those devices and links are outside of the pool that the user is restricted to, the user will still be able to see them.

1.5.9 Pool recalculation

All Pools are subject to automatic updates by eNMS (contingent upon the fact that its ‘Manually Defined’ flag is NOT set) after creation:

- When the eNMS starts up or restarts
- When a device is manually added to the inventory
- When a device is modified
- When, after pulling or cloning the content from the git configuration repository
- When a service runs that has *Update pools before running* selected in Step 3 Targets
- When the *poller service* runs (service responsible for fetching all device configurations), ONLY the pools for which the device `Current Configuration` are not empty, are updated.

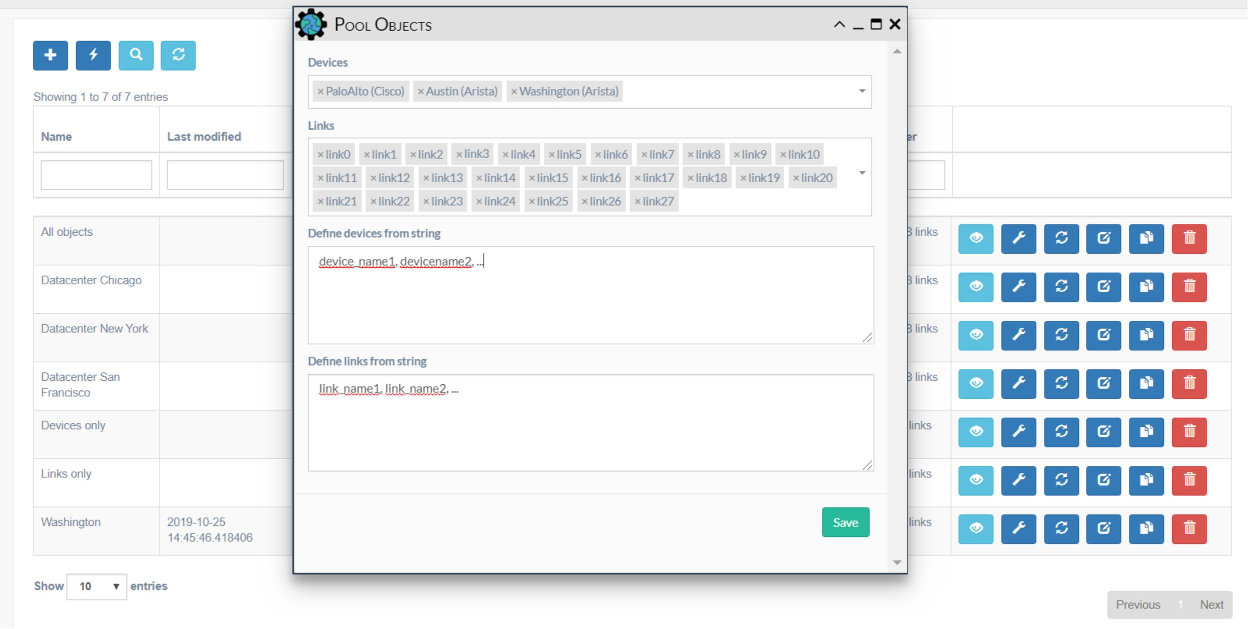
To manually update a Pool:

- Click on the `Update` button of a desired pool in Pool Management table listing
- Click on the `Update all pools` button at the top of Pool Management UI

1.5.10 Manual definition and “Manually Defined” option

Initially, by default, the devices and links within a pool are determined based on the pool properties. The individual pools can be edited by allowing the user to define the devices and links by selecting them directly and there are a couple of ways of doing this:

- Click on `edit` icon: Will allow user to modify the Device Properties and Link Properties.
- Click on `wrench` icon: Will open a “Pool Object” screen to allow a user to copy/pasting a string of comma separated devices and links names as well as selecting devices and links from a drop-down menu field.



Devices: ×PaloAlto (Cisco) ×Austin (Arista) ×Washington (Arista)

Links: ×link0 ×link1 ×link2 ×link3 ×link4 ×link5 ×link6 ×link7 ×link8 ×link9 ×link10 ×link11 ×link12 ×link13 ×link14 ×link15 ×link16 ×link17 ×link18 ×link19 ×link20 ×link21 ×link22 ×link23 ×link24 ×link25 ×link26 ×link27

Define devices from string: device_name1, devicename2, ...

Define links from string: link_name1, link_name2, ...

Note: Pools with manually selected objects need to have the ‘Manually Defined’ checkbox selected. This prevents manually selected pools from being re-calculated based on pool criteria. If the user wants to run against a pool that has some criteria specified as well as some manually specified devices, it is advised to have 2 pools, one with the criteria specified and another with the manually selected devices. When running a service, multiple pools and multiple devices can be specified, and the service will run against all specified objects.

1.6 WebSSH Connection

WebSSH allows the connection to a device via eNMS. Where eNMS handles the establishment of the connection to the device using the information stored in the inventory. eNMS uses GoTTY to automatically start an SSH or Telnet session to a device in the inventory. GoTTY is a terminal sharing web solution that can be found on github: https://github.com/yudai/gotty.

1.6.1 Installation

GoTTY is shipped with eNMS by default in the following directory `/eNMS/files/apps`. You must make sure that the file *gotty* can be executed (`chmod 755 gotty`).

To pass the login passwords to ssh `sshpass` has to be installed on the server where eNMS is running. For the multiplexing feature to work *tmux* has to be installed on the server where eNMS is running.

1.6.2 Port allocation

By default, eNMS will use the range of ports as defined in the *ssh section* in the settings.

eNMS uses a rotation system to allocate these ports sequentially as user requests come in. You can view this range directly from the web UI, via the *Settings* button on the top of the screen, and selecting the `ssh` section.

1.6.3 Custom URL

eNMS automatically redirects you to the address and port GoTTY is listening to, using JavaScript variables `window.location.hostname` and `window.location.protocol`. If these variables do not redirect to the correct URL, you can tell eNMS which protocol and URL to use by changing the `address` variable in the *app section* in the settings (just the URL, and eNMS will add the port GoTTY is listening to).

1.6.4 Port redirection

In a production environment, only one port should be allowed (to be exposed) by the HTTP web server. In that case, the reverse proxy must be configured to redirect the requests sent to `terminal<port_number>` to `localhost:<port_number>`.

With Nginx, this can be accomplished with the following *location* block :

```
location ~ ^/terminal(.*)$ {
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $remote_addr;
  proxy_set_header Host $host;
  rewrite ^/terminal(.*)/?$ / break;
  rewrite ^/terminal(.*)/(.*)$ /$2 break;
  proxy_pass http://127.0.0.1:$1;
  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
```

(continues on next page)

(continued from previous page)

```
  proxy_set_header Connection "upgrade";
}
```

A full example of the nginx configuration can be found in `eNMS/files/nginx`.

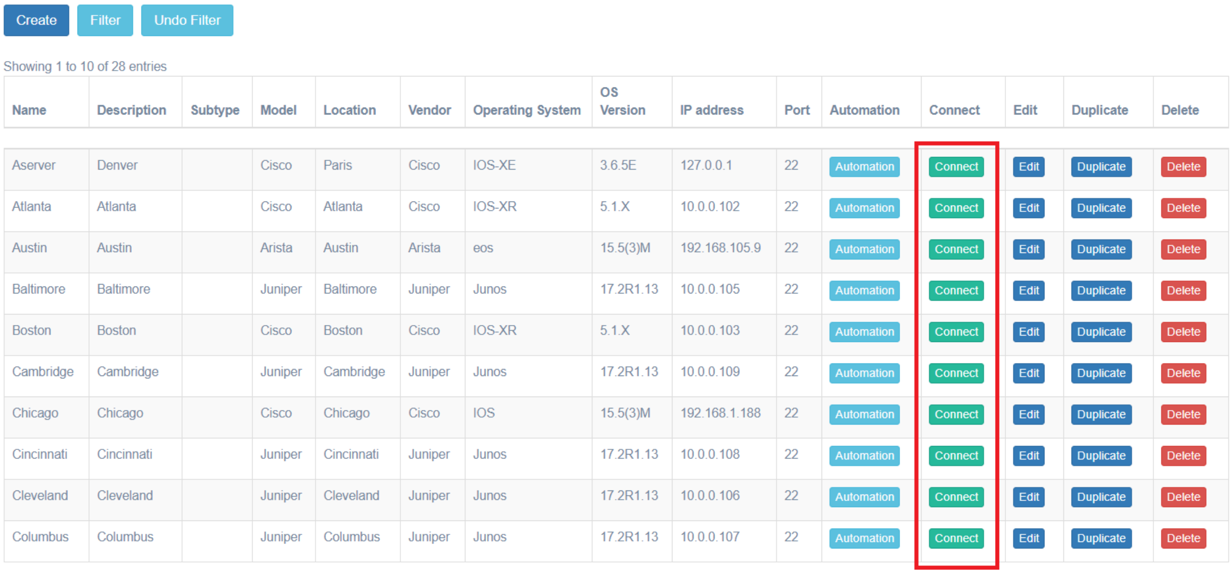
eNMS does not by default perform any port redirection: you must set the `port_redirection` variable to `true` in the *ssh section* in the settings to enable it.

1.6.5 Ignore fingerprint prompt

If the remote device is not in `~/.ssh/known_hosts`, `ssh` prompts the user to add ssh fingerprint to `known_hosts` file, causing GoTTY to fail. To bypass that prompt, you can set the `bypass_key_prompt` to `true` in the *ssh section* in the settings to run the `ssh` command with the options `-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null`.

From the device management table

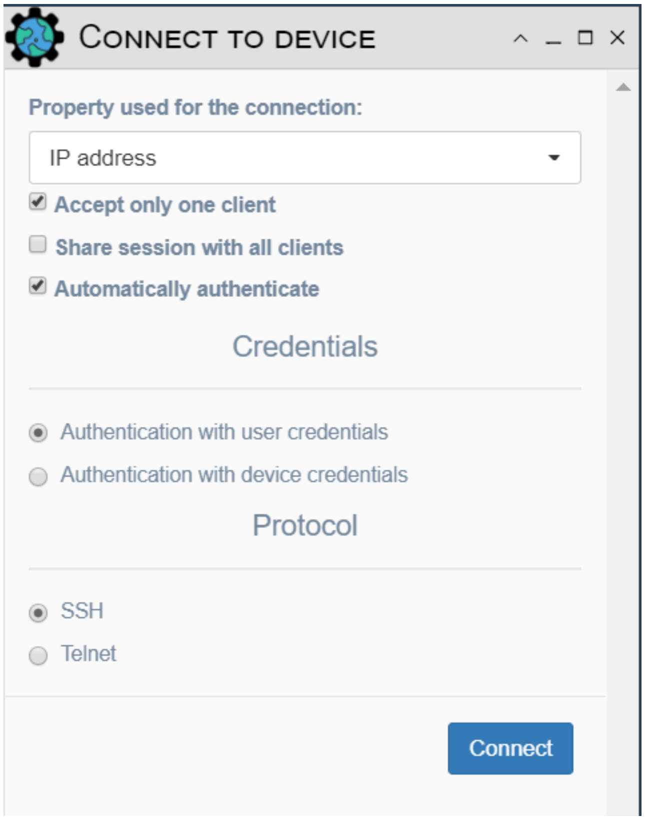
You can connect to a device by clicking on the `Connect` button next to each device entry in the *Inventory / Device* table as shown below.



Showing 1 to 10 of 28 entries

Name	Description	Subtype	Model	Location	Vendor	Operating System	OS Version	IP address	Port	Automation	Connect	Edit	Duplicate	Delete
Aserver	Denver		Cisco	Paris	Cisco	IOS-XE	3.6.5E	127.0.0.1	22	Automation	Connect	Edit	Duplicate	Delete
Atlanta	Atlanta		Cisco	Atlanta	Cisco	IOS-XR	5.1.X	10.0.0.102	22	Automation	Connect	Edit	Duplicate	Delete
Austin	Austin		Arista	Austin	Arista	eos	15.5(3)M	192.168.105.9	22	Automation	Connect	Edit	Duplicate	Delete
Baltimore	Baltimore		Juniper	Baltimore	Juniper	Junos	17.2R1.13	10.0.0.105	22	Automation	Connect	Edit	Duplicate	Delete
Boston	Boston		Cisco	Boston	Cisco	IOS-XR	5.1.X	10.0.0.103	22	Automation	Connect	Edit	Duplicate	Delete
Cambridge	Cambridge		Juniper	Cambridge	Juniper	Junos	17.2R1.13	10.0.0.109	22	Automation	Connect	Edit	Duplicate	Delete
Chicago	Chicago		Cisco	Chicago	Cisco	IOS	15.5(3)M	192.168.1.188	22	Automation	Connect	Edit	Duplicate	Delete
Cincinnati	Cincinnati		Juniper	Cincinnati	Juniper	Junos	17.2R1.13	10.0.0.108	22	Automation	Connect	Edit	Duplicate	Delete
Cleveland	Cleveland		Juniper	Cleveland	Juniper	Junos	17.2R1.13	10.0.0.106	22	Automation	Connect	Edit	Duplicate	Delete
Columbus	Columbus		Juniper	Columbus	Juniper	Junos	17.2R1.13	10.0.0.107	22	Automation	Connect	Edit	Duplicate	Delete

The following window will pop up:



CONNECT TO DEVICE

Property used for the connection:

IP address

☑ Accept only one client

☐ Share session with all clients

☑ Automatically authenticate

Credentials

● Authentication with user credentials

○ Authentication with device credentials

Protocol

● SSH

○ Telnet

You can configure the following parameters :

- Property used for the connection: by default, eNMS uses the IP address but you can also select to use the name, or any custom property.
- Automatically authenticate (SSH only): eNMS will use the credentials stored in the Vault (production mode) or the database (test mode) to automatically authenticate to the network device. eNMS uses `sshpass` for the authentication: it must be installed if you activate the automatic authentication (`sudo apt-get install sshpass`). By default, eNMS uses the user credentials for the authentication (the ones you use to log in to eNMS). However, it can be configured to use the device credentials instead (the credentials that you can specify when creating a new device).
- Protocol: SSH or Telnet.

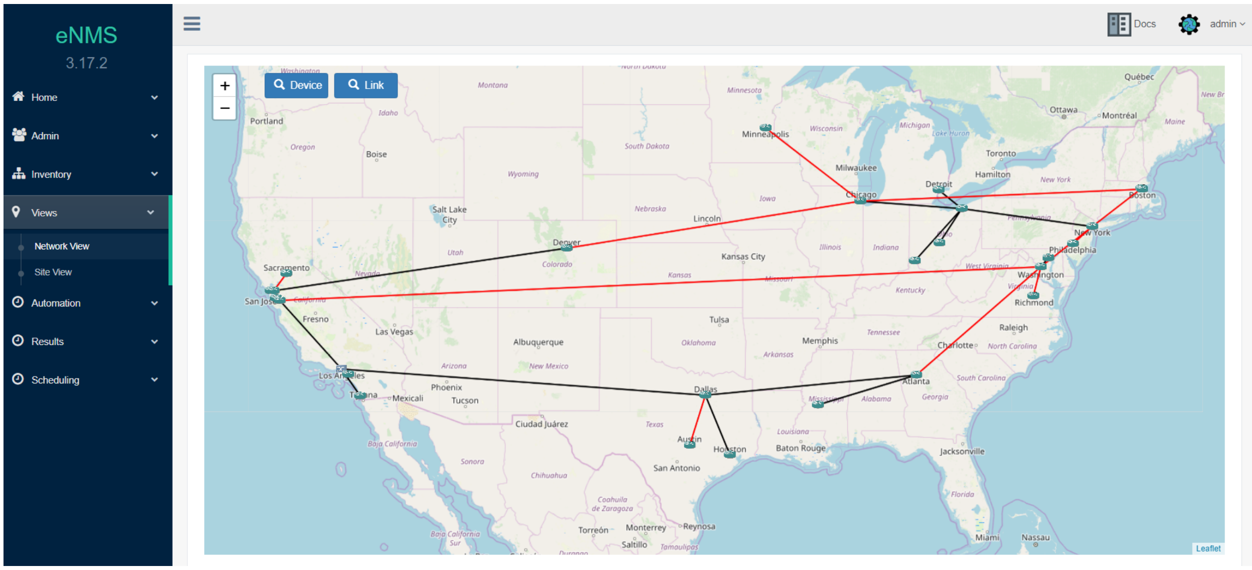
From the Views

You can also connect to a device via the context menu in the geographical view in *Vizualization / Network View* and *Views / Site View*. Hover over a device (the cursor will change to an index finger with a device name pop-up), right click and select `Connect`.

