JUNIPER
NETWORKS

Engineering
Simplicity

REST API Orchestration Guide

Published
2023-07-07

RELEASE
4.2

Table of Contents

Introduction

In recent years, REST (REpresentational State Transfer) has emerged as the standard architectural design for web services and web APIs.

This documentation describes how to integrate Paragon Active Assurance with a network service orchestrator via the Paragon Active Assurance REST API. This is a RESTful API that adheres to the standardized OpenAPI/Swagger 2.0 definition format, thereby enabling easy and efficient orchestration of Paragon Active Assurance tasks.

(As explained here: "OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API, including: available endpoints (/users) and operations on each endpoint (GET /users, POST /users); operation parameters Input and output for each operation; authentication methods; and contact information, license, terms of use and other information.")

Hands-on examples are given of the principal tasks involved, including:

- creating and deploying Virtual Test Agents

- running tests and monitors

- retrieving results from these activities.

Orchestration through the REST API (that is, using the "write" functions of the API) is available only with on-premise installations of Paragon Active Assurance where the REST API has been installed. Orchestration is not available to users of the Paragon Active Assurance SaaS solution. However, SaaS users do have access to the data retrieval (that is, "read") functions of the REST API.

Conventions

The following abbreviations are used in this document:

Abbreviation	Meaning
ES	Errored Second
MEP	MEG (Maintenance Entity Group) End Point *(ITU-T Y.1731 definition)*
	or Maintenance End Point *(Cisco definition)*
NFV	Network Function Virtualization
NFVO	Network Function Virtualization Orchestrator
REST	REpresentational State Transfer
SLA	Service Level Agreement
VNF	Virtual Network Function
vTA	Virtual Test Agent

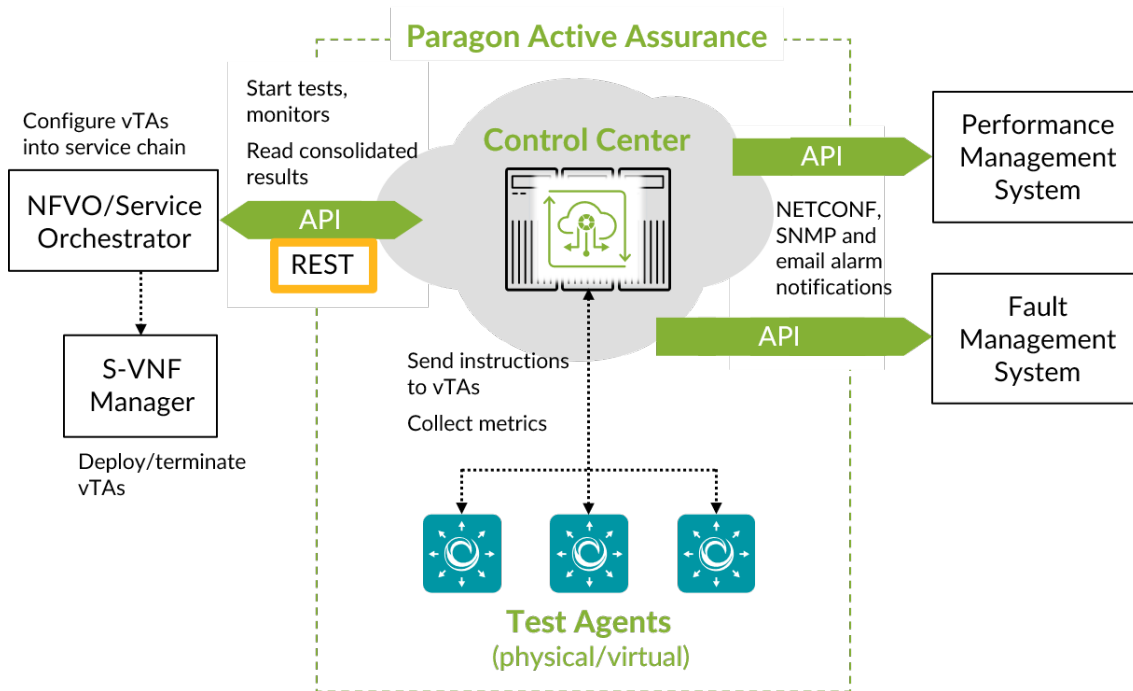# Orchestration with Paragon Active Assurance

IN THIS SECTION

Overview

A third-party NFVO or service orchestrator is typically the component which initiates test and monitoring sessions using the Control Center API. This orchestrator also retrieves the aggregated measurement results from the Test Agent activities. Performance KPIs may be retrieved by third-party

Performance Management Systems, while events – once triggered by threshold violations set in the Control Center – can be sent to third-party Fault Management systems.

To summarize, the figure below shows how Paragon Active Assurance interacts with other third-party systems in the OSS landscape.

- **NFVO/Service Orchestrator:** Instructs the VNF Manager to deploy the vTAs and configure Paragon Active Assurance into the service chain. Once the service has been activated, the orchestrator uses the API towards Control Center to trigger service activation tests and retrieve pass/fail results. If the tests pass, the orchestrator will use the API towards Control Center to start active monitoring of the service. KPIs from the monitoring are retrieved continuously either by the orchestrator or by a separate Performance Management platform.

- **Control Center (G-VNF Manager):** Deploys, scales, and terminates the vTA as instructed by the NFVO or service orchestrator.

- **Performance Management system or Service Quality Management system:** Reads KPIs from active monitoring via the Control Center API.

- **Fault Management system:** Receives NETCONF, SNMP, or email notifications from Control Center if SLAs are violated.

Definitions of Paragon Active Assurance Concepts

- **Test Agents:** The components that perform measurements (for tests as well as monitors) in a Paragon Active Assurance system. Test Agents consist of software with the ability to generate, receive, and analyze real network traffic.

    - The kind of Test Agent discussed in this document is the *Virtual Test Agent (vTA)*, a virtual network function (VNF) deployed on a hypervisor. Other types of Test Agent also exist.

- *Measurement types:* There are two basic types of measurement in Paragon Active Assurance, *tests* and *monitors*.

    - **Test:** A test consists of one or several *steps*, each of which has a *specified*, *finite duration*. Steps are executed sequentially. Each step may entail running multiple *tasks* concurrently.

    - **Monitor:** A monitor does not have a specified duration but executes *indefinitely*. Like a step in a test, a monitor may execute multiple concurrent tasks.

- **Template:** When Paragon Active Assurance is controlled by an orchestrator, tests and monitors are always executed by means of templates in which the test or monitor is defined. Parameter settings can be passed as inputs to the template at runtime.

Work Flow for Automation

Design Time

At design time, you prepare measurements by creating templates for tests and monitors in Paragon Active Assurance. How to do that is covered in the chapter "Test and Monitor Templates" on page 28.

Runtime

At runtime, you set up your devices and perform the actual measurements.

- An overview of all examples given is found in the chapter "Examples of Controlling Paragon Active Assurance via REST API" on page 29.

- How to deploy and configure Test Agents is gone through in the chapter "Examples: Test Agents" on page 30.

- How to configure inventory items is gone through in the chapter "Examples: Inventory Items" on page 48:

    - TWAMP reflectors

- Y.1731 MEPs

- IPTV channels

- SIP accounts

- How to configure alarms is covered in the chapter "Examples: Alarms" on page 54.

- How to run tests and monitors by executing Paragon Active Assurance templates through the REST API is described in the chapters "Examples: Tests" on page 75 and "Examples: Monitors" on page 62.

There is also the possibility of pushing SSH keys to Test Agents to enable logging in to the Test Agents via SSH. The details are covered in the chapter "Examples: SSH Keys" on page 60.

Supported Paragon Active Assurance Features

IN THIS SECTION

- Product Features Available in the REST API | 5

All test and monitor types in Paragon Active Assurance can be created and executed from the REST API through the use of templates. How to set up test and monitor templates is explained in the in-app help under "Tests and monitors" > "Creating templates".

Creation of Paragon Active Assurance accounts is currently not supported; however, one or several predefined accounts will have been set up for the user.

The tables that follow detail what product features are available in this release.

For full details on syntax, refer to `https://<Control Center host IP>/rest`.

Product Features Available in the REST API

Each REST API operation consists of an HTTP method (for example, GET, PUT, PATCH, POST, or DELETE) applied to a URL with a specified syntax. Each operation is therefore represented below as follows:

HTTP method	Path

where the omitted initial part of the URL is `{protocol}://{Control Center host IP}/rest`.