1.7 Network Visualization

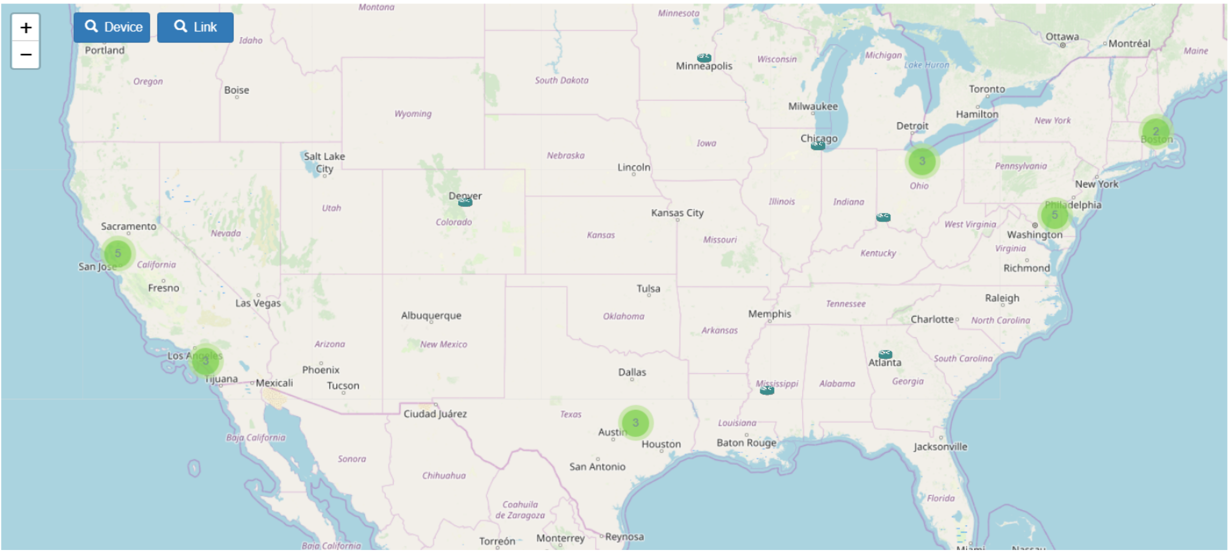
Once the network has been created, it can be displayed on a geographical map.

All devices with a longitude and a latitude are displayed on a map at their exact location. Devices have an `icon` property that defines which image is used, and links have a `color` property.



1.7.1 Clustered view

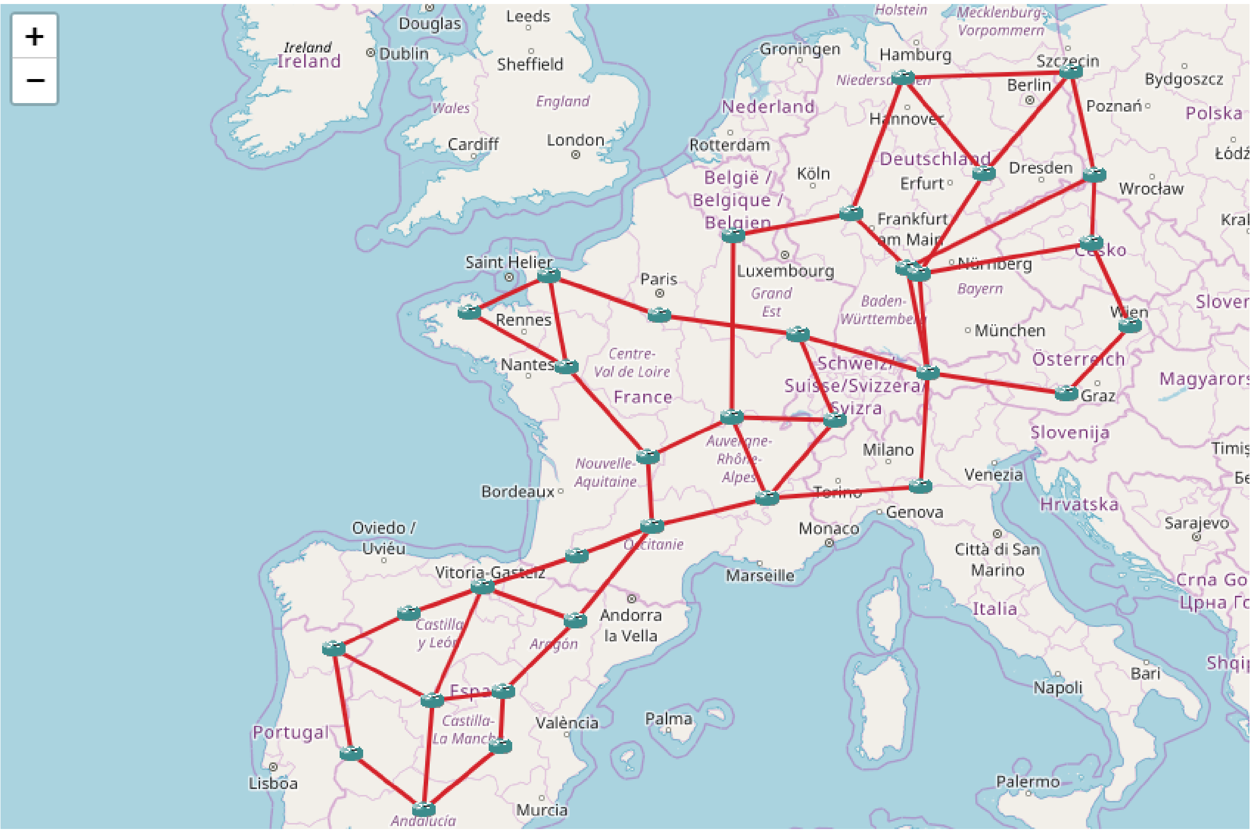
You can right-click on the map and select `Type of View` > `Clustered` to display your network devices as a set of clusters.



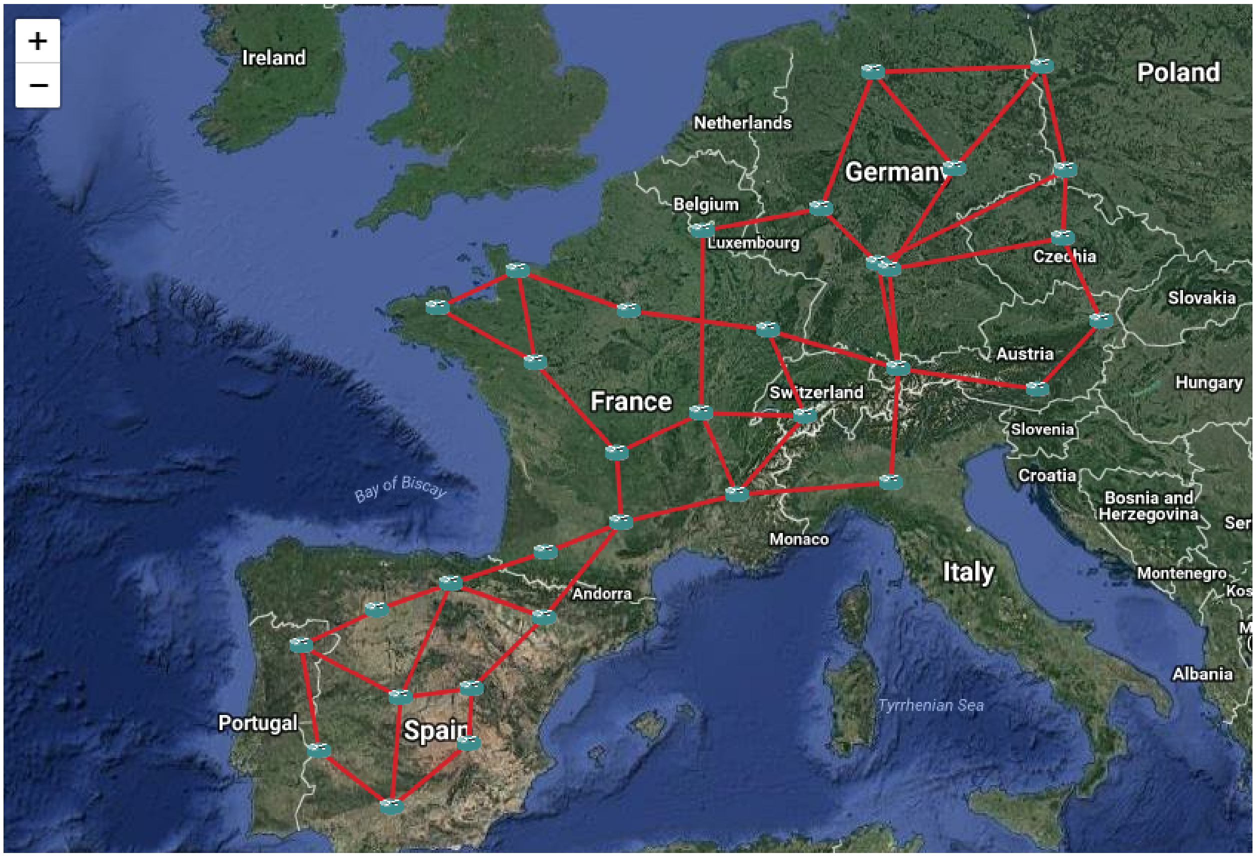
1.7.2 Tile layers

There are two types of tile layers available for the geographical display.

OpenStreetMap tiles

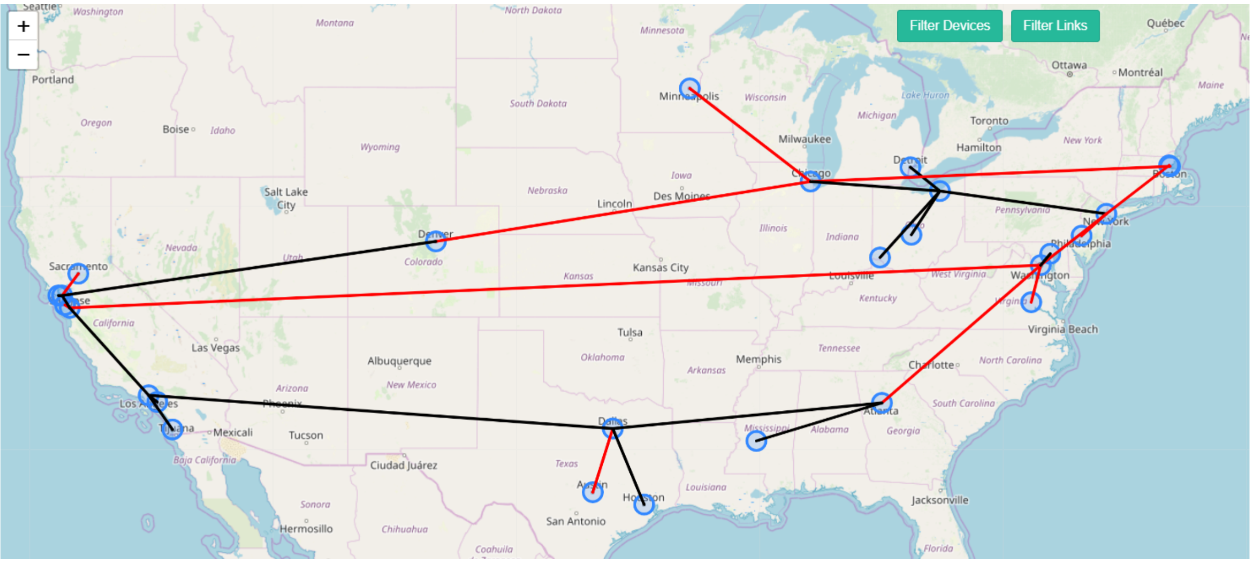


Google Map tiles



1.7.3 Marker Types

There are three different types of markers: images, circle, and circle marker. Displaying images can have an impact on performance above 10K devices; in that case, it is best to use circles or circle markers for scalability.

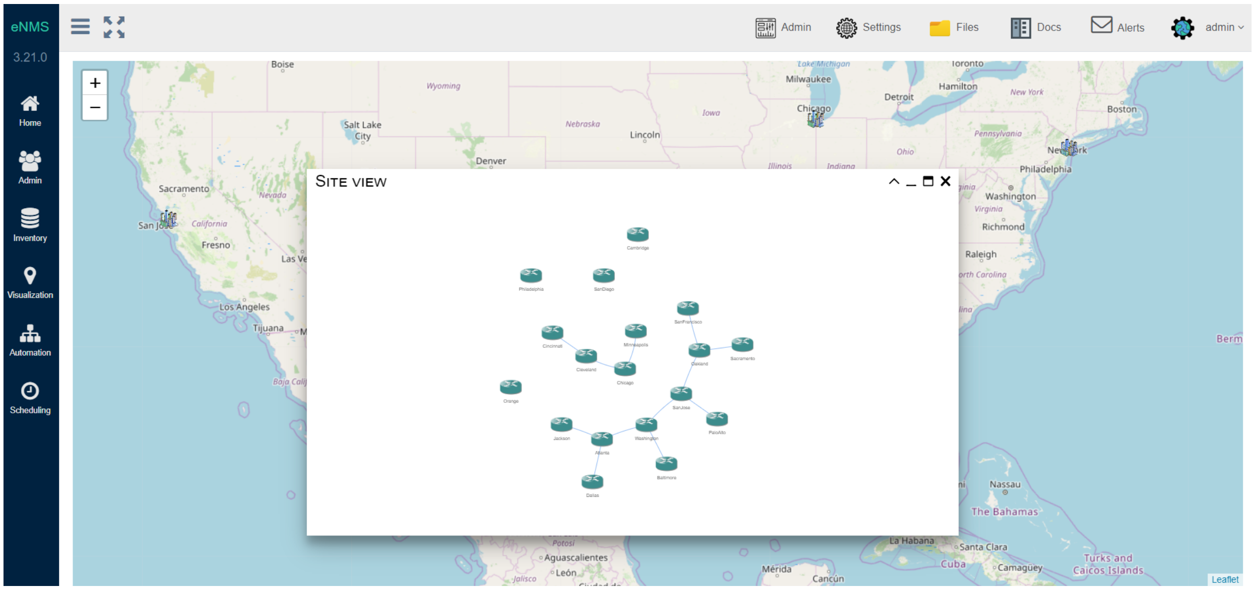


You can change the type of marker and tiles from the right-click menu. You can also configure in the settings which are used by default when you open the view.

1.7.4 Site View

The geographical view displays all devices at their GPS coordinates. If several devices are colocated (e.g same building), they can be grouped in a site. A site is a pool with a longitude and a latitude. The site view displays all sites on the map.

Clicking on a site will open a panel where all pool devices and links are organized in a logical and visually pleasing fashion using a force-based algorithm.