All operations are performed on the Paragon Active Assurance account given in the URL.

Pagination

Many of the API resources in the REST API are used to list multiple items. These resources all share the same mechanism for paginating the returned results by specifying the query parameters:

- `limit`: Setting this to n means that at most n elements will be returned. By default, this limit is set to 10 elements.

- `offset`: Setting this to n means that elements will be returned starting at position n in the list. By default, there is no offset.

The result is returned in the following format:

```
{
    "items":
    {
      ...
    },
    "offset": 0,
    "limit": 10,
    "count": 123
}
```

Here, `items` contains the returned result items, while `count` denotes the total number of items available in the list.

The resources are listed in the subsections that follow.

Resource: Alarms

Action	Path
GET	/accounts/{account}/alarms/

Retrieve a list of all alarms with their full definitions. For each alarm, an active config is returned; this is what is used to actually trigger an alarm. It consists of parameters from an alarm template overridden by values from an alarm config.

Action	Path
PUT	/accounts/{account}/alarms/{alarm_id}/

Suppress an alarm with a given ID so that it will never trigger.

Action	Path
GET	/accounts/{account}/alarms/{alarm_id}/

Retrieve the full definition of an alarm with a given ID. An active config is returned for the alarm; this is what is used to actually trigger it. The config consists of parameters from an alarm template overridden by values from an alarm config.

Action	Path
DELETE	/accounts/{account}/alarms/{alarm_id}/

Delete an alarm with a given ID.

Resource: Alarm Email Lists

Action	Path
PUT	/accounts/{account}/alarm_emails/{alarm_emails_id}/

Modify the configuration of an alarm email list with a given ID.

Action	Path
GET	/accounts/{account}/alarm_emails/{alarm_emails_id}/

Retrieve an alarm email list with a given ID.

Action	Path
DELETE	/accounts/{account}/alarm_emails/{alarm_emails_id}/

Delete an alarm email list with a given ID.

Action	Path
POST	/accounts/{account}/alarm_emails/

Create a new alarm email list.

Action	Path
GET	/accounts/{account}/alarm_emails/

Retrieve a list of all alarm email lists.

Resource: Alarm Templates

Action	Path
PUT	/accounts/{account}/alarm_templates/{alarm_template_id}/

Modify the configuration of an alarm template with a given ID.

Action	Path
GET	/accounts/{account}/alarm_templates/{alarm_template_id}/

Retrieve the full definition of an alarm template with a given ID.

Action	Path
DELETE	/accounts/{account}/alarm_templates/{alarm_template_id}/

Delete an alarm template with a given ID.

Action	Path
POST	/accounts/{account}/alarm_templates/

Create a new alarm template.

Action	Path
GET	/accounts/{account}/alarm_templates/

Retrieve a list of all alarm templates.

Resource: External URL Share

External URL shares consist of test or monitor results shared externally by means of a URL.

Action	Path
PUT	/accounts/{account}/external_url_shares/{share_id}/

Modify the configuration of an external URL share with a given ID.

Action	Path
GET	/accounts/{account}/external_url_shares/{share_id}/

Retrieve the configuration of an external URL share with a given ID.

Action	Path
DELETE	/accounts/{account}/external_url_shares/{share_id}/

Delete an external URL share with a given ID.

Action	Path
POST	/accounts/{account}/external_url_shares/

Create a new external URL share.

Action	Path
GET	/accounts/{account}/external_url_shares/

Retrieve a list of all external URL shares.

Resource: IPTV Channels

Action	Path
PUT	/accounts/{account}/iptv_channels/{iptv_id}/

Modify the configuration of an IPTV channel with a given ID.

Action	Path
GET	/accounts/{account}/iptv_channels/{iptv_id}/

Retrieve the configuration of an IPTV channel with a given ID.

Action	Path
DELETE	/accounts/{account}/iptv_channels/{iptv_id}/

Delete an IPTV channel with a given ID.

Action	Path
POST	/accounts/{account}/iptv_channels/

Create a new IPTV channel.

Action	Path
GET	/accounts/{account}/iptv_channels/

Retrieve a list of all IPTV channels.

Resource: Monitors

Action	Path
GET	/accounts/{account}/monitors/{monitor_id}/pdf_report

Generate a PDF report on a monitor with a given ID.

Action	Path
POST	/accounts/{account}/monitors/

Create a new monitor based on a monitor template.

Action	Path
GET	/accounts/{account}/monitors/

Retrieve a list of all defined monitors as well as data on SLA fulfillment during their execution. SLA fulfillment is by default indicated for the last 15 minutes; a different time period can be specified with the query string parameters start and end.

Action	Path
PUT	/accounts/{account}/monitors/{monitor_id}/

Start or stop a monitor with a given ID. This is done by setting the `started` parameter to True or False. This operation can also be used to change the monitor name or description, or to modify any parameter in its config.

Action	Path
PATCH	`/accounts/{account}/monitors/{monitor_id}/`

Start or stop a monitor with a given ID. This is done by setting the `started` parameter to True or False. PATCH can also be used to change the monitor name or description.

> **NOTE**: PATCH cannot be used to edit other monitor parameters; PUT needs to be used instead.

Action	Path
GET	`/accounts/{account}/monitors/{monitor_id}/`

Retrieve the full definition of a monitor with a given ID, as well as data on SLA fulfillment during its execution and comprehensive data metrics for successive time intervals (the length of these intervals is governed by the `resolution` parameter). SLA fulfillment is by default indicated for the last 15 minutes; a different time period can be specified with the query string parameters `start` and `end`. – Periodic tests are run as part of a monitor. If a task has a `task_type` value of `periodic`, the `results` list will always be empty. Instead, an `executions` list is provided containing a list of test IDs for each run of the tests. Here the actual results are found.

Action	Path
DELETE	`/accounts/{account}/monitors/{monitor_id}/`

Delete a monitor with a given ID.

Resource: Monitor Templates

Action	Path
GET	`/accounts/{account}/monitor_templates/`

Retrieve a list of all monitor templates.

Action	Path
POST	/accounts/{account}/monitor_templates/import/

Import previously exported monitor templates.

Action	Path
GET	/accounts/{account}/monitor_templates/{template_id}/

Retrieve the full definition of a monitor template with a given ID.

Action	Path
GET	/accounts/{account}/monitor_templates/export/

Export all monitor templates.

Creation of monitor templates must be done through the Control Center web GUI; for details of this procedure, see the in-app help under "Tests and monitors" > "Creating templates".

Resource: SIP Accounts

Action	Path
POST	/accounts/{account}/sip_accounts/

Create a new SIP account.

Action	Path
GET	/accounts/{account}/sip_accounts/

Retrieve a list of all SIP accounts.

Action	Path
PUT	/accounts/{account}/sip_accounts/{sip_id}/

Modify the configuration of a SIP account with a given ID.

Action	Path
GET	/accounts/{account}/sip_accounts/{sip_id}/

Retrieve the configuration of a SIP account with a given ID.

Action	Path
DELETE	/accounts/{account}/sip_accounts/{sip_id}/

Delete a SIP account with a given ID.

Resource: SNMP Managers

Action	Path
PUT	/accounts/{account}/snmp_managers/{snmp_manager_id}/

Modify the configuration of an SNMP manager with a given ID.

Action	Path
GET	/accounts/{account}/snmp_managers/{snmp_manager_id}/

Retrieve an SNMP manager with a given ID.

Action	Path
DELETE	/accounts/{account}/snmp_managers/{snmp_manager_id}/

Delete an SNMP manager with a given ID.

Action	Path
POST	`/accounts/{account}/snmp_managers/`

Create a new SNMP manager.

Action	Path
PUT	`/accounts/{account}/snmp_managers/`

Retrieve a list of all SNMP managers.

Resource: Speedtest

Action	Path
GET	`/accounts/{account}/speedtest/results/`

Retrieve a list of all Speedtest results.

Action	Path
GET	`/accounts/{account}/speedtest/results/{speedtest_id}`

Returns a Speedtest result for a given instance ID.

> **NOTE**: The Speedtest commands are not intended to be used directly, but rather by a custom web page set up for Speedtest. See the document *Creating a Custom Speedtest Web Page*.

Resource: SSH Keys

Action	Path
PUT	/accounts/{account}/ssh_keys/{ssh_key_id}/

Modify the configuration of an SSH key with a given ID.

Action	Path
GET	/accounts/{account}/ssh_keys/{ssh_key_id}/

Retrieve an SSH key with a given ID.