1.8 Services

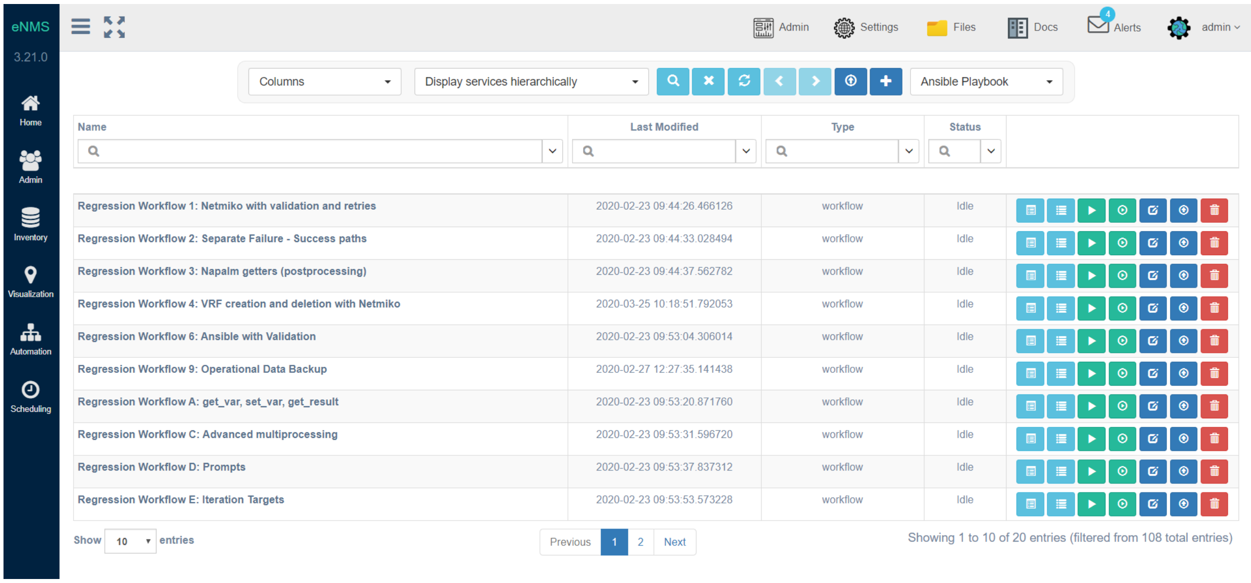
Services provide the smallest unit of automation in eNMS. Each service type provides unique functionality that is easily configured to perform complex operations in the network. Examples: remote command execution, ReST API calls, Ansible playbook execution, and many more.

Services can be powerful on their own, (e.g. ping all devices in the network and send a status email). They can also be combined within workflows to automate complex operations such as device upgrades.

Each service type provides a form for configuring its unique functionality. Common forms are available on every service for defining device targets, iteration, retries, pre and post processing, result validation, notifications, etc.

eNMS comes with a number of "default" services based on network automation frameworks such as `netmiko`, `napalm` and `ansible`, but you are free to create your own custom service types. Each service must return a python dictionary as a result.

All services are displayed in *Automation / Services*, where you can create new services and edit, duplicate, delete, and run existing ones.



Name	Last Modified	Type	Status
Regression Workflow 1: Netmiko with validation and retries	2020-02-23 09:44:26.466126	workflow	Idle
Regression Workflow 2: Separate Failure - Success paths	2020-02-23 09:44:33.028494	workflow	Idle
Regression Workflow 3: Napalm getters (postprocessing)	2020-02-23 09:44:37.562782	workflow	Idle
Regression Workflow 4: VRF creation and deletion with Netmiko	2020-03-25 10:18:51.792053	workflow	Idle
Regression Workflow 6: Ansible with Validation	2020-02-23 09:53:04.306014	workflow	Idle
Regression Workflow 9: Operational Data Backup	2020-02-27 12:27:35.141438	workflow	Idle
Regression Workflow A: get_var, set_var, get_result	2020-02-23 09:53:20.871760	workflow	Idle
Regression Workflow C: Advanced multiprocessing	2020-02-23 09:53:31.596720	workflow	Idle
Regression Workflow D: Prompts	2020-02-23 09:53:37.837312	workflow	Idle
Regression Workflow E: Iteration Targets	2020-02-23 09:53:53.573228	workflow	Idle

Showing 1 to 10 of 20 entries (filtered from 108 total entries)

A service type is a Python script that performs an action. A service type is defined by:

- A **model** class: the service parameters, and what the service is doing via a `job` function.
- A **form** class: the different fields eNMS displays in the UI, and their corresponding validation.

Note: You can also export services: this creates a YAML file with all service properties in the `files / exported_services` directory. This allows migrating services from one VM to another if you are using multiple VMs.

1.8.1 Service Panel

Section `General`

Main Parameters

- `Scoped Name` (**mandatory**) Must be unique either within the enclosing workflow or unique among top-level services.
- `Name` (**display only**) Fully qualified service name including all workflow nesting and a **[Shared]** tag.
- `Service Type` (**display only**) The service type of the current service instance.
- `Shared` Checked for **Shared** services. Once set, this value cannot be changed.

Note: Services can be standalone, shared, or scoped within a workflow. Shared services (or subworkflows) can exist inside multiple workflows, and a change to a shared service affects all workflows that use it. A service which is scoped within a workflow, either by creating the service inside the workflow or by deep copying the service into the workflow, exists only inside that workflow, so changes to it only affect its parent workflow. A standalone service exists outside of any workflow. A superworkflow acts as a template or wrapper around another workflow and allows for services to be run before and after the main workflow (which exists inside the superworkflow as a Placeholder). Because multiple workflows can specify the same superworkflow, the superworkflow acts as if it is shared.

- `Workflows` (**display only**) Displays the list of workflows that reference the service.

- `Description` / `Vendor` / `Operating System` Useful for filtering services in the table.
- `Initial Payload` User-defined dictionary that can be used anywhere in the service.

Note: The retry will affect only the devices for which the service failed. Let's consider a service configured to run on 3 devices D1, D2, and D3 with 2 "retries". If it fails on D2 and D3 when the service runs for the first time, eNMS will run the service again for D2 and D3 at the first retry. If D2 succeeds and D3 fails, the second and last retry will run on D3 only.

- `Number of retries` (default: `0`) Number of retry attempts when the service fails (if the service has device targets, this is the number of retries for each device).
- `Time between retries (in seconds)` (default: `10`) Number of seconds to wait between each attempt.
- `Maximum number of retries (loop prevention mechanism)` (default: `100`) Used to prevent infinite loops in workflows with circular paths.
- `Logging` (default: `Info`) The log level to use when running the service; it governs logs written to the log window in the UI, as well as the logs that are written to the log files.

Workflow Parameters

This section contains the parameters that apply **when the service runs inside a workflow only**.

- `Preprocessing` Section where you can write a python script that will run before the service is executed. If the service has device targets, the code will be executed for each device independently, and a `device` global variable is available. Note: Preprocessing is executed for standalone services and those within a workflow.
- `Skip` If ticked, the service is skipped.
- `Skip Query` This fields expect a python expression that evaluates to either `True` or `False`. The service will be skipped if `True` and will run otherwise.
- `Skip Value` Defines the success value of the service when skipped (in a workflow, the success value will define whether to follow the success path (success edge) or the failure path (failure edge).
- `Maximum number of runs` (default: `1`) Number of times a service is allowed to run in a workflow
- `Time to Wait before next service is started (in seconds)` (default: `0`) Number of seconds to wait after the service is done running.

Custom Properties

The Custom Properties section allows each instance of eNMS to add extra properties to the service form. Additional information for these fields may be available using the help icon next to the field label.

The location for the help file can specified in the `setup/properties.json` file like this:

  - "help": "custom/impacting"

Section `Specific`

This section contains all parameters that are specific to the service type. For instance, the "Netmiko Configuration" service that uses Netmiko to push a configuration will display Netmiko parameters (delay factor, timeout, etc) and a field to enter the configuration that you want to push.

The content of this section is described for each service in the `Default Services` section of the docs.

Section `Targets`

Devices

Most services are designed to run on devices from the inventory. There are three properties for selecting devices. The full list of targets is the union of all devices coming from these properties.

- `Run Method` Defines whether the service should run once, or if it should run once per device. Most default services are designed to run once per device.
- `Devices` Direct selection by device names
- `Pools` and `Update pools before running`
  - `Pools` Direct selection from pools. The set of all devices from all selected pools is used.
  - `Update pools before running` When selected, the pools are updated before reading their set of devices.
- `Device query` and `Query Property Type` Programmatic selection with a python query
  - `Device query` Query that must return an **iterable** (e.g python list) of **strings (either IP addresses or names)**.
  - `Query Property Type` Indicates whether the iterable contains IP addresses or names, for eNMS to look up actual devices from the inventory.
- `Multiprocessing` Run on devices **in parallel** instead of **sequentially**. - Only standalone services and services run in a workflow using a service by service run method benefit from this option. - Services in a workflow with run method **Run the workflow device by device** only have a single device. Instead, use multiprocessing on the workflow.
- `Maximum Number of Processes` (default: `15`) The maximum number of concurrent threads for this service when multiprocessing is enabled.

Iteration

Multiple actions are sometimes necessary when the service is triggered for a target device. Use iteration devices when those actions should be performed on a set of devices related to the current target device. Use iteration values when the actions should be performed on the current target device.

- `Iteration Devices` Query that returns an **iterable** (e.g. Python list) of **strings (either IP addresses or names)**.
  - The service is run for each device as the target device, allowing operations against a set of devices related to the original target.
  - `Iteration Devices Property` Indicates whether iterable `Iteration Devices` contains IP addresses or names, for eNMS to look up actual devices from the inventory.
- `Iteration Values` Query that returns an **iterable** (e.g. Python list) of **strings**.
  - The service is run for each value.
  - `Iteration Variable Name` Python variable name to contain each successive value from the `Iteration Values` query.

Section `Result`

The `Result` section defines operations on the service result. Each form group offers a different type of results operation. These operations are performed in the order found on the `Result` page. Result operations are executed for each device for `Run method` **Run the service once for each device**, and are executed only once for `Run method` **Run the service once**.

Conversion and Postprocessing

- `Conversion Method` The type of automatic conversion to perform on the service result.
  - `No conversion` (default) Use the result with no modification.
  - `Text` Convert the result to a python string.
  - `JSON` Convert a string representing JSON data to a python dictionary.
  - `XML` Convert a string representing XML data to a python dictionary.

Python can be used to inspect or modify the service result. This is typically used to perform complex validation or to extract values from the result for use in subsequent services.

- `Postprocessing Mode` Control whether or not the `Postprocessing` script is executed
  - `Always run` (**default**) The `Postprocessing` script will execute for each device
  - `Run on success only`
  - `Run on failure only`
- `PostProcessing` A python script to inspect or update the current result.
  - Variable **results**
    - Contains the full results dictionary for the current device, exactly as seen in the results view.
      - Changes to this dictionary are reflected in the final result of the service.
    - **results[“success”]** The overall service status.
    - **results[“result”]** The resulting data from running the service.
  - See *Using python code in the service panel* for the full list of variables and functions.

Validation

Validation can consist of:

- Text matching: looking for a string in the result, or matching the result against a regular expression.
- Dictionary matching: check that a dictionary is included or equal to the result.
- Anything else: you can use python code to change the result, including the value of the `success` key.

- `Validation Method` The validation method depends on whether the result is a string or a dictionary.
  - `No validation` No validation is performed
  - `Text match` Matches the result against `Content Match` (string inclusion, or regular expression if `Match content against Regular expression` is selected)
  - `dictionary Equality` Check for equality against the dictionary provided in `Dictionary Match`

  - `dictionary Inclusion` Check that all `key : value` pairs from the dictionary provided in `Dictionary Match` can be found in the result.

- `Negative Logic` Reverses the `success` boolean value in the results: the result is inverted: a success becomes a failure and vice-versa. This prevents the user from using negative look-ahead regular expressions.
- `Delete spaces before matching` (`Text` match only) All whitespace is stripped from both the output and `Content Match` before comparison to prevent these differences from causing the match to fail.

Notification

When a service finishes, you can choose to receive a notification with the results. There are three types of notification:

- Mail notification: eNMS sends a mail to the address(es) of your choice.
- Slack notification: eNMS sends a message to a channel of your choice.
- Mattermost notification: same as Slack, with Mattermost.

You can configure the following parameters:

- `Send notification` Enable sending results notification
- `Notification Method` Mail, Slack or Mattermost.
- `Notification header` A header displayed at the beginning of the notification.
- `Include Result Link in summary`: whether the notification contains a link to the results.
- `Mail recipients` Must be a list of email addresses, separated by comma.
- `Display only failed nodes` the notification will not include devices for which the service ran successfully.

To set up the mail system, you must set the variable of the `mail` section in the settings. `server`, `port`, `use_tls`, `username`, `sender`, `recipients`. Besides, you must set the password via the `MAIL_PASSWORD` environment variable.

The `Mail Recipients` parameter must be set for the mail system to work; the *Admin / Administration* panel parameter can also be overriden from Step2 of the Service Instance and Workflow configuration panels. For Mail notification, there is also an option in the Service Instance configuration to display only failed objects in the email summary versus seeing a list of all passed and failed objects.

In Mattermost, if the `Mattermost Channel` is not set, the default `Town Square` will be used.

1.8.2 Using python code in the service panel

There are two types of fields in the service panel where the user is allowed to use pure python code: substitution fields (light blue background) and python fields (light red background). In these fields, you can use any python code, including a number of **variables** that are made available to the user.

Variables

- `device`
  - **Meaning**: this is the device on which the service is running.
  - **Type** Database Object.
  - **Available**: when the service is running on a device.

  - **Properties**: member attributes which can be referenced as `{{device.property}}`, such as `{{device.name}}` or `{{device.ip_address}}`, inside of forms. The following base properties are supported:
    - device.name
    - device.subtype
    - device.description
    - device.model
    - device.location
    - device.vendor
    - device.operating_system
    - device.os_version
    - device.ip_address
    - device.latitude
    - device.longitude
    - device.port
    - device.configuration
    - device.last_failure (last failure timestamp for configuration collection)
    - device.last_status (last status timestamp for configuration collection)
    - device.last_update (last update timestamp for configuration collection)
    - device.last_runtime (last runtime timestamp for configuration collection)
    - device.last_duration (last time duration for configuration collection)

Custom Device Properties (if defined in setup/properties.json)

Currently None

- `link`
  - **Meaning**: this is a link between devices
  - **Type**: Database Object.
  - **Available**: when the service is running on a device.
  - **Properties**: member attributes which can be referenced as `{{link.property}}`, such as `{{link.model}}` or `{{link.source_name}}`, inside of forms. The following base properties are supported:
    - link.name
    - link.description
    - link.subtype
    - link.model
    - link.source_name (source device name)
    - link.destination_name (destination device name)

Custom Link Properties (if defined in setup/properties.json)

Currently None

- `get_result` (see *Using the result of previous services*)
  - **Meaning**: Fetch the result of a service in the workflow that has already been executed.
  - **Type** Function.
  - **Return Type** Dictionary
  - **Available**: when the service runs inside a workflow.
  - **Parameters**:
    - `service` (**mandatory**) Name of the service
    - `device` (**optional**) Name of the device, when you want to get the result of the service for a specific device.
    - `workflow` (**optional**) If your workflow has multiple subworkflows, you can specify a subworkflow to get the result of the service for a specific subworkflow.
- `get_var`
  - **Meaning**: Retrieve a value by `name` that was previously saved in the workflow. Use `set_var` to save values. Always use the same `device` and/or `section` values with `get_var` that were used with the original `set_var`.
  - **Type** Function.
  - **Return Type** None
  - **Available**: always.
  - **Parameters**:
    - `name` Name of the variable
    - `device` (**optional**) The value is stored for a specific device.
    - `section` (**optional**) The value is stored in a specific “section”.
- `log` - **Meaning**: Write a - **Type**: - **Return Type**: None - **Available**: always. - **Parameters**:
  - **severity**: (**string**) Valid values in escalating priority order: **info**, **warning**, **error**, **critical**.
  - **message**: (**string**) Verbiage to be logged.
  - **device**: (**string**, **optional**) Associate log message to a specific device.
  - **app_log**: (**boolean**, **optional**) Write log message to application log in addition to custom logger.
  - **logger**: (**string**, **optional**) When specified, the log message is written to the named custom logger instead of the application log. Set **app_log** = True to send log message to both the custom and application logs. Contact the administrator to create a custom logger, if needed.
- `parent_device`
  - **Meaning**: parent device used to compute derived devices.
  - **Type** Database Object.
  - **Available**: when the iteration mechanism is used to compute derived devices.
- `result`
  - **Meaning**: this is the result of the current service.