Action	Path
DELETE	/accounts/{account}/ssh_keys/{ssh_key_id}/

Delete an SSH key with a given ID.

Action	Path
POST	/accounts/{account}/ssh_keys/

Add a new SSH key.

Action	Path
GET	/accounts/{account}/ssh_keys/

Retrieve a list of all SSH keys.

Resource: Tags

Action	Path
POST	/accounts/{account}/tags/

Create a new tag.

Action	Path
GET	/accounts/{account}/tags/

Retrieve a list of all tags.

Action	Path
PUT	/accounts/{account}/tags/{tag_id}/

Modify a tag with a given ID.

Action	Path
GET	/accounts/{account}/tags/{tag_id}/

Retrieve the tag and all items (monitors, templates, Test Agents, TWAMP reflectors) it has been applied to.

Action	Path
DELETE	/accounts/{account}/tags/{tag_id}/

Delete a tag with a given ID.

Action	Path
POST	/accounts/{account}/tags/assign/

Assign a tag to specified resources.

Action	Path
POST	/accounts/{account}/tags/unassign/

Unassign a tag from specified resources.

Resource: Test Agents

Action	Path
POST	/accounts/{account}/test_agents/reboot/

Trigger a reboot on target Test Agents. The target Test Agents can be provided by either specifying their IDs in a list or using the "?all" query parameter.

Action	Path
POST	/accounts/{account}/test_agents/

Create a new virtual Test Agent.

Action	Path
GET	/accounts/{account}/test_agents/

Retrieve a list of all Test Agents.

Action	Path
POST	/accounts/{account}/test_agents/update/

Reboot and update Test Agents. This action can be run in two ways: for all Test Agents, by appending the ?all query string in the URL; or for specified Test Agents, listed by ID in the body as described in the schema.

Action	Path
PUT	/accounts/{account}/test_agents/{test_agent_id}/

Modify the configuration of a Test Agent with a given ID. With the PUT method, you need to supply the configuration in its entirety.

Action	Path
PATCH	/accounts/{account}/test_agents/{test_agent_id}/

You can modify the name, description, or GPS coordinates of a Test Agent with a given ID. Note: PATCH cannot be used to edit other Test Agent properties; for this purpose PUT needs to be used instead.

Action	Path
GET	/accounts/{account}/test_agents/{test_agent_id}/

Retrieve the configuration of a Test Agent with a given ID.

Action	Path
DELETE	/accounts/{account}/test_agents/{test_agent_id}/

Delete a Test Agent with a given ID.

Action	Path
PUT	/accounts/{account}/test_agents/{test_agent_id}/move/

Move a Test Agent with a given ID to a different Control Center.

Action	Path
DELETE	/accounts/{account}/test_agents/{test_agent_id}/move/

Cancel the move of a Test Agent with a given ID to a different Control Center.

Action	Path
PUT	/accounts/{account}/test_agents/{test_agent_numeric_id}/wifiscan/ {test_agent_interface_name}/

Start a Wi-Fi scan on a named interface of a Test Agent with a given ID.

Action	Path
GET	/accounts/{account}/test_agents/{test_agent_numeric_id}/wifiscan/ {test_agent_interface_name}/

Get results from a Wi-Fi scan on a named interface of a Test Agent with a given ID.

Action	Path
DELETE	/accounts/{account}/test_agents/{test_agent_numeric_id}/wifiscan/ {test_agent_interface_name}/

Stop the Wi-Fi scan on a named interface of a Test Agent with a given ID.

Resource: Tests

Action	Path
GET	/accounts/{account}/tests/{test_id}/

Retrieve results for a test with a given ID.

Action	Path
DELETE	/accounts/{account}/tests/{test_id}/

Delete a test with a given ID.

Action	Path
GET	/accounts/{account}/tests/{test_id}/pdf_report

Generate a PDF report on a test with a given ID.

Action	Path
POST	/accounts/{account}/tests/

Create a test based on a test template, and run the test.

Action	Path
GET	/accounts/{account}/tests/

Retrieve a list of all defined tests.

Resource: Test Templates

Action	Path
GET	/accounts/{account}/test_templates/

Retrieve a list of all test templates.

Action	Path
GET	/accounts/{account}/test_templates/{template_id}/

Retrieve the full definition of a test template with a given ID.

Action	Path
POST	/accounts/{account}/test_templates/import/

Import previously exported test templates.

Action	Path
GET	/accounts/{account}/test_templates/export/

Export all test templates.

Creation of test templates must be done through the Control Center web GUI; for details of this procedure, see the in-app help under "Tests and monitors" > "Creating templates".

Resource: TWAMP Reflectors

Action	Path
PUT	/accounts/{account}/twamp_reflectors/{twamp_id}/

Modify the configuration of a TWAMP reflector with a given ID.

Action	Path
PUT	/accounts/{account}/twamp_reflectors/{twamp_id}/

Retrieve the configuration of a TWAMP reflector with a given ID.

Action	Path
DELETE	/accounts/{account}/twamp_reflectors/{twamp_id}/

Delete a TWAMP reflector with a given ID.

Action	Path
POST	/accounts/{account}/twamp_reflectors/

Create a new TWAMP reflector.

Action	Path
GET	/accounts/{account}/twamp_reflectors/

Retrieve a list of all TWAMP reflectors.

Resource: Y.1731 MEPs

Action	Path
PUT	/accounts/{account}/y1731_meps/{mep_id}/

Modify the configuration of a Y.1731 MEP with a given ID.

Action	Path
GET	/accounts/{account}/y1731_meps/{mep_id}/

Retrieve the configuration of a Y.1731 MEP with a given ID.

Action	Path
DELETE	/accounts/{account}/y1731_meps/{mep_id}/

Delete a Y.1731 MEP with a given ID.

Action	Path
POST	/accounts/{account}/y1731_meps/

Create a new Y.1731 MEP.

Action	Path
GET	/accounts/{account}/y1731_meps/

Retrieve a list of all Y.1731 MEPs.

Getting Started with the REST API

Installing and Configuring the REST API

How to download the REST API, how to install it on the Control Center server, and how to configure it for use is described in the Paragon Active Assurance Installation Guide. Please refer to this document.

> **NOTE**: It is vital to set the `SITE_URL` parameter correctly in `/etc/netrounds/netrounds.conf`. This is to provide a public URL at which the REST API can be accessed.

Configuring the Lifetime of REST Authorization Tokens

The lifetime of REST API authorization tokens is limited and is 10 years by default. This is governed by the parameter `REST_TOKEN_LIFETIME` in the file `/etc/netrounds/netrounds.conf`.

When you intend to use the REST API, you need to change the value of this parameter to whatever is required.

Obtaining an Authorization Token

Before you can start using the REST API, you need to create an authorization token for it. This can be done either in the Control Center web GUI or from the command line using the `ncc` command. See below.

You will need to include this token in all code calling the REST API; see for an example.

Obtaining an Authorization Token in the Web GUI

- In the webapp, click the Paragon Active Assurance user name in the top right corner, and select **Edit profile** from the pop-up menu.

- Select the **API tokens** tab.

- Click the button **Add API Token**.

- A new token is generated and added to the list (multiple tokens can be defined). Name it as desired in the **Name** column.

NOTE: The value of the API token is visible only at creation time. Therefore you must note it down to be able to use the token later on.

Creating and Managing Authorization Tokens from the Command Line

You can create an API token with the following command:

```
ncc rest-create-token --email <user email> [--accounts <list of accounts>] [--full-access] [--name <name of token>] [--force]
```

Note that only account Admins and Superusers are allowed to create API tokens.

Either `--accounts <list of accounts>` or `--full-access` must be provided.

The `--accounts` option specifies what accounts the token should have access to. The token will only have access to the specified accounts.

If you instead specify `--full-access`, then:

- if your user is an account Admin, the token will work for all accounts you have access to and have the Admin permission in;

- if your user is Superuser, the token will have access to the entire system. This means all accounts, even if the user is not an explicit member of them all.

The `--full-access` option gives the token access to *all* accounts. You are prompted to confirm this. Use the `--force` option to skip confirmation.

The `--name` option lets you name the token. This is optional.

You can list all API tokens with

```
ncc rest-list-tokens
```

You can delete an API token with

```
ncc rest-delete-token [--id <token id>] [--name <token name>]
```

where the `id` is obtained by listing the tokens.

Note that the API tokens created in the GUI are always tied to a single account by default. You must use the `ncc` command line tool to create tokens with more extensive access.

Accessing the REST API Browser

Once you have installed and configured the REST API, you can access an API browser for the REST API at `SITE_URL/rest/` (where `SITE_URL` is defined in `/etc/netrounds/netrounds.conf`). The API browser is interactive and enables you to try out the REST API operations manually. It also provides documentation of the REST API features. Furthermore, it links to a JSON schema which exhaustively describes the REST API (`/rest/apispec_1.json`). See the image below. The REST API browser itself was generated from this schema, and it is possible to generate client code based on it.

Before you can call the REST API from the API browser, you need to authorize such actions by providing the authorization token you have obtained (as explained in the section "Obtaining an Authorization Token" on page 25).

- Click the **Authorize** button.

- In the dialog that opens, enter your authorization token under **Value** and click the **Authorize** button in the dialog. Then close the dialog.

You can now perform REST API operations repeatedly from this web page. However, if you reload the page in your browser, you will need to repeat the above authorization procedure.

In practice, calls to the REST API will usually be scripted from an orchestrator. In this document, the orchestrator is mostly represented by Python code calling the REST API. However, it should be emphasized that any applicable language can be used. The authorization token needs to be included in any code making calls to the REST API.

Test and Monitor Templates

Templates for test and monitor types need to be set up manually through the Paragon Active Assurance front-end user interface. How to do this is covered in the in-app help under "Tests and monitors" > "Creating templates".

Exporting Test and Monitor Templates

The code below shows how to export a monitor template in JSON format. Export of tests is similar (just replace `monitor_templates` with `test_templates` in the definition of `url`).

```
# Request settings
url = '%s/accounts/%s/monitor_templates/export/' % (args.ncc_url, args.account)

# Export templates
print url
response = requests.get(url=url, headers={'API-Token': args.token})
```

Previously exported monitor templates can be imported back into Control Center as follows (again, test templates are handled analogously):

```
# Request settings
url = '%s/accounts/%s/monitor_templates/import/' % (args.ncc_url, args.account)

json_data = json.dumps({
"templates": [
{
"input": [
```

```
{
"require_bridge": False,
```

Insert the rest of the JSON string here.

```
"description": "Monitor for HTTP",
"name": "mtpl-2"
}
],
"version": "2.29",
"export_date": "2019-05-16T10:03:37.042474Z"
})


# Import monitor templates
response = requests.post(url=url, data=json_data, headers={
'API-Token': args.token,
'Accept': 'application/json; indent=4',
'Content-Type': 'application/json',
})
```

Examples of Controlling Paragon Active Assurance via REST API

IN THIS SECTION

●

In the following it is assumed that suitable test and monitor templates have been defined according to the chapter "Test and Monitor Templates" on page 28.

The examples given mostly consist of Python code calling the REST API.

Overview of Key Tasks Performed

(Some further tasks are also exemplified in what follows.)

Examples: Test Agents

IN THIS SECTION

Creating and Deploying a New Test Agent

To be able to perform measurements in Paragon Active Assurance, you first need to create one or more Test Agents in your Paragon Active Assurance account. This section tells how to create virtual Test Agents using the REST API.

We proceed in the following steps, which are detailed in the following:

1. At the outset, the account "demo" has no Test Agents in its inventory.

2. A Test Agent called "vta1" is created through the REST API. At this stage, no real Test Agent exists yet (that is, it has not yet been started).

3. The Test Agent is deployed in OpenStack. (Deployment on that platform is chosen here as one possibility among others.)

4. The Test Agent connects to the Control Center account "demo" and is now ready for use.

Step 1: At the outset, there are no Test Agents in the account "demo". See the screenshot below from the Control Center web GUI.
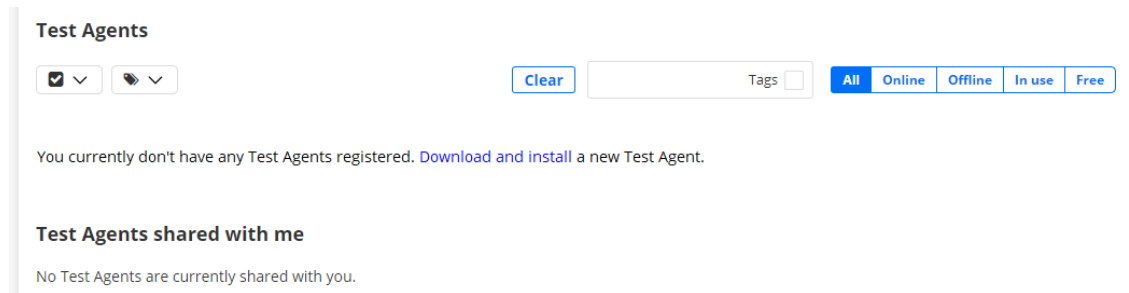
Step 2: A Test Agent is created in Control Center using the REST API POST operation for Test Agents.

We will do this in Python below. To write the Python code, it is helpful to inspect the full range of configuration options for a Test Agent in the REST API.

- Go to `https://<Control Center host IP>/rest`.

- Under **test_agent**, expand the POST operation for `/accounts/{account}/test_agents/`.

- Under **Description**, click **Model**.

Below, we create a Test Agent having three physical interfaces "eth0", "eth1", and "eth2", where eth0 has a DHCP address. An NTP server is configured on eth0, and eth0 is also the management interface (that is, the interface that connects to Control Center).

(The `import` commands are omitted from subsequent examples unless they differ from the code below. The `parser` commands are also omitted, since they are always the same. Regarding the REST API token and how to obtain it, read more in the section .)

test_agent ⌄

| GET | /accounts/{account}/test_agents/ | List Test Agents. |

| POST | /accounts/{account}/test_agents/ | Create new Test Agent. |

Create a new virtual Test Agent.

```
import argparse
import json
import requests

parser = argparse.ArgumentParser(description='Example for')
parser.add_argument('--ncc-url', help='The URL where NCC is found, for example https://
<hostname>/rest', required=True)
parser.add_argument('--token', help='The REST API token', required=True)
parser.add_argument('--account', help='The Netrounds account', required=True)

args = parser.parse_args()

# Request settings
url = '%s/accounts/%s/test_agents/' % (args.ncc_url, args.account)

# JSON content
json_data = json.dumps({
    'description': 'Test agent description',
    'interface_config': {
        'eth0': {
            'address': {
                'type': 'dhcp_ip4',
            },
            'address6': {
                'type': 'dhcp_ip6',
            },
            'description': 'eth0',
            'mtu': 1500,
            'speed': 'AUTO',
            'type': 'physical'
```

```
            },
            'eth1': {
                'address': {
                    'type': 'none_ip4'
                },
                'address6': {
                    'type': 'none_ip6'
                },
                'description': 'eth1',
                'mtu': 1500,
                'speed': 'AUTO',
                'type': 'physical'
            },
            'eth2': {
                'address': {
                    'type': 'none_ip4'
                },
                'address6': {
                    'type': 'none_ip6'
                },
                'description': 'eth2',
                'mtu': 1500,
                'speed': 'AUTO',
                    'type': 'physical'
            }
        },
        'license': 'UNLIMITED',
        'management_interface': 'eth0',
        'name': 'Test Agent 2',
        'ntp_config': {
            'interface_name': 'eth0',
            'server': 'time.google.com',
            'enable_ipv6': False
        },
        'type': 'appliance'
})

# Create Test Agent
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

```
print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

> ⚠ **WARNING**: If no signed SSL certificates are present, you need to add `verify=False` to the "requests" command: see below. This is however strongly discouraged in a production environment. You should obtain proper, signed SSL certificates to ensure an encrypted and secure connection. See also the section on SSL certificate configuration in the Installation Guide, chapter Service Configuration.

```
# Create Test Agent
response = requests.post(url=URL, data=json_data, headers={
    'API-Token': TOKEN,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json'
  },
  verify=False
)
```

Just to show that you might as well use a different programming language, here is how to accomplish the same thing in curl:

```
curl -X POST "https://<Control Center host and port>/accounts/demo/test_agents/"
                  -H "accept: application/json" -H "content-type: application/json" -d
"{ \"description\":
                  \"Created via REST API\", \"interface_config\": { \"eth0\": { \"address\":
{ \"type\":
                  \"dhcp_ip4\" }, \"address6\": { \"type\": \"dhcp_ip6\" }, \"description\":
\"eth0\",
                  \"mtu\": \"1500\", \"speed\": \"AUTO\", \"type\": \"physical\" }, \"eth1\":
{ \"address\":
                  { \"type\": \"none_ip4\" }, \"address6\": { \"type\": \"none_ip6\" },
\"description\":
                  \"eth1\", \"mtu\": \"1500\", \"speed\": \"AUTO\", \"type\": \"physical\" } },
\"license\":
                  \"UNLIMITED\", \"management_interface\": \"eth0\", \"name\": \"vta1\",
\"ntp_config\":
                  { \"interface_name\": \"eth0\", \"server\": \"time.google.com\" }, \"type\":
\"appliance\" }"
```

In the absence of signed SSL certificates, you need to add the `--insecure` flag here.