  - **Type** Dictionary.
  - **Available**: after a service has run.
- `set_var`
  - **Meaning**: Save a value by `name` for use later in a workflow. When `device` and/or `section` is specified, a unique value is stored for each combination of device and section. Use `get_var` for value retrieval.
  - **Type** Function.
  - **Return Type** None
  - **Available**: always.
  - **Parameters**:
    - `name` Name of the variable
    - `device` (**optional**) The value is stored for a specific device.
    - `section` (**optional**) The value is stored in a specific "section".

Variables saved globally (i.e. set_var("var1", value) and for a device (i.e. set_var("var2", device=device.name)) are made available within every Python code can be used. Only device specific variables for the current device are available. Device specific variables override global variables of the same name.

- `settings`
  - **Meaning**: eNMS settings, editable from the top-level menu. It is initially set to the content of `settings.json`.
  - **Type** Dictionary.
  - **Available**: Always.
- `send_email` lets you send an email with optional attached file. It takes the following parameters:
  - `title` (mandatory, type `string`)
  - `content` (mandatory, type `string`)
  - `sender` (optional, type `string`) Email address of the sender. Default to the sender address of eNMS settings.
  - `recipients` (optional, type `string`) Mail addresses of the recipients, separated by comma. Default to the recipients addresses of eNMS settings.
  - `reply_to` (optional, type `string`) Single mail address for replies to notifications
  - `filename` (optional, type `string`) Name of the attached file.
  - `file_content` (optional, type `string`) Content of the attached file.

```
send_email(
    title,
    content,
    sender=sender,
    recipients=recipients,
    reply_to=reply_to,
    filename=filename,
    file_content=file_content
)
```

- `dict_to_string` - convert a dictionary to a string form with indentation. It takes the following parameters:
  - `input` (mandatory, type `dict` or `any`)

    - `depth` (optional, type `int`) - how many tabs to indent. Defaults to 0.

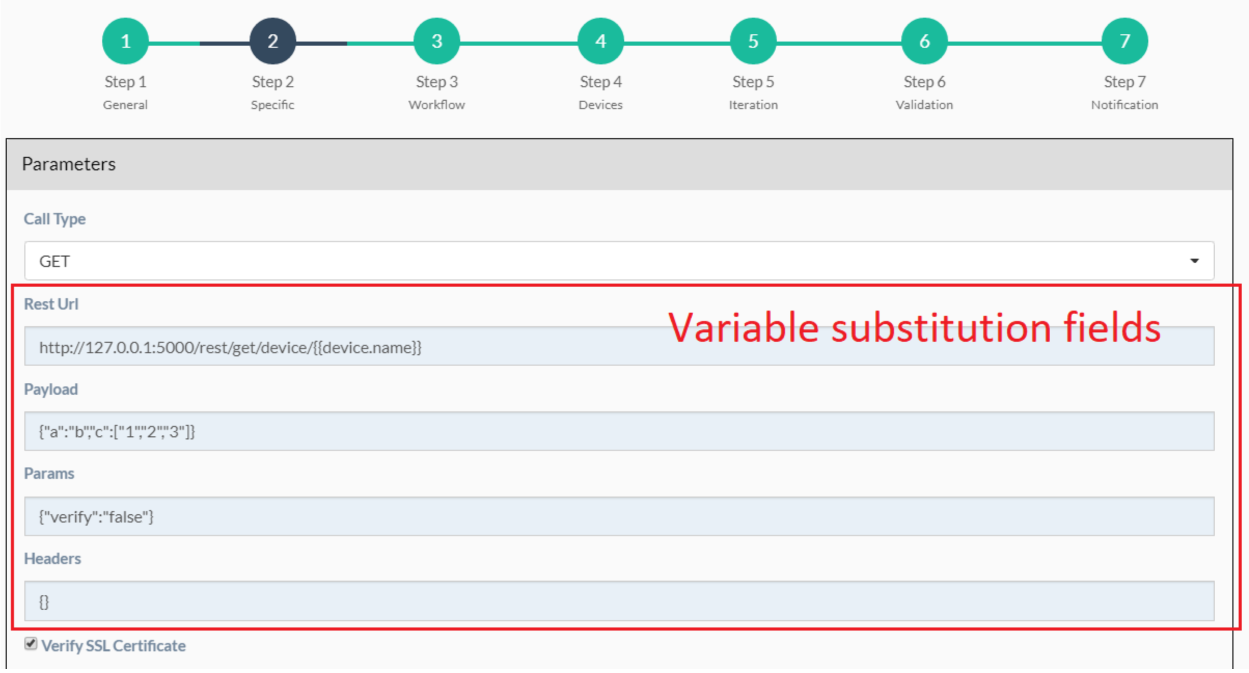
```
# Variable substitution example
{{dict_to_string(get_var("your_var_name"), depth=1)}}
```

```
# General example
test = {'key': 'value', 'key2': [45, 1135, 544]}
print(dict_to_string(test, depth=0))
# output:
key: value
key2:
        - 45
        - 1135
        - 544
```

- `target_devices`
  - **Meaning**: the full list of devices for the service.
  - **Type**: List of database objects.
  - **Available**: Always.
- `target_pools`
  - **Meaning**: the full list of pools for the service.
  - **Type**: List of database objects.
  - **Available**: Always.
- `workflow`
  - **Meaning**: current workflow.
  - **Type** Database Object.
  - **Available**: when the service runs inside a workflow.

Substitution fields

Substitution fields, marked in the interface with a light blue background, lets you include python code inside double curved brackets (`{{your python code}}`). For example, the URL of a REST call service is a substitution field. If the service is running on device targets, you can use the global variable `device` in the URL. When the service is running, eNMS will evaluate the python code in brackets and replace it with its value. See *Using python code in the service panel* for the full list of variables and functions available within substitution fields.



Parameters

Call Type: GET

Rest Url: http://127.0.0.1:5000/rest/get/device/{{device.name}}

Payload: {"a":"b","c":["1","2","3"]}

Params: {"verify":"false"}

Headers: {}

☑ Verify SSL Certificate

Running the service on two devices `D1` and `D2` will result in sending the following GET requests:

```
"GET /rest/get/device/D1 HTTP/1.1" 302 219
"GET /rest/get/device/D2 HTTP/1.1" 302 219
```

Python fields

Python fields, marked with a light red background, accept valid python code.

- In the `Device Query` field of the "Devices" section of a service. An expression that evaluates to an iterable containing the name(s) or IP address(es) of the desired inventory devices.
- In the `Skip Service if True` field of the "Workflow" section of a service. The expression result is treated as a boolean.
- In the `Query` field of the Variable Extraction Service. The expression result is used as the extracted value.
- In the code of a Python Snippet Service, or the `Preprocessing` and `Postprocessing` field on every service.

1.8.3 Custom Services

In addition to the services provided by default, you are free to create your own services. When the application starts, it loads all python files in `eNMS / eNMS / services` folder. If you want your custom services to be in a different folder, you can set a different path in the `settings`, section `paths`. Creating a service means adding a new python file in that folder. You are free to create subfolders to organize your own services any way you want: eNMS will automatically detect them. Just like all other services, this python file must contain a model and a form. After adding a new custom service, you must reload the application before it appears in the web UI.

1.8.4 Running a service

You can run a service from the "Services" page ("Run" button) or from the "Workflow Builder" (right-click menu).

There are two types of runs:

- Standard run: uses the service properties during the run.
- Parameterized run: a window is displayed with all properties initialized to the service

properties. You can change any property for the current run, but these changes won't be saved back to the service properties.

Results

A separate result is stored for each run of a service / workflow, plus a unique result for every device and for every service and subworkflow/superworkflow within a workflow. Each result is displayed as a JSON object. If the service is run on several devices, you can display the results for a specific device, or display the list of all "failed" / "success" device. In the event that retries are configured, the results dictionary will contain an overall results section, as well as a section for each attempt, where failed and retried devices are shown in subsequent sections starting with attempt2.

1.9 Default Services

1.9.1 Ansible Playbook Service

An `Ansible Playbook` service sends an ansible playbook to the devices. The output can be validated with a command / pattern mechanism, like the `Netmiko Validation Service`.

Configuration parameters for creating this service instance:

- `Playbook Path` path and filename to the Ansible Playbook. The location for displaying playbooks is configurable in eNMS settings.
- `Arguments` ansible-playbook command line options, which are documented here: (https://docs.ansible.com/ansible/latest/cli/ansible-playbook.html)
- `Pass device properties to the playbook` Pass inventory properties using –extra-vars to the playbook if checked (along with the options dictionary provided below). All device properties are passed such as `device.name` or `device.ip_address`
- `options` Additional –extra-vars to be passed to the playbook using the syntax {'key1':value1, 'key2': value2}. All inventory properties are automatically passed to the playbook using –extra-vars (if pass_device_properties is selected above). These options are appended.
- Ansible itself supports a number of standard return codes; these are returned in the results of the service and include:
  - 0 : OK or no hosts matched
  - 1 : Error
  - 2 : One or more hosts failed
  - 3 : One or more hosts were unreachable
  - 4 : Parser error
  - 5 : Bad or incomplete options
  - 99 : User interrupted execution
  - 250 : Unexpected error

Note: This Service supports variable substitution (as mentioned in the previous section) in the *arguments* and *options* input fields of its configuration form.

1.9.2 Netmiko Services

The Netmiko services provide the ability to perform multiple actions through an SSH connection. Below are the values common to each Netmiko service and the specifics for each are in their own sections.

Common Netmiko Parameters

- `Driver` This selects which Netmiko driver to use when connecting to the device. This is not used if `Use Device Driver` is checked.
- `Use Device Driver` If checked, the driver assigned to the device in the inventory will be used.
- `Enable mode` If checked, Netmiko should enter enablepriviledged mode on the device before applying the above configuration block. For the Linux driver, this means root/sudo.
- `Config mode` If checked, Netmiko should enter config mode
- `Fast CLI` If checked, Netmiko will disable internal wait states and delays in order to execute the service as fast as possible.
- `Timeout` Netmiko internal timeout in seconds to wait for a connection or response before declaring failure.
- `Delay factor` Netmiko multiplier used to increase internal delays (defaults to 1). Delay factor is used in the send_command Netmiko method. See here for more explanation: (https://pynet.twb-tech.com/blog/automation/netmiko-what-is-done.html)
- `Global delay factor` Netmiko multiplier used to increase internal delays (defaults to 1). Global delay factor affects more delays beyond Netmiko send_command. Increase this for devices that have trouble buffering and responding quickly.

Jump on connect Parameters

Jump on connect is designed to allow a second connection after connecting to the original device.

- `Jump to remote device on connect` If checked, the config items below will be processed for connection to the secondary device.
- `Command that jumps to device` Command to initiate secondary connection
- `Expected username prompt` Prompt expected when connecting secondary connection
- `Device username` The username to send when the expected username prompt is detected
- `Expected password prompt` Prompt expected when connecting secondary connection
- `Device password` The password to send when the expected password prompt is detected
- `Expected prompt after login` Prompt expected after successfully negotiating a connection
- `Command to exit device back to original device` Command required to exit the secondary connection

Netmiko Configuration Service

Uses Netmiko to send a list of commands to be configured on the devices.

Configuration parameters for creating this service instance:

- All Common Netmiko Parameters (see above)
- `Content` Paste a configuration block of text here for applying to the target device(s)
- `Commit Configuration` Calls netmiko `commit` function of the driver to commit the configuration
- `Exit config mode` Determines whether or not to exit config mode after complete
- `Config Mode Command` The command that will be used to enter config mode
- Advanced Netmiko Parameters
- `Strip command` Remove the echo of the command from the output (default: True)
- `Strip prompt` Remove the trailing router prompt from the output (default: True)

Note: This Service supports variable substitution (as mentioned in the previous section) in the *content* input field of its configuration form.

Netmiko File Transfer Service

Uses Netmiko to send a file to a device, or retrieve a file from a device. Only Cisco IOS and some Juniper devices are supported at this time for SCP file transfer.

Configuration parameters for creating this service instance:

- All Common Netmiko Parameters (see above)
- `Source file` Source absolute path and filename of the file to send
- `Destination file` Destination file; absolute path and filename to send the file to
- `File system` Mounted filesystem for storage on the default. For example, disk1:
- `Direction` Upload or Download from the perspective of running on the device
- `Disable_md5` Disable checksum validation following the transfer
- `Inline_transfer` Cisco specific method of transferring files between internal components of the device
- `Overwrite_file` If checked, overwrite the file at the destination if it exists

Netmiko Data Backup Service

This service uses Netmiko to send commands to store information from devices.

Target Property and Commands

- Property to update (e.g `Configuration`)
- `Commands` - This is a series of twelve commands that are used to pull data from the device.
- `Label` This is the label the data will be given in the results

Search Response and Replace

- Used to filter out unwanted information
- `Pattern` The pattern to search through the retrieved data to replace
- `Replace With` This is what will be substituted when the `pattern` is found.

Netmiko Prompts Service

Similar to Netmiko Validation Service, but expects up to 3 interactive prompts for your remote command, such as 'Are you sure? Y/N'. This service allows the user to specify the expected prompt and response to send for it.

Configuration parameters for creating this service instance:

- All Common Netmiko Parameters (see above)
- `Command` CLI command to send to the device
- `Confirmation1` Regular expression to match first expected confirmation question prompted by the device
- `Response1` response to first confirmation question prompted by the device
- `Confirmation2` Regular expression to match second expected confirmation question prompted by the device
- `Response2` response to second confirmation question prompted by the device
- `Confirmation3` Regular expression to match third expected confirmation question prompted by the device
- `Response3` response to third confirmation question prompted by the device

Note: This Service supports variable substitution (as mentioned in the previous section) in the *command* input field of its configuration form.

Netmiko Validation Service

Uses Netmiko to send commands to a device and validates the output to determine the state of that device. See the `Workflow` section for examples of how it is used in a workflow.

There is a `command` field and an `expect string` field in the Advanced Netmiko Parameters. eNMS will check if the expected pattern can be found in the output of the command. The values for a `pattern` field can also be a regular expression.

Configuration parameters for creating this service instance:

- All Common Netmiko Parameters (see above)
- All Validation parameters (see above)
- `Command` CLI command to send to the device

Also included in Netmiko Advanced Parameters: - `Expect String` This is the string that signifies the end of output. - `Auto Find Prompt` Tries to detect the prompt automatically.

Note: `Expect String` and `Auto Find Prompt` are mutually exclusive; both cannot be enabled at the same time. If the user does not expect Netmiko to find the prompt automatically, the user should provide the expected prompt instead.

- `Command to Enter Config Mode` The default command used to enter config mode is determined by the Netmiko driver. An alternate configuration mode entry command can be specified here to overrride the default. Examples are the Juniper `config private` and `config exclusive` commands that isolate configuration changes from other users.
- `Strip command` Remove the echo of the command from the output (default: True).
- `Strip prompt` Remove the trailing router prompt from the output (default: True).
- `Use Genie` Use Cisco's Genie implementation to create structured data from cli commands. (Currently does not work with some vendors. Refer to this link to see which CLI commands are currently supported: https://developer.cisco.com/docs/genie-docs/)

Note: This Service supports variable substitution (as mentioned in the previous section) in the *command* input field of its configuration form.

1.9.3 Napalm Services

Napalm connections are SSH connections to equipment in which a pre-defined set of data is retrieved from the equipment and presented to the user in a structured (dictionary) format.

Napalm Common Parameters

- `Driver` Which Napalm driver to use when connecting to the device
- `Use driver from device` If set to True, the driver defined at device level (`napalm_driver` property of the device) is used, otherwise the driver defined at service level (`driver` property of the service) is used.
- `Optional arguments` Napalm supports a number of optional arguments that are documented here: (https://napalm.readthedocs.io/en/latest/support/index.html#optional-arguments)

Napalm Data Backup

This service uses Napalm to pull data from devices and store it for later comparison and for historical tracking.

- All Napalm Common Parameters (See Above)
- `Configuration Getters` - Choose the configuration getter named 'Configuration'

Napalm Configuration service

Uses Napalm to configure a device.

Configuration parameters for creating this service instance:

- All Napalm parameters (see above)
- **`Action` There are two types of operations:**
  - `Load merge`: add the service configuration to the existing configuration of the target
  - `Load replace`: replace the configuration of the target with the service configuration
- `Content` Paste a configuration block of text here for applying to the target device(s)

Note: This service is supported by a limited set of products.