Once the Test Agent has been created, it will exist in the configuration database and in Control Center. See the screenshot below of the Test Agent inventory, showing the Test Agent "vta1":
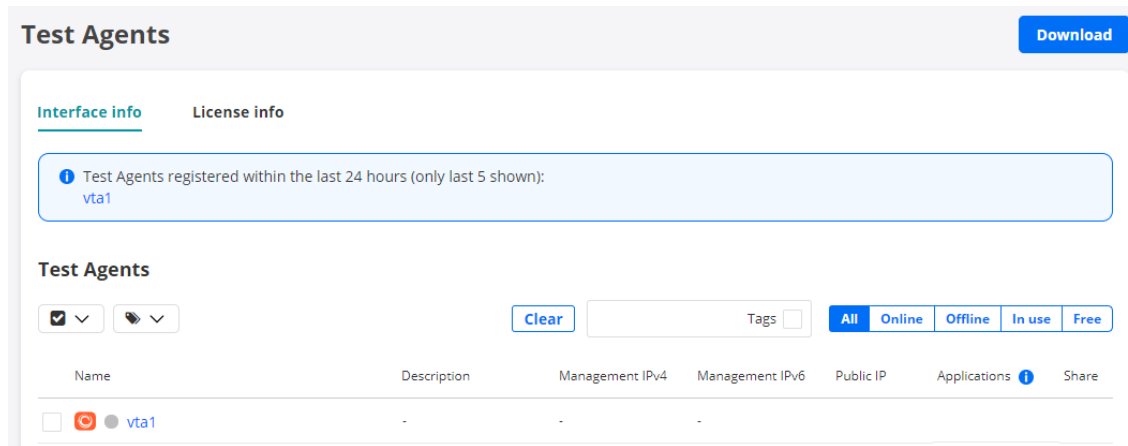
Step 3: It is now time to deploy the Test Agent "vta1".

In this guide we will deploy the virtual Test Agent in OpenStack. However, it is equally possible to do the deployment in some other virtualized environment.

In OpenStack the Test Agent will use cloud-init user data to retrieve the information on how to connect to Control Center. Specifically, the user data text file has the following contents:

> **NOTE**: The `#cloud-config` and `netrounds_test_agent` lines must be present, and the remaining lines must be indented.

```
#cloud-config
netrounds_test_agent:
  name: MyvTA                          # Name of vTA to appear in Control Center inventory
  email: john.doe@example.com          # Email you use when logging in to the system
  password: secret                     # Login password
  account: theaccount                  # Account name
  server: <login-server>:<port>        # Control Center host and port (default == SaaS)
                                       #   Note: With an IPv6 server address the whole string
                                       #         including port must be in double quotes
```

For further information, please refer to the document "How to Deploy Virtual Test Agents in OpenStack", available at https://www.juniper.net/documentation/product/en_US/paragon-active-assurance.

Once the Test Agent has been deployed and has connected to Control Center, the configuration will be pushed from Control Center to the Test Agent.
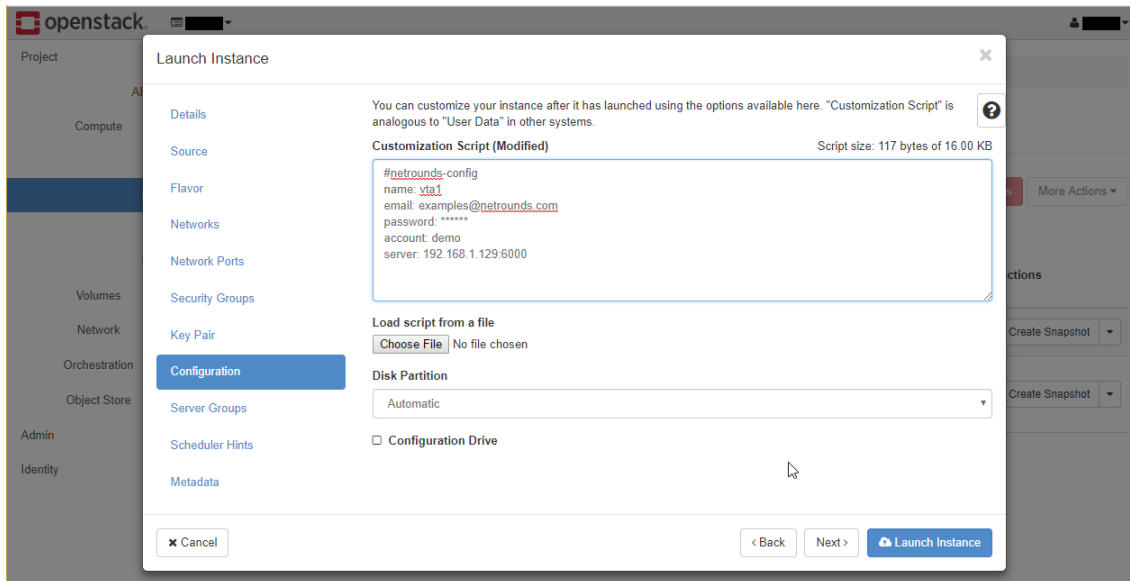
Step 4: The Test Agent is now online in Control Center and has obtained its configuration. The Test Agent is ready for use in tests and monitoring. See these sections:

- "Creating and Running a Test" on page 76

- "Creating a Monitor" on page 63 and "Starting and Stopping a Monitor" on page 69

Listing the Test Agents in Your Paragon Active Assurance Account

Below is example Python code for listing the Test Agents in a Paragon Active Assurance account:

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/test_agents/%s' % (args.ncc_url, args.account, args.query_params)


# Get list of Test Agents
response = requests.get(url=url, headers={'API-Token': args.token})
```

Running this code gives output similar to that below:

```
{
  "count": 2,
  "exclude_default": [
    "interface_config",
    "ntp_config",
    "management_interface",
    "interface_names",
    "interface_states",
    "uptime",
    "memory",
    "cpu",
    "load_avg"
  ],
  "items": [
    {
      "description": "Created via REST API",
      "gps_lat": 22.222222,
      "gps_long": -33.333333,
      "id": 1,
      "interface_names": [
        "eth0",
        "eth1"
      ],
      "is_owner": true,
      "license": "SW_LARG",
      "name": "Test Agent A",
      "online": true,
      "type": "appliance",
      "use_public_address": false,
      "version": "3.3.0"
    }
    {
      ...
      "name": "Test Agent B",
      ... (same items listed for this Test Agent)
    }
  ],
  "limit": 10,
  "next": null,
  "offset": 0,
```

```
    "previous": null
}
```

Retrieving Configuration Data and Status for a Test Agent

By applying a GET operation to an individual Test Agent, you retrieve both configuration and status information for the Test Agent and its interfaces. The status information includes online status, uptime, CPU load, and memory usage.