Note: This Service supports variable substitution (as mentioned in the previous section) in the *content* input field of its configuration form.

Napalm Getters service

Uses Napalm to retrieve a list of getters whose output is displayed in the logs. The output can be validated with a command / pattern mechanism like the `Netmiko Validation Service`.

Configuration parameters for creating this service instance:

- All Validation parameters (see above)
- All Napalm parameters (see above)
- `Getters` Choose one or more getters to retrieve; Napalm getters (standard retrieval APIs) are documented here: (https://napalm.readthedocs.io/en/latest/support/index.html#getters-support-matrix)

Note: This Service supports variable substitution (as mentioned in the previous section) in the *content_match* input field of its configuration form.

Napalm Ping service

Uses Napalm to connect to the selected target devices and performs a ping to a designated target. The output contains ping round trip time statistics. Note that the iosxr driver does not support ping, but you can use the ios driver in its place by not selecting `Use_device_driver`.

Configuration parameters for creating this service instance:

- All Napalm parameters (see above)
- `Count`: Number of ping packets to send
- `Size` Size of the ping packet payload to send in bytes
- `Source IP address` Override the source ip address of the ping packet with this provided IP
- `Timeout` Seconds to wait before declaring timeout
- `Ttl` Time to Live parameter, which tells routers when to discard this packet because it has been in the network too long (too many hops)
- `Vrf` Ping a specific virtual routing and forwarding interface

Napalm Rollback Service

Use Napalm to rollback a configuration.

Configuration parameters for creating this service instance:

- All Napalm parameters (see above)

Napalm Traceroute service

Uses Napalm to connect to the selected target devices and performs a traceroute to a designated target.

Configuration parameters for creating this service instance:

- All Napalm parameters (see above)
- `Source IP address` Override the source ip address of the ping packet with this provided IP
- `Timeout` Seconds to wait before declaring timeout
- `ttl` Time to Live parameter, which tells routers when to discard this packet because it has been in the network too long (too many hops)
- `vrf` Ping a specific virtual routing and forwarding interface

1.9.4 REST Call Service

Send a REST call (GET, POST, PUT or DELETE) to a URL with optional payload. The output can be validated with a command / pattern mechanism, like the `Netmiko Validation Service`.

Configuration parameters for creating this service instance:

- `Call Type` REST type operation to be performed: GET, POST, PUT, DELETE
- `Rest Url` URL to make the REST connection to
- `Payload` The data to be sent in POST Or PUT operation
- `Params` Additional parameters to pass in the request. From the requests library, params can be a dictionary, list of tuples or bytes that are sent in the body of the request.
- `Headers` Dictionary of HTTP Header information to send with the request, such as the type of data to be passed. For example, {“accept”:”application/json”,”content-type”:”application/json”}
- `Verify SSL Certificate` If checked, the SSL certificate is verified. Default is to not verify the SSL certificate.
- `Timeout` Requests library timeout, which is the number of seconds to wait on a response before giving up
- `Username` Username to use for authenticating with the REST server
- `Password` Password to use for authenticating with the REST server

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.5 Generic File Transfer Service

Transfer a single file to/from the eNMS server to the device using either SFTP or SCP.

Configuration parameters for creating this service instance:

- `Direction` Get or Put the file from/to the target device’s filesystem
- `Protocol` Use SCP or SFTP to perform the transfer
- `Source file` For Get, source file is the path-plus-filename on the device to retrieve to the eNMS server. For Put, source file is the path-plus-filename on the eNMS server to send to the device.

- `Destination file` For Get, destination file is the path-plus-filename on the eNMS server to store the file to. For Put, destination file is the path-plus-filename on the device to store the file to.
- `Missing Host Key Policy` If checked, auto-add the host key policy on the ssh connection
- `Load Known Host Keys` If checked, load host keys on the eNMS server before attempting the connection
- `Look For Keys` Flag that is passed to the paramiko ssh connection to indicate if the library should look for host keys or ignore.
- `Source file includes glob pattern (Put Direction only)` Flag indicates that for Put Direction transfers only, the above Source file field contains a Glob pattern match (https://en.wikipedia.org/wiki/Glob_(programming)) for selecting multiple files for transport. When Globing is used, the Destination file directory should only contain a destination directory, because the source file names will be re-used at the destination.
- `Max Transfer Size` This is that maximum packet size that will be used during transfer. This may adversely impact transfer times.
- `Window Size` This is the requested windows size during transfer. This may adversely impact transfer times.

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.6 ICMPTCP Ping

Implements a Ping from this automation server to the selected devices from inventory using either ICMP or TCP.

Configuration parameters for creating this service instance:

- `Protocol`: Use either ICMP or TCP packets to ping the devices
- `Ports (TCP ping only)` Which ports to ping (should be formatted as a list of ports separated by a comma, for example “22,23,49”).
- `Count`: Number of ping packets to send
- `Timeout` Seconds to wait before declaring timeout
- `Ttl` Time to Live parameter, which tells routers when to discard this packet because it has been in the network too long (too many hops)
- `Packet Size` Size of the ping packet payload to send in bytes

1.9.7 UNIX Command Service

Runs a UNIX command **on the server where eNMS is installed**.

Configuration parameters for creating this service instance: - `Command`: UNIX command to run on the server

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.8 UNIX Shell Service

Runs a BASH script on the server where eNMS is installed. - `Source Code` Bash code to be run on the server.

1.9.9 Mail Notification Service

This service is used to send an automatically generated email to a list of recipients.

- `Title` Subject Line of the Email
- `Sender` If left blank, the email address set in the `settings.json` will be used.
- `Recipients` A comma delimited list of recipients for the email
- `Reply-to Address` If left blank, the reply-to address from `settings.json` is used. If populated, this email will be used by anyone replying to the automated email notification.
- `Body` This is the body of the email.

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.10 Mattermost Notification Service

This service will send a message to a mattermost server that is configured in the site settings.

- `Channel` The channel the message will be posted to
- `Body` The body of the message that will be posted to the above channel

---

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.11 Python Snippet Service

Runs any python code.

In the code, you can use the following variables / functions : - `log`: function to add a string to the service logs. - `parent`: the workflow that the python snippet service is called from. - `save_result`: the results of the service.

Additionally, you can use all the variables and functions described in the “Advanced / Python code” section of the docs.

Configuration parameters for creating this service instance: - `Source code`: source code of the python script to run.

1.9.12 Payload Extraction Service

Extract some data from the payload with a python query, and optionally post-process the result with a regular expression or a TextFSM template.

Configuration parameters for creating this service instance: - `Variable Name`: name of the resulting variable in the results. - `Python Extraction Query`: a python query to retrieve data from the payload. - `Post Processing`: choose the type of post-processing: Use Value as Extracted, Apply Regular Expression(findall), or TextFSM template. - `Regular Expression/ TestFSM Template Text`: regular expression or TextFSM template, depending on the value of the “Match Type1”. - `Operation` Choose the operation type: Set/Replace, Append to a list, Extend List, Update dictionary - Same fields replicated twice (2,3 instead of 1): the service can extract / post-process up to 3 variables.

1.9.13 Payload Validation Service

Extract some data from the payload, and validate it against a string or a dictionary. This is used for conducting extra validations of a prior service's result later in a workflow.

Configuration parameters for creating this service instance:

- `Python Query`: a python query to retrieve data from the payload.

1.9.14 Slack Notification Service

This service will send a message to the slack server that is configured in the site settings.

- `Channel` The channel the message will be posted to
- `Token` API Token to allow communications to the workspace
- `Body` The body of the message that will be posted to the above channel

---

Note: This Service supports variable substitution (as mentioned in the previous section) in the *url* and *content_match* input fields of its configuration form.

1.9.15 Topology Import Service

Import the topology from an instance of LibreNMS, Netbox or OpenNMS.

- `Import Type` Choose LibreNMS, Netbox or OpenNMS

Netbox

Configuration settings and options for importing topology from a Netbox Server

- `Netbox Address` Address for the netbox server
- `Netbox Token` API token to allow netbox interactions

OpenNMS

Options available for importing a known set of devices from OpenNMS

- `Opennms Adress` Address for the OpenNMS server
- `Opennms Devices` A list of devices to query in the OpenNMS server
- `Opennms Login` Login for the OpenNMS Server
- `Opennms Password` Password for the OpenNMS Server

LibreNMS

Configuration settings and options for importing topology from LibreNMS - `Librenms Address` Address for the LibreNMS Server - `Librenms Token` API token for allowing interaction with LibreNMS.

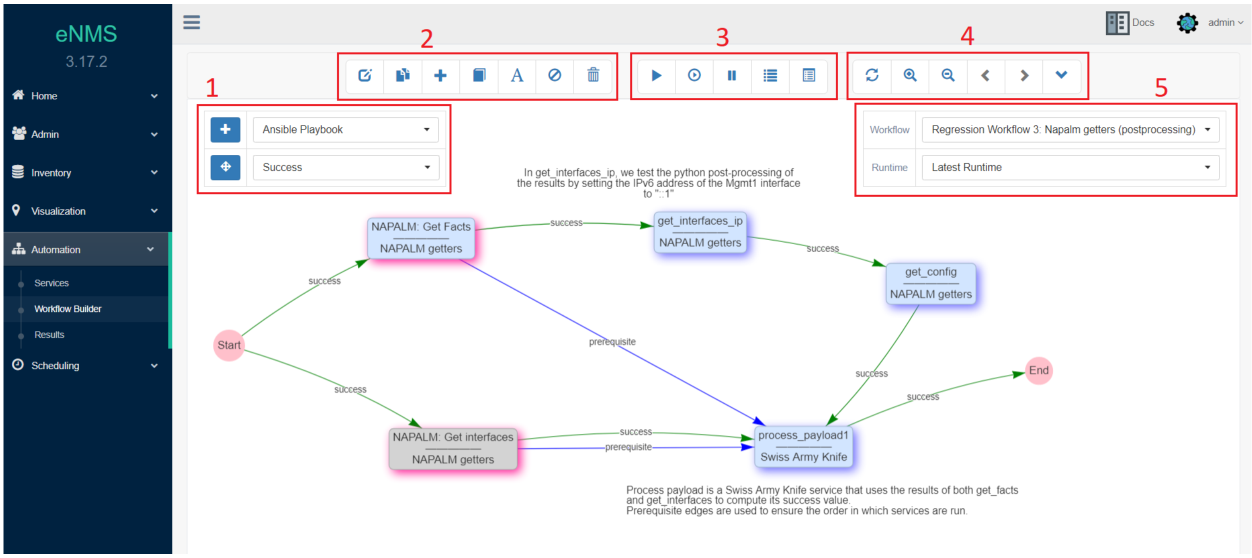
1.10 Workflow System

A workflow is a graph of services connected with `success` and `failure` edges. If a service is executed successfully, the workflow continues down the `success` path to the next service, otherwise it goes down the `failure` path. A workflow is considered to have run successfully if the "End" service is reached. For this reason, a `failure` edge should be used for:

- Recovery from a failure, using some corrective action, that allows the workflow to then continue on the `success` path
- Cleanup or finalization of the state of a device, following the failure, before the workflow ends

Workflows are managed from the *Workflow Builder*. When a workflow is running, the results are automatically updated in real-time in the workflow builder.

1.10.1 Workflow Builder



In get_interfaces_ip, we test the python post-processing of the results by setting the IPv6 address of the Mgmt1 interface to "::1"

Process payload is a Swiss Army Knife service that uses the results of both get_facts and get_interfaces to compute its success value.
Prerequisite edges are used to ensure the order in which services are run.

- Section 1: services and edges creation.
  - The first row lets you choose a service type and click on the "plus" button to create a new service that will be added to the workflow.
  - On the second row, you can change the mode to "Edge creation" and select which type of edge you want to create.
- Section 2: edit and duplicate the current workflow, create a new top-level workflow, add existing services to the workflow, create labels, skip or unskip services, and delete selected services and edges.
- Section 3: run or stop a workflow, and display the workflow logs and results.
- Section 4: refresh the workflow, zoom and unzoom, move to the previous or next workflow, and move to the selected subworkflow.
- Section 5: choose which workflow to display, and which results. The `Latest Runtime` results selection will make workflow builder continue refreshing on the most recent run activity for this particular workflow on this instance of eNMS. The `Normal Display` will show the workflow without run results. Selecting a specific runtime result will prevent refreshing for any new runtimes and will continue to display that historical run.

Note: Some of these actions are also available from the right-click menus (clicking on the background, on a service or on an edge generate different menus).

1.10.2 Workflow Devices

The devices used when running a workflow depend on the workflow `Run Method` that you can configure in the edit panel, section `Targets`. There are three run methods for a workflow:

Device by device

- Uses the devices configured at **workflow** level.
- Multiprocessing, if desired, must be enabled at **workflow** level.
- Workflow will run for each device independently: one at a time if multiprocessing is disabled, or in parallel otherwise.

Service by service using workflow targets

- Uses the devices configured at **workflow** level.
- Multiprocessing, if desired, must be enabled at **service** level.
- The workflow will run one service at a time, but each device can follow a different path depending on the results of each service for that device.

Service by service using service targets

- Uses the devices configured at **service** level.
- Multiprocessing, if desired, must be enabled at **service** level.
- The workflow will run one service at a time. A service is considered successful if it ran successfully on all of its targets (if it fails on at least one target, it is considered to have failed).

1.10.3 Transfer of data among services

Using the result of previous services

When a service starts, you can have access to the results of all services in the workflow that have already been executed with a special function called `get_result`. This and other functions are available for use in form fields that support variable substitution (designated in the UI with a colored tint). The result of a service is the dictionary that is returned by the `job` function of the service, and calling `get_result` will return that dictionary. There are two types of results: top-level and per-device. If a service runs on 5 devices, 6 results will be created: one for each device and one top-level result for the service itself.

Examples:

- `get_result("get_facts")` Get the top-level result for the service `get_facts`
- `get_result("get_interfaces", device="Austin")` Get the result of the device `Austin` for the `get_interfaces` service.

- `get_result("get_interfaces", device=device.name)` Get the result of the current device for the `get_interfaces` service.
- `get_result("Payload editor")["runtime"]` Get the `runtime` key of the top-level result of the `Payload editor` service.

The `get_result` function works everywhere that python code is accepted.

1.10.4 Saving and retrieving values in a workflow

You can define variables in the payload with the `set_var` function, and retrieve those variables' data from the payload with the `get_var` function using the first positional argument (the variable name) and the same optional arguments defined in `set_var`. If neither of the optional arguments are used the variable will be global.

- The first argument for set_var is positional and names the variable being set.
- The second argument for set_var is positional and assigns the value for the variable.
- An optional argument for set_var uses keyword "device", which can scope the variable to the device the service is using when the variable is set.
- An optional argument for set_var uses keyword "section", which can scope the variable to a user provided custom scope.