```
# Request settings
url = '%s/accounts/%s/test_agents/%s/' % (args.ncc_url, args.account, args.test_agent_id)

# Get Test Agent
response = requests.get(url=url, headers={'API-Token': args.token})
```

The output will look something like this:

```
{
  "cpu": 0.8,
  "description": "",
  "gps_lat": 22.222222,
  "gps_long": -33.333333,
  "hidden": false,
  "id": 5,
  "interface_config": {
    "eth0": {
      "address": {
        "type": "dhcp_ip4",
        "vendor": null
      },
      "address6": {
        "type": "none_ip6"
      },
      "description": "",
      "mac": null,
      "management": true,
      "mtu": 8900,
      "speed": "AUTO",
      "tags": [],
```

```
        "type": "physical"
      },
      "eth1": {
        "address": {
          "dhcpd": null,
          "dns": [],
          "ip": "10.1.1.66/24",
          "routes": {},
          "type": "static_ip4"
        },
        "address6": {
          "dns": null,
          "ip": "2001:0DB8::1/64",
          "routes": {},
          "type": "static_ip6"
        },
        "description": "",
        "mac": null,
        "management": false,
        "mtu": 1500,
        "speed": "AUTO",
        "tags": [],
        "type": "physical"
      }
    },
    "interface_states": {
      "eth0": {
        "ip4_address": "192.168.100.4/24",
        "ip6_address": [],
        "mac_hw": "08:00:27:30:be:03"
      },
      "eth1": {
        "ip4_address": "10.1.1.66/24",
        "ip6_address": [
          "2001:0DB8::1/64"
        ],
        "mac_hw": "08:00:27:80:57:51"
      }
    },
    "is_owner": true,
    "license": "SW_MEDI",
    "load_avg": [
      0,
```

```
      0.01,
      0.05
    ],
    "management_interface": "eth0",
    "memory": 22.18,
    "name": "VTA1",
    "ntp_config": {
      "interface_name": "eth0",
      "server": "time.google.com"
    },
    "online": true,
    "tags": [],
    "type": "appliance",
    "uptime": 47319,
    "use_public_address": false,
    "version": "2.23.0"
  }
```

Modifying the Configuration of a Test Agent

To modify the configuration of a Test Agent you use the PUT command.

PUT requires that you supply the entire configuration in the JSON data, just as when creating the Test Agent as described in the section . The code for doing this is therefore the same as for Test Agent creation, except that the URL points to a specific, existing Test Agent

```
# Request settings
url = '%s/accounts/%s/test_agents/%s/' % (args.ncc_url, args.account, args.test_agent_id)
```

and the PUT command

```
# Update Test Agent
response = requests.put(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

is used instead of POST.

Updating Test Agent Software

The REST API provides a POST operation for updating Test Agent software to the latest version, either for all Test Agents or for a specified subset.

Below is an example of how to apply a software update to a subset of Test Agents:

```
# Request settings
url = '%s/accounts/%s/test_agents/update/' % (args.ncc_url, args.account)

# JSON content: Subset of Test Agents on which to update software
# POST http://server/accounts/acount_name/test_agents/update
json_data = json.dumps({
    'test_agent_ids': args.test_agent_ids.split(',')
})

# Update Test Agent version
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

If you want to update software on all Test Agents, leave `json_data` empty and append the `?all` switch to the URL:

```
# Request settings
url = '%s/accounts/%s/test_agents/update/?all' % (args.ncc_url, args.account)
```

Test Agents: Advanced Examples

In order to make use of all configuration options for the Test Agent (or indeed any other entity in the REST API), you need to be familiar with the REST API Swagger schema.

As previously mentioned, a link to the schema in JSON format is provided at the top of the page. This is rather hard to read; to make the content easier to parse, you can copy the file into the following page:

https://editor.swagger.io/

which will convert the schema to more human-readable YAML.

Note that in the examples below, some parts of the schemas have been rearranged for greater readability, rather than sorted alphabetically.

For example, say that we want to configure a Test Agent with a static IPv4 address.

We will do this by construcing the body of a HTTP POST request.

The end goal is to construct this request:

```
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent",
  "interface_config": {
    "eth0": {
      "type": "physical",
      "address": {
        "type": "static_ip4",
        "ip": "192.168.0.123/24"
      }
    }
  }
}
```

The question is, how do we know this is how we should format it?

To find out the correct syntax, we need to consult the Test Agent schema. In fact, we need to check out the extended version of it; search for TestAgentExtendedSchema in the Swagger schema:

```
TestAgentExtendedSchema:
  properties:
    ...name:
      type: stringmaxLength: 100minLength: 1...description:
      type: stringmaxLength: 100
...
```

To begin with, this schema has `name` and `description` as attributes. The HTTP request `body` should therefore begin as follows:

```json
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent"
}
```

Some of the `properties` are defined as `read-only`. These can be omitted from the request, since they will be ignored by the server.

Now for the interface configuration, and more specifically the IP address, which take a bit more work.

The address is specified in the `interface_config` property, which contains a reference to `InterfaceConfigSchema` as seen below:

```yaml
TestAgentExtendedSchema:
type: objectproperties:
  ...interface_config:
    type: objectadditionalProperties:
      $ref: '#/definitions/InterfaceConfigSchema'
```

We can start by adding to our request:

```json
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent",
  "interface_config": {}
}
```

This would add an empty `interface_config`, so to understand how to construct the `interface_config` JSON object we must go to the `InterfaceConfigSchema` part of the schema:

```yaml
InterfaceConfigSchema:
  type: objectrequired:
  - typediscriminator: typeproperties:
    description:
      type: stringtype:
      type: stringenum:
      - physical
      - vlan
```

```
      - bridge
      - bridged
      - mobile
      - wifi
```

The schema has a property named `type` which is required. This is used to specify which type of interface we want to define. This `type` has a list of valid choices, which is represented in the schema as an `enum`.

The `discriminator` part of the schema is used to point us to the specific type we need to use.

For people familiar with object-oriented programming (OOP) an analogy to this would be "inheritance", or polymorphism.

The `InterfaceConfigSchema` has a list of subtypes which are defined in the `enum`.

In our case, we want a "normal" network interface, so we should specify `physical` as our `type`.

So now we can look up the schema for the `physical` type:

```
physical:
  description: A representation of Physical Interface
  allOf:
    - $ref: '#/definitions/InterfaceConfigSchema'
    - properties:
        type: objectaddress:
          $ref: '#/definitions/InterfaceIPv4AddressSchema'address6:
          $ref: '#/definitions/InterfaceIPv6AddressSchema'...
```

Here the `allOf` attribute is used to say that the `physical` interface type has all properties from `InterfaceConfigSchema`, as well as the `properties` listed inline.

That is, it "inherits" the `properties` from `InterfaceConfigSchema` in OOP terminology.

So we can continue constructing our HTTP `POST` request `body`. Since we want a static IPv4 address, we specify `address` rather than `address6` which is for IPv6.

```
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent",
  "interface_config": {
    "eth0": {
      "type": "physical",
      "address": {}
    }
```

```
  }
}
```

So, using the same process we now look up `InterfaceIPv4AddressSchema`:

```
InterfaceIPv4AddressSchema:
  type: objectrequired:
  - typediscriminator: typeproperties:
    type:
      type: stringenum:
        - none_ip4
        - static_ip4
        - dhcp_ip4
```

Again, we specify the `type`, which is used to identify the specific schema to use. We want `static_ip4` as found in the `enum`:

```
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent",
  "interface_config": {
    "eth0": {
      "type": "physical",
      "address": {
        "type": "static_ip4"
      }
    }
  }
}
```

We then look up the schema for `static_ip4`:

```
static_ip4:
  description: IPv4 static addressallOf:
  - $ref: '#/definitions/InterfaceIPv4AddressSchema'
  - properties:
    type: object...ip:
        type: string...
```

Here we can see that we need to specify the `ip` property. In general, in Netrounds, IP addresses are entered using the CIDR notation.

All the other `properties` here are optional.

So now we can finalize our JSON:

```json
{
  "name": "Test Agent 1",
  "description": "This is my Test Agent",
  "interface_config": {
    "eth0": {
      "type": "physical",
      "address": {
        "type": "static_ip4",
        "ip": "192.168.0.123/24"
      }
    }
  }
}
```

Deleting a Test Agent

After a test has completed, it might be relevant in some use cases to delete the Test Agent.

Below is code for doing this through the REST API:

```python
# Request settings
url = '%s/accounts/%s/test_agents/%s/' % (args.ncc-url, args.account, args.test-agent-id)

# Delete Test Agent
response = requests.delete(url=url, headers={'API-Token': args.token})
```

Examples: Inventory Items

Creating (importing) and managing inventory items such as TWAMP reflectors and Y.1731 MEPs is done in a similar way as for Test Agents. Below is Python code for defining such entities in Paragon Active Assurance through the REST API and for retrieving lists of the items defined.

TWAMP Reflectors

Creating a TWAMP Reflector

```python
# Request settings
url = '%s/accounts/%s/twamp_reflectors/' % (args.ncc_url, args.account)

# JSON content
json_data = json.dumps({
    'ctrl_port': 862,  # Control port (optional)
    'host': '10.0.33.1',  # Host name or IP address
    'name': 'TWAMP Reflector A',
    'port': 9876  # Test port
})

# Create TWAMP reflector
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
```

```
    'Content-Type': 'application/json',
})
```

Retrieving TWAMP Reflectors

Here is how to retrieve all TWAMP reflectors defined in an account.

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/twamp_reflectors/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of TWAMP reflectors
response = requests.get(url=url, headers={'API-Token': args.token})
```

Running this code gives output like that below:

```
{
  "count": 1,
  "items": [
    {
      "ctrl_port": 862,
      "gps_lat": 22.222222,
      "gps_long": -33.333333,
      "host": "10.0.33.1",
      "id": 1,
      "name": "TWAMP Reflector A",
      "port": 9876,
      "tags": []
    }
  ],
  "limit": 10,
  "next": null,
  "offset": 0,
  "previous": null

}
```

Y.1731 MEPs

Creating a Y.1731 MEP

```python
# Request settings
url = '%s/accounts/%s/y1731_meps/' % (args.ncc_url, args.account)

# JSON content
json_data = json.dumps({
    'mac': '00:11:22:33:AA:BB',  # MAC address
    'meg_level': 2,  # MEG (Maintenance Entity Group) level
    'name': 'Y.1731 MEP 1'
})

# Create Y.1731 MEP
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving Y.1731 MEPs

Here is how to retrieve all Y.1731 MEPs defined in an account.

```python
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/y1731_meps/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of Y.1731 MEPs
response = requests.get(url=url, headers={'API-Token': args.token})
```

Running this code gives output like that below:

```json
{
  "count": 1,
  "items": [
```

```
  {
    "id": 1,
    "mac": "00:11:22:33:AA:BB",
    "meg_level": 2,
    "name": "Y.1731 MEP 1"
  }
],
"limit": 10,
"next": null,
"offset": 0,
"previous": null
}
```

IPTV Channels

Creating an IPTV Channel

```
url = '%s/accounts/%s/iptv_channels/' % (args.ncc_url, args.account)

# Parameter settings for IPTV channel
json_data = json.dumps({
    "name": 'Some TV Channel',
    "ip": '224.5.0.0',          # IPv4 multicast address of IPTV channel
    "port": 5500,               # UDP destination port in IPTV multicast stream
    "source": "172.16.0.1",     # Multicast source address (optional)
    "pnum": 1,                  # Program number (in Multi Program Transport Stream)
})

# Create IPTV channel
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving IPTV Channels

Below is code for retrieving all IPTV channels defined in an account.

```python
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/iptv_channels/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of IPTV channels
response = requests.get(url=url, headers={'API-Token': args.token})
```

Running this code gives output like that below:

```json
{
  "count": 1,
  "items": [
    {
      "id": 1,
      "ip": "224.5.0.0",
      "name": "Some TV Channel",
      "pnum": 1,
      "port": 5500,
      "source": "172.16.0.1"
    }
  ],
  "limit": 10,
  "next": null,
  "offset": 0,
  "previous": null
}
```

SIP Accounts

Creating a SIP Account

```
# Request settings
url = '%s/accounts/%s/sip_accounts/' % (args.ncc_url, args.account)

# Parameter settings for SIP account
json_data = json.dumps({
    "sip_domain": "example.com",     # SIP domain name
    "registrar": "10.10.10.10",      # Domain name/IP address of SIP registrar (optional)
    "username": "my_user",           # User name for registration
    "password": "my_password",       # Password for registration
    "proxy": "example.com",          # Proxy server IP (optional)
    "user_auth": "my_auth_id",       # User ID for authentication (optional)
    "uri_rewrite": "abc"             # User ID in URI as rewritten by SIP server (optional)
})

# Create IPTV channel
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving SIP Accounts

To retrieve all SIP accounts defined in a Paragon Active Assurance account, do as follows.

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/sip_accounts/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of SIP accounts
response = requests.get(url=url, headers={'API-Token': args.token})
```

Running this code gives output like that below:

```
{
  "count": 1,
  "items": [
    {
      "password": "my_password",
      "proxy": "192.168.0.1",
      "registrar": "",
      "sip_domain": "example.com",
      "uri_rewrite": "",
      "user_auth": "my_auth_id",
      "username": "my_user"
    }
  ],
  "limit": 10,
  "next": null,
  "offset": 0,
  "previous": null
}
```

Examples: Alarms

IN THIS SECTION

Monitors can have alarms associated with them. This section explains how to do the following through the REST API:

- Creating alarm templates

- Setting up SNMP managers and lists of email recipients, specifying where to send alarms when they are triggered

We will begin with the alarm recipients, as these are a prerequisite when defining an alarm template. We then proceed to show how to create such templates, which among other things hold the trigger conditions for alarms.

Instances of alarms cannot be created in isolation through the REST API. Rather, alarm instances are created in the course of creating a monitor (most conveniently by referring to an alarm template, but optionally without doing so), as described in the section "Creating a Monitor with an Alarm" on page 67. However, you can perform some operations on alarm instances through the API:

- list alarm instances

- retrieve an individual alarm instance

- suppress an alarm instance so that it will never trigger, even if thresholds are exceeded

- delete an alarm instance.

Alarm Email Lists

Creating an Alarm Email List

```
# Request settings
url = '%s/accounts/%s/alarm_emails/' % (args.ncc_url, args.account)


# Parameter settings for alarm email list
json_data = json.dumps({
    "addresses": [    # Email addresses to include in list
        "a@a.com"
    ],
    "name": "Email list 1"
})


# Create alarm email list
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving All Alarm Email Lists

Here is how to retrieve a list of all alarm email lists and see what IDs they have been assigned.

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/alarm_emails/%s' % (args.ncc_url, args.account, args.query_params)


# Get list of alarm emails
response = requests.get(url=url, headers={'API-Token': args.token})
```

SNMP Managers

Creating an SNMP Manager

The URL should be defined as shown below.

```
# Request settings
url = '%s/accounts/%s/snmp_managers/' % (arg.ncc_url, args.account)
```

To create a *version 2c* SNMP manager, use the following settings:

```
# Parameter settings for SNMP version 2c
json_data = json.dumps({
  "community": "Community string",
  "ip": "8.8.8.8",
  "version": "2c"
})
```

To create a *version 3* SNMP manager, use the following settings:

```
# Parameter settings for SNMP Manager in version 3
json_data = json.dumps({
    "auth_password": "12345678",
    "engine_id": "080005f85050000001",
```

```
    "ip": "8.8.8.8",
    "priv_password": "12345678",
    "security": "no_auth_no_priv",
    "user_name": "my_user_name",
    "version": "3",
    "name": "SNMP manager"
})


# Create SNMP manager
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving All SNMP Managers

Here is how to retrieve a list of all SNMP managers and see what IDs they have been assigned.

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/snmp_managers/%s' % (args.ncc_url, args.account, args.query_params)


# Get list of SNMP managers
response = requests.get(url=url, headers={'API-Token': args.token})
```

Alarm Templates

Creating an Alarm Template

In practice, probably only one of `email` and `snmp` will be used.

```
# Request settings
url = '%s/accounts/%s/alarm_templates/' % (args.ncc_url, args.account)


# Parameter settings for alarm template
```

```python
json_data = json.dumps({
    "action": "",
    "email": 1,  # ID of email list previously defined
    "interval": 300,
    "name": "Alarm template",
    "snmp": 1,  # ID of SNMP manager previously defined
    "thr_es_critical": 10,
    "thr_es_critical_clear": 9,
    "thr_es_major": 8,
    "thr_es_major_clear": 7,
    "thr_es_minor": 6,
    "thr_es_minor_clear": 5,
    "thr_es_warning": 4,
    "thr_es_warning_clear": 3,
    "window_size": 60,
    "send_only_once": False,
    "no_data_timeout": 1800,
    "snmp_trap_per_stream": False,  # Set this to True to send SNMP traps per stream
    "no_data_severity": 1 # CLEAR = 0 | WARNING = 1 | MINOR = 2 | MAJOR = 3 | CRITICAL = 4
})


# Create alarm template
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Retrieving All Alarm Templates

```python
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/alarm_templates/%s' % (args.ncc_url, args.account, args.query_params)


# Get list of alarm templates
response = requests.get(url=url, headers={'API-Token': args.token})
print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Modifying an Existing Alarm Template

If you need to adjust some settings in an alarm template, you can do so as follows:

```python
# Request settings
url = '%s/accounts/%s/alarm_templates/%s/' % (args.ncc_url, args.account, args.alarm_template_id)

# JSON content
json_data = json.dumps({  # Same list of parameters as when creating the template
    "action": "",
    "email": 2,  # ID of email list
    "interval": 600,
    "name": "Alarm template",
    "snmp": 2,  # ID of SNMP manager
    "thr_es_critical": 20,
    "thr_es_critical_clear": 9,
    "thr_es_major": 8,
    "thr_es_major_clear": 7,
    "thr_es_minor": 6,
    "thr_es_minor_clear": 5,
    "thr_es_warning": 4,
    "thr_es_warning_clear": 3,
    "window_size": 60,
    "send_only_once": False,
    "no_data_timeout": 1800,
    "snmp_trap_per_stream": False,
    "no_data_severity": 1
})

# Update alarm template
response = requests.put(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Examples: SSH Keys

You can add SSH public keys to a Test Agent via the REST API. Using the corresponding private key you can then log in to the Test Agent via SSH.

The full list of available operations on SSH keys is as follows:

- Add an SSH key

- Modify an SSH key

- Inspect an SSH key

- List SSH keys

- Delete an SSH key.

Below, some of these operations are exemplified.

Adding an SSH Key

Here is how to create a new SSH key.