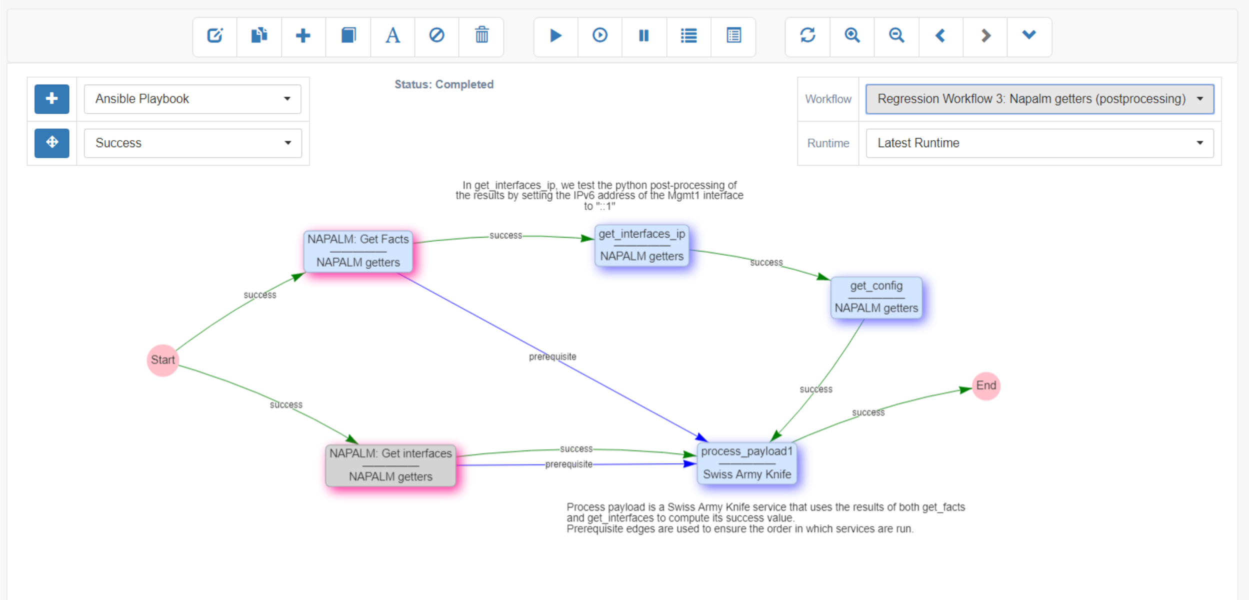
Examples:

```
set_var("global_variable", value=1050)
get_var("global_variable")
set_var("variable", "variable_in_variables", section="variables")
get_var("variable", section="variables")
set_var("variable1", 999, device=device.name)
get_var("variable1", device=device.name)
set_var("variable2", "1000", device=device.name, section="variables")
get_var("variable2", device=device.name, section="variables")
set_var("iteration_simple", "192.168.105.5", section="pools")
get_var("iteration_simple", section="pools")
set_var("iteration_device", devices, section="pools", device=device.name)
get_var("iteration_device", section="pools", device=device.name)
```

1.10.5 Miscellaneous

Service dependency

If a service must be run after another service, you can force that order by creating a `Prerequisite` edge. In the example below, the service `process_payload1` uses the results from `Get Facts` and `Get Interfaces`. By creating two prerequisite edges, we ensure that `process_payload1` will not be run until both `Get Facts` and `Get Interfaces` have been executed.



In get_interfaces_ip, we test the python post-processing of the results by setting the IPv6 address of the Mgmt1 interface to "::1"

Process payload is a Swiss Army Knife service that uses the results of both get_facts and get_interfaces to compute its success value.
Prerequisite edges are used to ensure the order in which services are run.

Workflow Restartability

A workflow can be restarted with any services set as “Entry points” and with the runtime payload from a previous run. This is useful if you are testing a workflow with a lot of services, or some of the services in the workflow are not idempotent, so you don’t want it to restart from the Start service each time.

Note: A restart can be performed inside a subworkflow such that the runtime result of the parent workflow is pulled in to satisfy variable dependencies.

The target device list for restarting is limited to the original target list as specified in the workflow or service. You cannot change the target list, for example to only failed devices, and restart a workflow that picks up and continues a historical runtime.

Connection cache

When using netmiko and napalm services in a workflow, eNMS will cache and reuse the connection automatically. In the `Specifics` section of a service, there are two properties to change this behavior :

- `Start New Connection`: **before the service runs**, the current cached connection is discarded and a new one is started.
- `Close Connection`: once the service is done running, the current connection will be closed.

Waiting times

Services and Workflows have a `Waiting time` property: this tells eNMS how much time it should wait after the service has run before it begins the next service.

A service can also be configured to “retry” if the results returned are not as designed. An example execution of a service in a workflow, in terms of waiting times and retries, is as follows:

```
First try
time between retries pause
Retry 1
time between retries pause
Retry 2  (Successful, or only 2 Retries specified)
Waiting time pause
```

Superworkflow

Just as a workflow can contain a subworkflow to subdivide and encapsulate related functionality, a workflow can also designate a superworkflow in its settings. The superworkflow allows for services to be run before and after the main workflow and using a potentially different workflow traversal mode (service by service or device by device). Superworkflow functions like a document template so that activities common to all workflows can be performed. When the same superworkflow is used by multiple main workflows, it behaves like a shared service: a change to the superworkflow affects all workflows that use it. In the superworkflow definition in Workflow Builder, the position of the main workflow is designated by adding the `Placeholder` service to the graph. And in the main workflow definition, the superworkflow must be selected from the list of existing workflows.

1.11 Scheduling

1.11.1 Tasks

Creating Tasks

Instead of running services immediately, they can be scheduled by creating a task, from *Scheduling / Tasks*. You need to enter a name, which service you want the task to execute, and a scheduling mode.

There are two scheduling modes:

- **Date Scheduling**: enter a start date and optionally a frequency and an end date. If a `Frequency` is defined without an end date, the service will keep running until manually stopped.
- **Cron Scheduling**: enter a crontab expression - Crontab format reference - reminder:

  ```
  minute(0-59) hour(0-23) day-of-month(1-31) month(1-12) day-of-week(0-6)
  ```

- Example - every 15 minutes on Tuesday and Thursday:

  ```
  */15 * * * 2,4
  ```

When creating a task, you can select a list of devices and pools. If these fields are left empty, the service will run on its own targets. Otherwise, the task targets (all selected devices, plus all devices of all selected pools) will override the service targets when the service runs.

A task can also have a payload (dictionary) that will be passed to the service when it runs.

Managing Tasks

Newly created tasks are set to **paused** by default. Tasks can be paused and resumed. Active tasks display the date that they will next be run by the scheduler, as well as the amount of time left until then.

Troubleshooting: if a task is **Active** without a next run date, it is likely that the scheduled job database was lost. Try editing the Task and saving it. This will restore the scheduled portion of the Task.

Timezone Considerations

When specifying a start time, you must take into account the server's time zone configuration. Normally, this is in UTC (Coordinated Universal Time). To run a scheduled task at a specific local time, the start time OR crontab expression will need to be adjusted depending on the local time zone - and Daylight Savings Time if applicable:

- Time zone conversion - CST/CDT
    - Central Time zone (CST) is UTC-6 (Fall back)
    - Central Daylight Time (CDT) is UTC-5 (Spring forward)
- Example
    - Reporting service: every 8:00 AM on Monday/Wednesday/Friday
    - Crontab - hour value is either **13:00 or 14:00** depending on the time of the year!

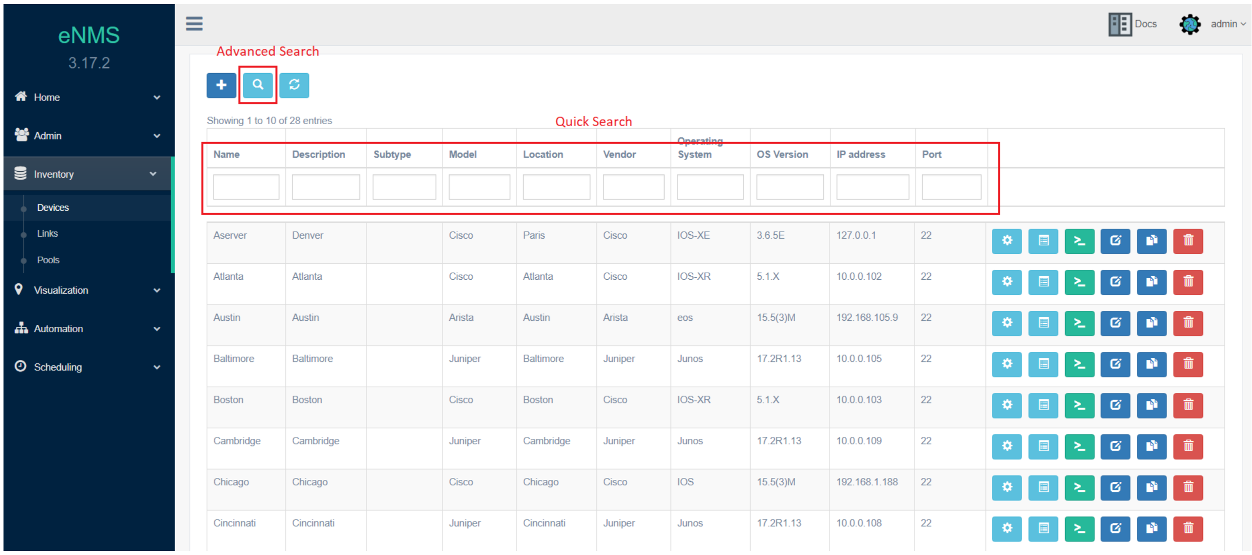
```
0 13 * * 1,3,5
```

1.12 Search System

All objects stored in the database are displayed in tables that you can filter using the following options:

1.12.1 Quick search

For a quick search, you can use the textbox displayed above columns in the table as pictured by the white magnifying glass. In it's default state (Inclusion), this filter finds matches that include the text provided and is not case sensitive (i.e. a or A). The drop down to the right of the magnifying glass has options to change the filter to return on exact matches (Equality), which is also not case sensitive. And, the last quick filter option is based on a user provided Regular Expression.



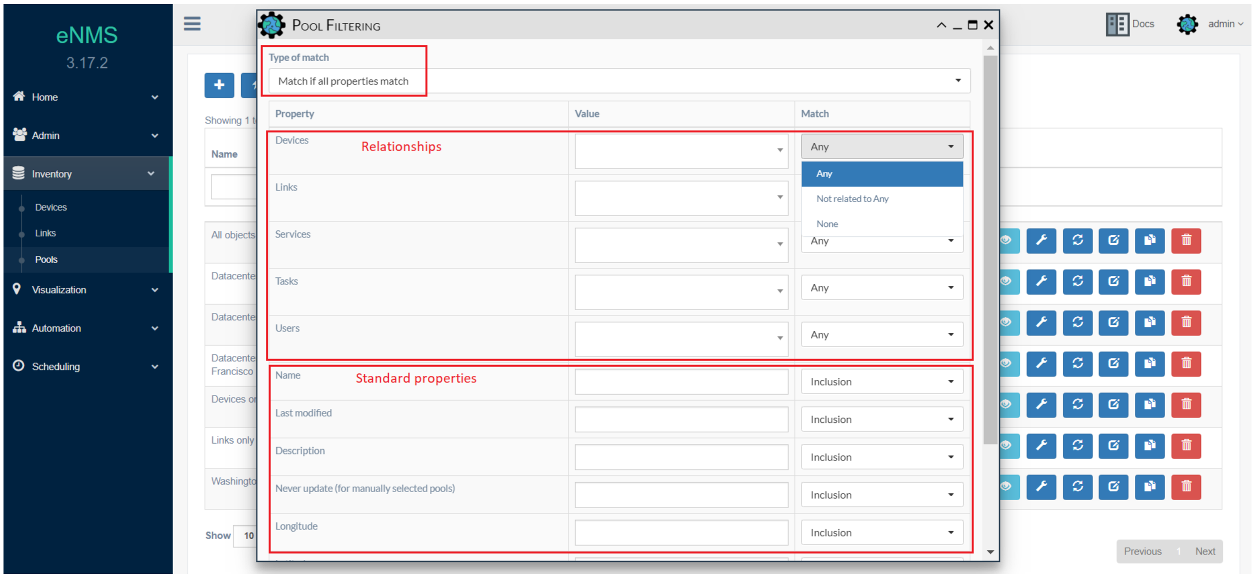
Showing 1 to 10 of 28 entries

Name	Description	Subtype	Model	Location	Vendor	Operating System	OS Version	IP address	Port
Aserver	Denver		Cisco	Paris	Cisco	IOS-XE	3.6.5E	127.0.0.1	22
Atlanta	Atlanta		Cisco	Atlanta	Cisco	IOS-XR	5.1.X	10.0.0.102	22
Austin	Austin		Arista	Austin	Arista	eos	15.5(3)M	192.168.105.9	22
Baltimore	Baltimore		Juniper	Baltimore	Juniper	Junos	17.2R1.13	10.0.0.105	22
Boston	Boston		Cisco	Boston	Cisco	IOS-XR	5.1.X	10.0.0.103	22
Cambridge	Cambridge		Juniper	Cambridge	Juniper	Junos	17.2R1.13	10.0.0.109	22
Chicago	Chicago		Cisco	Chicago	Cisco	IOS	15.5(3)M	192.168.1.188	22
Cincinnati	Cincinnati		Juniper	Cincinnati	Juniper	Junos	17.2R1.13	10.0.0.108	22

Note: If you use several of these fields the results will be based on all fields where input was provided. (i.e. it uses a boolean `AND`).

1.12.2 Advanced search

Additional filter criteria that will allow you to specify relationships between objects based on the following.



Type of match: Match if all properties match

Property	Value	Match
Devices		Any
Links		
Services		Any
Tasks		Any
Users		Any
Name		Inclusion
Last modified		Inclusion
Description		Inclusion
Never update (for manually selected pools)		Inclusion
Longitude		Inclusion

- **Type of match**: Whether a match requires all properties to match (boolean `AND`), or any of them (boolean `OR`).
- **Relationships**: All tables stored in the database are associated by SQL relationships. For example, a pool contains devices and links and a device can belong to one or more pools: there is a many-to-many relationship

between pools and devices. The advanced search system can perform a search based on these relationships. For a given relationship, you can choose between 3 types of match:

  - `Any`: match if related to at least one of the selected objects.
  - `All`: match if related to all selected objects.
  - `Unrelated`: match if not related to any of the selected objects.
  - `None`: match if the are no related objects.

- **Standard properties**: You can filter based on inclusion, equality or a regular expression.

Note: The search based on regular expression only works if the database you are using supports it. PostgreSQL and MySQL support regular expressions, but SQLite does not.

1.13 Configuration Management

eNMS can be used as a device configuration backup tool, like Oxidized/Rancid, with the following features:

- Poll network devices and store the latest configuration in the database
- Store any operational data that can be retrieved from the device CLI (e.g `show version`, `get facts` etc.)
- Search for any text or regular-expression in all configurations
- Download device configuration to a local text file
- Use the REST API support to return a specified device's configuration
- Export all configurations to a remote Git repository (e.g. Gitlab)
- View git-style differences between various revisions of a configuration

1.13.1 Device configuration

Configurations are retrieved by a service called "Netmiko Data Backup", which:

- Fetch the device configuration using Netmiko
- Updates the device `Configuration` property

For some devices, the configuration cannot be retrieved with only a netmiko command. In this case, you can either use the "Napalm Data Backup" service to substitute a napalm getter, such as get_config, in place of retrieving the configuration via CLI command, or you can create your own configuration backup service(s) if required. Targets are defined at the service level, like any other services.

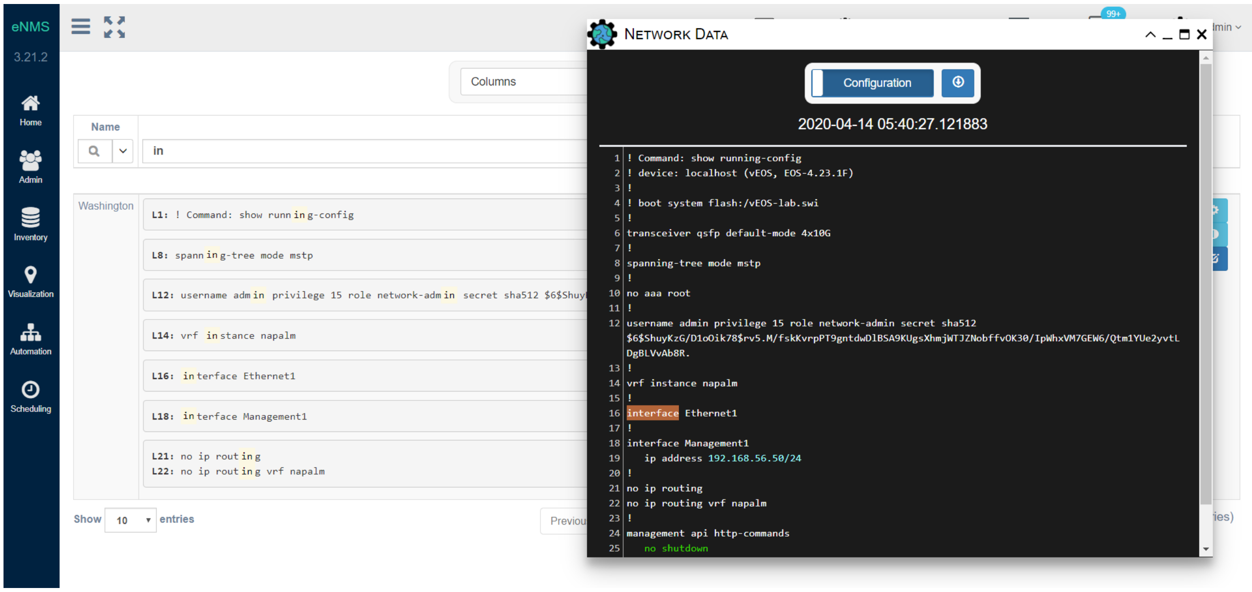
1.13.2 Push configurations to git

Configurations are written to a local text file, located in `/network_data/`, which is mapped in the `settings.json` to a git repository. Upon retrieving the current configuration from a device, the config is added to the database, as well to the local text file. Git is used for storing historical revisions of the data, and each additional instance of eNMS can retrieve the Git history using the `Admin Button -> Fetch Git Configurations Button`. Git fetch can also be configured in the cron to periodically get triggered through the CLI to update the Configurations that were pushed into Git by another instance of eNMS.

1.13.3 Search and display the configuration

From the *Inventory -> Configurations*, you can search for a specific word, a string that is included in a pattern or a regular expression in the current configuration of all devices, using the *Configuration* column. eNMS will filter the list of devices based on whether the current configuration of the device contains the search criteria. Select the “Lines of Context” slider at the top of the UI to see, up to 5 lines, before and after, the specified word that was searched.

By clicking on the `Network Data` button on the right of the screen, you can display the device Configuration

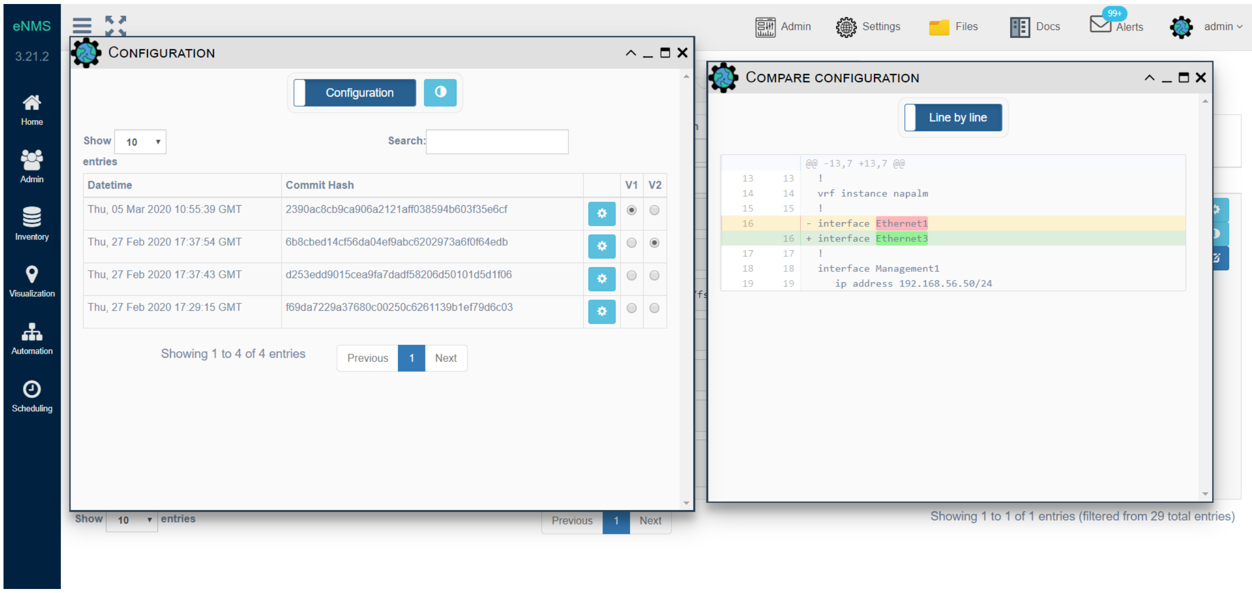


Washington

```
L1: ! Command: show running-config
L8: spanning-tree mode mstp
L12: username admin privilege 15 role network-admin secret sha512 $6$Shuy
L14: vrf instance napalm
L16: interface Ethernet1
L18: interface Management1
L21: no ip routing
L22: no ip routing vrf napalm
```

```
! Command: show running-config
! device: localhost (vEOS, EOS-4.23.1F)
!
! boot system flash:/vEOS-lab.swi
!
transceiver qsfp default-mode 4x10G
!
spanning-tree mode mstp
!
no aaa root
!
username admin privilege 15 role network-admin secret sha512
$6$ShuyKzG/D1oOik78$rv5.M/fskKvrpPT9gntdwDlBSA9KUgsXhmjWTJZNobffvOK30/IpWhxVM7GEW6/Qtm1YUe2yvtL
DgBLVvAb8R.
!
vrf instance napalm
!
interface Ethernet1
!
interface Management1
   ip address 192.168.56.50/24
!
no ip routing
no ip routing vrf napalm
!
management api http-commands
   no shutdown
```

By clicking on the `Historic` button on the right of the screen, you can view the differences between various revisions of the device configuration



Datetime	Commit Hash		V1	V2
Thu, 05 Mar 2020 10:55:39 GMT	2390ac8cb9ca906a2121aff038594b603f35e6cf	⚙	●	○
Thu, 27 Feb 2020 17:37:54 GMT	6b8cbed14cf56da04ef9abc6202973a6f0f64edb	⚙	○	●
Thu, 27 Feb 2020 17:37:43 GMT	d253edd9015cea9fa7dadf58206d50101d5d1f06	⚙	○	○
Thu, 27 Feb 2020 17:29:15 GMT	f69da7229a37680c00250c6261139b1ef79d6c03	⚙	○	○

Showing 1 to 4 of 4 entries

```
@@ -13,7 +13,7 @@
  !
  vrf instance napalm
  !
- interface Ethernet1
+ interface Ethernet3
  !
  interface Management1
     ip address 192.168.56.50/24
```

1.14 REST API

In this section, the word `instance` refers to any object type supported by eNMS. In a request, `<instance_type>` can be any of the following: `device`, `link`, `user`, `service`, `task`, `pool`.

eNMS has a REST API that can:

1.14.1 Run a service

Services can be run using a standard end point with a payload used to define service specifics.

Listing 1: **POST** Request

```
/rest/run_service
```

Service Payload Criteria

- `name` (**mandatory**) Name of the service.
- `devices` (default: `[]`) List of target devices. By default, the service will run on the devices configured from the web UI
- `pools` (default: `[]`) Same as devices for pools.
- `ip_addresses` (default: `[]`) Same as devices for pools.
- `payload` (default: `{}`) Payload of the service.

- `async` (default: `false`) JSON boolean.
  - `false` eNMS runs the service and responds to your request when the service is done running. The response will contain the result of the service, but the connection might time out if the service takes too much time to run.
  - `true` eNMS runs the service in a different thread and immediately responds with the service ID.

Listing 2: Service Payload Example

```
{
  "name": "my_service_or_workflow",
  "devices": ["Washington"],
  "pools": ["Pool1", "Pool2"],
  "ip_addresses": ["127.0.0.1"],
  "async": true,
  "payload": {"aid": "1-2-3", "user_identified_key": "user_identified_value"}
}
```

Note:

- If you do not provide a value for `devices` you will get the default devices built into the web UI, even if you provide a value in `pools` or `ip_address`.
- For Postman use the type "raw" for entering key/value pairs into the body. Body must also be formatted as application/JSON.
- Extra form data parameters passed in the body of the POST are available to that service or workflow in payload["rest_data"][your_key_name1] and payload["rest_data"][your_key_name2], and they can be accessed within a Service Instance UI form as {{payload["rest_data"][your_key_name].

Run Service Response - Synchronous

If the "async" argument is either **false** or omitted, then the request will block until the service has been run to completion or manually stopped.

This is subset of the JSON response returned for a device-by-device workflow.

Listing 3: Run Service Response Example - Synchronous

```
{
  "runtime": "2020-04-28 12:21:11.404910",
  "success": true,
  "summary": {
      "success": [
          "Device1_Name",
          "Device2_Name"
      ],
      "failure": []
  },
  "duration": "0:00:01",
  "trigger": "REST",
  "devices": {
     ...
  },
  "errors": []
}
```

Run Service Response - Asynchronous

If the "async" argument is true, then you will get JSON response with the **runtime** name needed to retrieve the results.

Listing 4: Run Service Response Example - async

```
{
    "errors": [],
    "runtime": "2020-04-28 12:16:45.201077"
}
```

1.14.2 Retrieve the status / results of a top-level service

Listing 5: GET Request

```
/rest/result/<service_name>/<runtime>
/rest/result/My%20Service/2020-04-29%2000:39:22.540921
```

- You will need to replace blank spaces ' ' in the service_name and runtime with '%20'
- The **status** property in the result will show either "Running" or "Completed"

Listing 6: Get run service result - result not ready yet (200)

```
{
    "status": "Running",
    "result": "No results yet."
}
```

The response when the result is ready will look very close to the synchronous result, above - but nested one level deeper inside the "result" property, below.

Listing 7: Get run service result - result is ready (200)

```
{
    "status": "Completed",
    "result": {
        "runtime": "2020-04-28 12:47:43.492570",
        "success": true,
        "summary": {
            "success": [
                "Device1_Name",
                "Device2_Name"
            ],
            "failure": []
        },
```

(continues on next page)

(continued from previous page)

```
        "duration": "0:00:02",
}
```

1.14.3 Retrieve or delete an instance

Retrieve all attributes for a given instance.

Listing 8: **GET** or **DELETE** Request

```
/rest/instance/<instance_type>/<instance_name>
```

1.14.4 Retrieve a list of instances with a simple query

Retrieve all instances that mach a simple query.

```
# via a GET method to the following URL
https://<IP_address>/rest/query/<instance_type>?parameter1=value1&parameter2=value2...

Example: http://enms_url/rest/query/device
Returns all devices

Example: http://enms_url/rest/query/device?port=22&operating_system=eos
Returns all devices whose port is 22 and operating system EOS
```

1.14.5 Retreive a list of instances with customized query

Custom table search that allows users to define desired columns to be returned. This search also allows user to define RegEx search to be used to find matching instances.

Custom Query Request

Listing 9: **POST** Request

```
/rest/search
```

Custom Query Payload

- `type` - Type of object to search (device, link, . . .)
- `columns` - List of attributes that will become keys in dictionary response
- `maximum_return_records` - Integer indicating the maximum number of records to return

- `search_criteria` - Dictionary requiring two key/value pairs to define a single search parameter

Listing 10: Example

```
{
  "type": "device",
    "columns": ["name", "ip_address", "configuration", "configuration_matches"],
    "maximum_return_records": 3,
    "search_criteria": {"configuration_filter": "inclusion", "configuration": "i"}
}
```

Listing 11: Example

```
{
  "type": "link",
    "columns": ["name", "source_name"],
    "maximum_return_records": 3,
    "search_criteria": {"name_filter": "inclusion", "name": "i"}
}
```

Listing 12: Retrieve all results for a service

```
{
  "type": "result",
  "columns": ["result", "service_name", "device_name", "workflow_name"],
  "search_criteria": {
    "service_name": "Regression Workflow L: superworkflow",
    "parent_runtime": "2020-05-25 11:45:25.721338"
  }
}
```

In order to retrieve a result for a specific device, it is possible to add the `device_name` key in the search criteria.

Note:

- Possible `columns` (or properties) can be found in `setup/properties.json`.
- Special `columns` "matches" is derived from a RegEX match "configuration", which returns the line where a regex was found
- The example above will search for configurations using the regex of "link-".
- Note the use of configuration attribute is used twice to define a single parameter in `search_criteria`. Additional pairs can be added to `search_criteria` to further refine the search.
- Note in the above example that the attribute used to search on is not required in `search_criteria`.
- (attribute)_filter: options include "regex", "inclusion", "exclusion".

1.14.6 Retrieve the configuration of a device

Returns the configuration for a device that has been previously retrieved from the network and stored in the application.

Listing 13: GET Request

```
/rest/configuration/<device_name>
```

1.14.7 Create or update an instance

Used to build or modify and instance in the application.

```
# via a POST or PUT method to the following URL
https://<IP_address>/rest/instance/<instance_type>
```

Example of payload to schedule a task from the REST API: this payload will create (or update if it already exists) the task `test`.

```
{
   "name": "test",
   "service": "netmiko_check_vrf_test",
   "is_active": true,
   "devices": ["Baltimore"],
   "start_date": "13/08/2019 10:16:50"
}
```

This task schedules the service `netmiko_check_vrf_test` to run at `20/06/2019 23:15:15` on the device whose name is `Baltimore`.

1.14.8 Migrate between eNMS applications

The migration system can be triggered from the REST API:

```
# Export: via a POST method to the following URL
https://<IP_address>/rest/migrate/export

# Import: via a POST method to the following URL
https://<IP_address>/rest/migrate/import
```

The body must contain the name of the project, the types of instance to import/export, and an boolean parameter called `empty_database_before_import` that tells eNMS whether or not to empty the database before importing.

Example of body:

```
{
 "name": "test_project",
 "import_export_types": ["user", "device", "link", "pool", "service", "workflow_edge",
↪ "task"],
 "empty_database_before_import": true
}
```

You can also trigger the import/export programmatically. Here's an example with the python `requests` library.

```
from requests import post
from requests.auth import HTTPBasicAuth

post(
    'yourIP/rest/migrate/import',
    json={
        "name": "Backup",
        "empty_database_before_import": False,
        "import_export_types": ["user", "device", "link", "pool", "service",
↪"workflow_edge", "task"],
    },
    headers={'content-type': 'application/json'},
    auth=HTTPBasicAuth('admin', 'admin')
)
```

Topology Import / Export

The import and export of topology can be triggered from the REST API, with a POST request to the following URL:

```
# Export: via a POST method to the following URL
https://<IP_address>/rest/topology/export

# Import: via a POST method to the following URL
https://<IP_address>/rest/topology/import
```

For the import, you need to attach the file as part of the request (of type "form-data" and not JSON). You must also set the two following `key` / `value` pairs.

replace: Whether or not the existing topology must be replaced by the newly imported objects

Example of python script to import programmatically:

```
from pathlib import Path
from requests import post
from requests.auth import HTTPBasicAuth

with open(Path.cwd() / 'project_name.xls', 'rb') as f:
    post(
        'https://IP/rest/topology/import',
        json={'replace': True},
        files={'file': f},
        auth=HTTPBasicAuth('admin', 'admin')
    )
```

For the export, you must set the name of the exported file in the JSON payload:

```
{
    "name": "rest"
}
```

1.14.9 Ping eNMS

Test that eNMS is alive.

Listing 14: GET Request

```
/rest/is_alive
```

Listing 15: Response

```
{
    "name": 153558346480170,
    "cluster_id": true,
}
```

1.14.10 Administration functionality

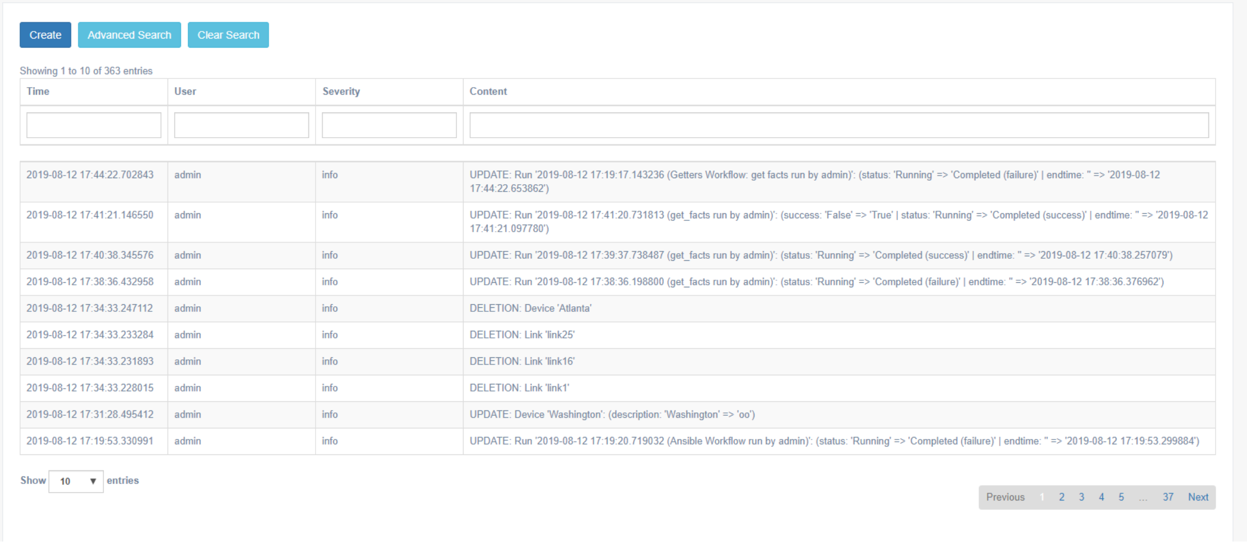
Some of the functionalities available in the administration panel can be accessed from the REST API as well:

- `update_database_configurations_from_git`: download and update device configuration from a git repository.
- `update_all_pools`: update all pools.
- `get_git_content`: fetch git configuration and automation content.

1.15 Administration

1.15.1 Changelog

eNMS changelog feature can be found under *Home / Changelog*.



Time	User	Severity	Content
2019-08-12 17:44:22.702843	admin	info	UPDATE: Run '2019-08-12 17:19:17.143236 (Getters Workflow: get facts run by admin)': (status: 'Running' => 'Completed (failure)' | endtime: '' => '2019-08-12 17:44:22.653862')
2019-08-12 17:41:21.146550	admin	info	UPDATE: Run '2019-08-12 17:41:20.731813 (get_facts run by admin)': (success: 'False' => 'True' | status: 'Running' => 'Completed (success)' | endtime: '' => '2019-08-12 17:41:21.097780')
2019-08-12 17:40:38.345576	admin	info	UPDATE: Run '2019-08-12 17:39:37.738487 (get_facts run by admin)': (status: 'Running' => 'Completed (success)' | endtime: '' => '2019-08-12 17:40:38.257079')
2019-08-12 17:38:36.432958	admin	info	UPDATE: Run '2019-08-12 17:38:36.198800 (get_facts run by admin)': (status: 'Running' => 'Completed (failure)' | endtime: '' => '2019-08-12 17:38:36.376962')
2019-08-12 17:34:33.247112	admin	info	DELETION: Device 'Atlanta'
2019-08-12 17:34:33.233284	admin	info	DELETION: Link 'link25'
2019-08-12 17:34:33.231893	admin	info	DELETION: Link 'link16'
2019-08-12 17:34:33.228015	admin	info	DELETION: Link 'link1'
2019-08-12 17:31:28.495412	admin	info	UPDATE: Device 'Washington': (description: 'Washington' => 'oo')
2019-08-12 17:19:53.330991	admin	info	UPDATE: Run '2019-08-12 17:19:20.719032 (Ansible Workflow run by admin)': (status: 'Running' => 'Completed (failure)' | endtime: '' => '2019-08-12 17:19:53.299884')

Changelog contains searchable information, search on:

- All types of creation and deletion related activity of objects supported by the application
- Modification activity, made to objects (e.g updates of values, naming etc.)
- Running of services / workflows; when they ran, who ran them
- Various administration logs, such as database migration, parameters update, etc.
- Custom logs, defined in services / workflows

1.15.2 Custom Properties

Properties of devices can be managed or extended with your own "custom" properties. Customized properties are read from the `properties.json` file in the `setup` folder, with the following variables:

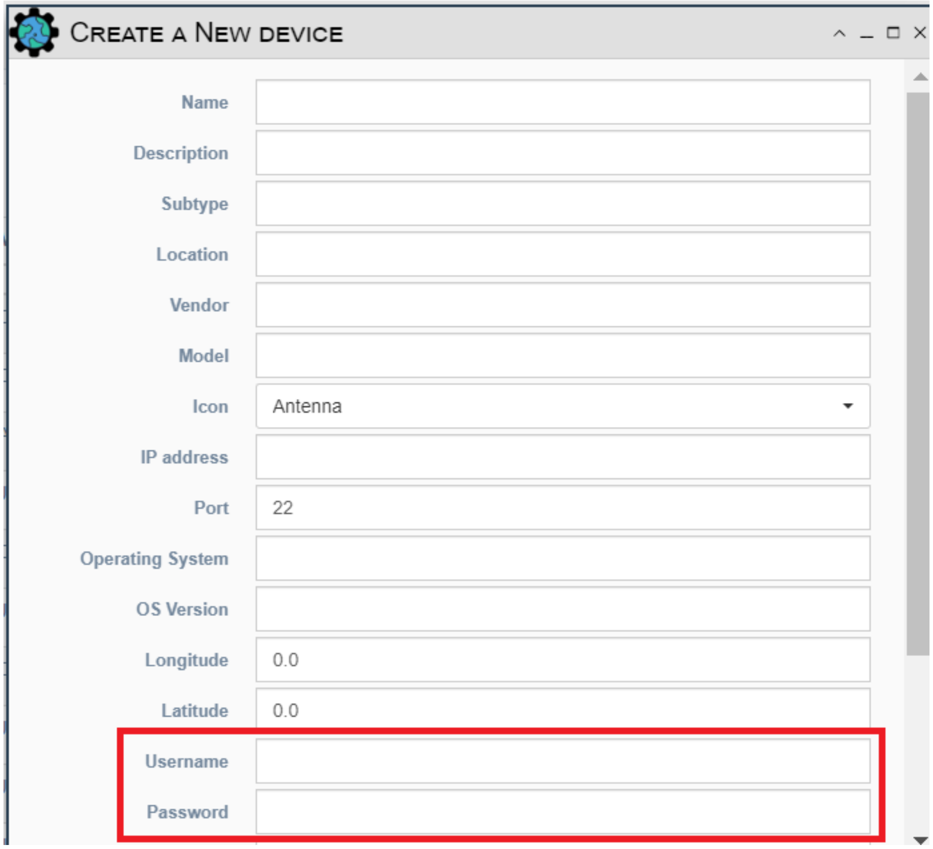
- `type` (**mandatory**): `string`, `integer`, `float`, and `boolean`
- `pretty_name` (**optional**): Custom name of the property in the UI
- `default` (**optional**): Default value
- `add_to_dashboard` (**optional**): Set to `true`, to allow the property to appear in the dashboard
- `private` (**optional**): If `true`, the value is considered sensitive: it will not be displayed in the UI. It will be encrypted in the database and stored in the Vault (if the Vault has been set up)
- `is_address` (**optional**): Set to `true` if you want to property to be usable by GoTTY to connect to network devices (e.g hostnames, IP addresses, etc)

Note: Customized properties are defined ONCE, prior to eNMS starting up for the first time, since they are mapped into the database schema. Changes to customized properties require the database to be altered or dropped and reloaded to allow the object relational mapping to recreate the schema.

1.15.3 Credentials

All Credentials are stored in a Vault or in the database if a Vault has not been configured. For a production eNMS system, a Hashicorp Vault is recommended.

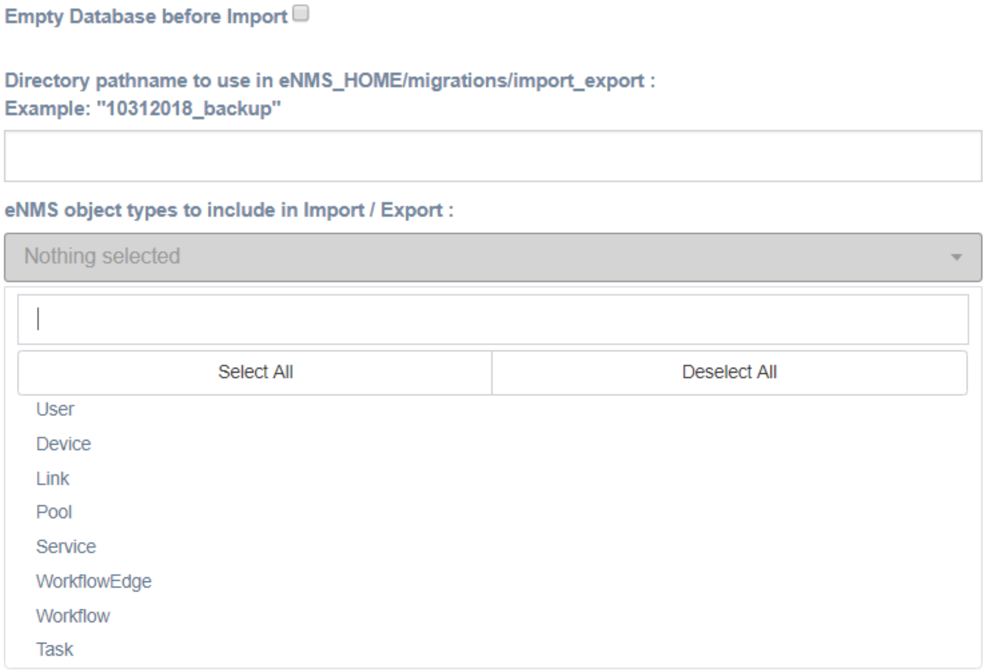
- User credentials can be used to authenticate to eNMS, as well as to authenticate to a network device
- Device credentials are properties of the device, consisting of `username`, `password` and `enable password`, (required on some devices to enter the "enable" mode)



Icon: Antenna
Port: 22
Longitude: 0.0
Latitude: 0.0

1.15.4 Database: Migration, Backup and Restore

The eNMS migration system handles exporting the complete database content into YAML files. By providing a directory name and selecting which eNMS object types to export/backup, eNMS serializes the stored objects in the directory `eNMS/files/migrations/directory_name`. These yaml files can then be copied into the same directory on a new VM instance of eNMS, and then the Import function can be used to import/restore the configuration and living data of those object types. These migration files are used for migrating from one version of eNMS to the next version. They are also used for Backup and Restore of eNMS. The migration system is accessed from the *Admin* icon at the top of the UI or from the REST API.


Empty Database before Import ☐
Directory pathname to use in eNMS_HOME/migrations/import_export :
Example: "10312018_backup"
eNMS object types to include in Import / Export :
Nothing selected
Select All
Deselect All
User
Device
Link
Pool
Service
WorkflowEdge
Workflow
Task

- `Migration Import/Export` Restore/Import database on a newly created instance of eNMS:
- Launch the GUI and login into a freshly built/installed eNMS system
- Ensure that the migration YAML files are present in the desired folder
- At the top of the UI screen, Click the *Admin* button
- Click `Migration Import/Export` and select options, all the object types from drop down menu and the directory of where the migration YAML files are kept and select Import
- Alternatively, from the REST API, Select 'Empty_database_before_import' = True, specify the location of the directory to import from, select all object types to be imported: "user", "device", "link", "pool", "service", "workflow_edge", "task"
- `Migration Import/Export` Backing/Exporting database on an existing eNMS:
- Launch the GUI and login into an existing eNMS system
- At the top of the UI screen, Click the *Admin* button
- Click `Migration Import/Export` and select all the object types from the drop down menu and select Export

Note: Exported backup files do not contain the secure credentials for each of the inventoried devices in plain text. The credentials are considered to be stored in a Vault in production mode.

Note: If you are migrating data onto an existing instance (as opposed to a fresh instance) of eNMS, you can select the option `Empty Database before Import` to empty the database before starting the migration.

Note: See additional discussion of migration in the Installation Section

- `Per-type Mass Deletion`: Select object type(s) to be deleted
- `Import Service`: Services and workflow can be exported and imported individually (as a .tgz archive); There may be a need to send a service / workflow from one VM to another. First, move the archive to the *files/services* folder, Click the *Admin* button, click on the `Import services` button and select the service from the drop down menu
- `Delete Results/Logs`: Will allow historical results, logs, and changelog to be deleted
- `Delete Corrupted Edges`: Scans workflows to find duplicate edges between services or reference to an edge between services where a service does not exist and then deletes them

1.15.5 Miscellaneous

- `Fetch Git Configurations`: Will retrieve configurations from the git 'configurations' repository and load those into the database for each matching inventory device. This is performed automatically when eNMS starts up: the git configurations repository is quietly cloned and loaded into the database. This feature allows manual pulling of updated configurations data
- `Scan Cluster Subnet`: Will populate the `Admin -> Servers` table with related VMs where eNMS has been deployed as a cluster of Servers

1.15.6 Inventory

eNMS inventory, devices and links, can be exported and Imported into an Excel based format. When executing an exporting function, the file will be exported to a folder, local to the VM. When executing an importing function, the application will request for the desired file, local to the workstation/laptop. file,

- `Excel Import`: Will import the excel file from your local workstation or laptop
- `Excel Export`: Will export the excel file onto your VM, in the directory `eNMS/files/spreadsheets`

1.15.7 Local Server CLI interface

The local VM terminal can be used as a CLI interface, that interacts with the eNMS application. The prerequisite is to ensure that you are in the correct application directory and to deactivate any specific proxy settings. The proxy settings are company specified and will prevent commands from running. The user can now "ssh" into the VM and perform the following operations:

Run a service

If a service has been created on the application, the user can run a service via this CLI Interface.

General syntax: `flask run_service <service_name> --devices <list_of_devices> --payload '{json dict}'`

Options: –devices = List of comma separated device names (Optional) –payload = JSON dictionary of key/values, serving as starting data for the service to be used later (Optional)

Examples:

```
`flask run_service get_facts`
`flask run_service get_facts --devices Washington,Denver`
`flask run_service get_facts --payload '{"a": "b"}'`
`flask run_service get_facts --devices Washington,Denver --payload '{"a": "b"}'`
```

Delete old log entries

This command will purge logs changelog or result. By default, logs older than 15 days will be removed from their respective tables

General syntax: `flask delete_log --keep-last-days <value> --log <value>`

Options: –keep-last-days = Number of days to keep the logs (Optional: default to 15) –log = The log information to remove the logs from, either “changelog” or “result” (Required)

Examples:

```
`flask delete_log --keep-last-days 10 --log result`    // will retain the last 10␣
↪days of result
`flask delete_log --log changelog`                     // will retain the last 15␣
↪days of changelogs
```

Refresh Network Configuration Data

The Network Configuration data can be gathered and then stored in a central location, namely the git repository. eNMS can be used to fetch the Network Configuration from git and have it stored locally in `/network_data/`

General syntax: `flask pull_git`

Options: None

1.16 Contributing

1.16.1 For developers

eNMS uses :

- Black for python code formatting
- Flake8 to make sure that the python code is PEP8-compliant
- Mypy for python static type hinting
- Prettier for javascript code formatting
- Eslint to make sure the javascript code is compliant with google standards for javascript
- Pytest for the test suite.

There is a dedicated `requirements_dev.txt` file to install these python libraries:

```
pip install -r requirements_dev.txt
```

Before opening a pull request with your changes, you should make sure that:

```
# your code is black compliant
# Black is a code formatting enforcement tool; see (https://github.com/ambv/black)
black --check --verbose .

# your code is PEP8 (flake8) compliant (python)
flake8 --config tests/linting/.flake8 .

# your code is mypy compliant
```

(continues on next page)

(continued from previous page)

```
# Mypy is a adds static type hinting to Python for the whole project; see (http://
↪mypy-lang.org/)
mypy --config-file tests/linting/mypy.ini .

# your code is prettier compliant (javascript)
npm run prettier

# your code is eslint compliant (javascript)
npm run lint

# all the tests are passing
pytest
```

If one of these checks fails, so will Travis CI after opening the pull request.

The CI/CD and PR processes are the same, because when you open a PR, this automatically triggers Travis.

If you are updating the documentation, you can build a local version of the docs:

```
# build a local version of the docs
cd /docs
make html
```