```
# Request settings
url = '{}/accounts/{}/ssh_keys/?test_agent_id={}'.format(args.ncc_url,
args.account,
args.test_agent_id)

# JSON content
json_data = json.dumps({
'key': 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCWDqRALdALDJARxa85',
```

```
'name': 'SSH key'
})


# Create the SSH key
response = requests.post(url=url, data=json_data, headers={
'API-Token': args.token,
'Accept': 'application/json; indent=4',
'Content-Type': 'application/json',
})
```

Retrieving All SSH Keys

To list all SSH keys, proceed as follows:

```
# request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: changes number of elements returned from api
# offset: changes element from which results will be returned
url = '{}/accounts/{}/ssh_keys/?test_agent_id={}'.format(args.ncc_url,
args.account,
args.test_agent_id)


# Get a list of SSH keys
response = requests.get(url=url, headers={'API-Token': args.token})
```

Deleting an SSH Key

If you want to delete an SSH key, use the following command:

```
# request settings
url = '{}/accounts/{}/ssh_keys/{}/?test_agent_id={}'.format(args.ncc_url,
args.account,
args.ssh_key_id,
args.test_agent_id)
```

```
# Delete SSH key
response = requests.delete(url=url, headers={'API-Token': args.token})
```

Examples: Monitors

IN THIS SECTION

This section assumes that Test Agents (as many as are required by the monitors) have been created according to the section "Creating and Deploying a New Test Agent" on page 31.

Overview of Monitor Orchestration

Before you can create and start a monitor through the REST API, you must have a template on which to base the monitor defined in Control Center, as explained in the chapter "Test and Monitor Templates" on page 28. All parameters specified in that template as **Template input** then need to be assigned values when you create a monitor from it through the REST API.

Creating a Monitor

Suppose that two templates have been set up in Control Center: one for UDP monitoring between two Test Agent interfaces, and another where a Test Agent acts as TWAMP initiator towards a TWAMP reflector.
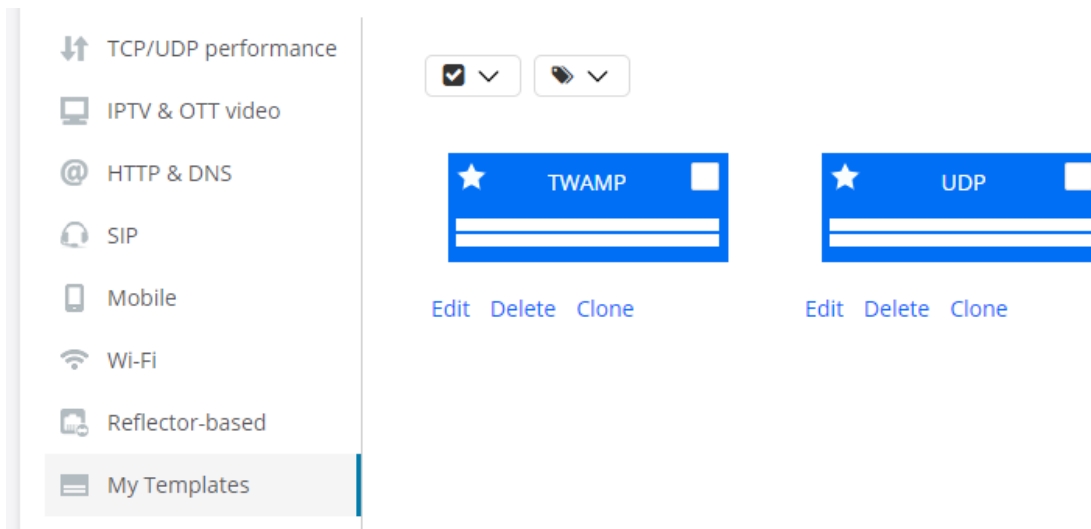
Below is Python code for listing the monitor templates in an account through the REST API:

```python
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/monitor_templates/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of monitor templates
response = requests.get(url=url, headers={'API-Token': args.token})
```

The output will look something like this (below, two monitor templates have been defined):

```json
{
  "count": 2,
  "items": [
    {
      "description": "",
      "id": 1,
      "inputs": {
        "clients": {
```

```json
        "input_type": "interface_list"
      },
      "server": {
        "input_type": "interface"
      }
    },
    "name": "UDP"
  },
  {
    "description": "",
    "id": 2,
    "inputs": {
      "rate": {
        "input_type": "float"
      },
      "reflectors": {
        "input_type": "twamp_reflector_list"
      },
      "senders": {
        "input_type": "interface_list"
      },
      "time_sync": {
        "input_type": "string"
      },
      "time_sync_inband": {
        "input_type": "string"
      }
    },
    "name": "TWAMP"
  }
],
"limit": 10,
"next": null,
"offset": 0,
"previous": null
}
```

If you want to inspect just a single template, you can do so as follows, indicating the template's ID:

```python
# Request settings
url = '%s/accounts/%s/monitor_templates/%s/' % (args.ncc_url, args.account,
args.monitor_template_id)  # Template ID specified in last argument
```

```
# Get monitor template
response = requests.get(url=url, headers={'API-Token': args.token})
```

Continuing the previous example, if you run this code it will produce the output below:

```
{
  "description": "",
  "id": 1,
  "inputs": {
    "clients": {
      "input_type": "interface_list"
    },
    "server": {
      "input_type": "interface"
    }
  },
  "name": "UDP"
}
```

Now suppose you want to create a monitor based on the TWAMP template. This is done using the POST operation for monitors. By default, the monitor will also start immediately as a result of this operation (`started` defaults to true). Alternatively, you can set `started` to False and use a separate operation to start up the monitor: see the section "Starting and Stopping a Monitor" on page 69.

You need to provide values for the parameters under `inputs`, which are left to be defined at runtime. The parameter names are those defined as **Variable name** in Control Center. Here, they are simply lowercase versions of the Control Center display names ("senders" vs. "Senders", etc.).

Below is code supplying the required parameter settings for the monitor. For a monitor template with a different set of inputs, the details of this procedure will of course differ.

In this example, no *alarm* is associated with the monitor. For examples involving alarms, go to the section "Creating a Monitor with an Alarm" on page 67.

```
# Request settings
url = '%s/accounts/%s/monitors/' % (args.ncc_url, args.account)

# JSON content
json_data = json.dumps({
    'name': 'TWAMP monitor',
    'input_values': {
```

```
        'senders': {
            'input_type': 'interface_list',
            'value': [{
                'test_agent_id': 1,
                'interface': 'eth0',
                'ip_version': 4
            }]
        },
        'reflectors': {
            'input_type': 'twamp_reflector_list',
            'value': [1]
        },
        'rate': {
            'input_type': 'float',
            'value': 1.0
        },
        'time_sync': {
            'input_type': 'string',
            'value': '0'
        }
    },
    'started': True,  # Set this to False in order not to start the monitor
    'template_id': 1  # Reference to monitor template
})


# Create monitor
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Some comments on the `senders` input value are helpful here to explain how the input is structured.

This input value has `input_type = interface_list`, so in its `value` you need to provide a list of Test Agent interfaces. In the example above, a list of two interfaces is passed. For each interface we need to specify the Test Agent ID, the Test Agent interface, and the IP version to use.

Note that IPv6 is supported only for certain task types (as detailed in the support documentation), so `ip_version` = 6 is a valid setting only for those tasks.

Creating a Monitor with an Alarm

To associate an alarm with a monitor, you can either point to an alarm template that has been defined, or you can supply the entire alarm configuration with the POST operation. We will give one example of each approach below.

Setting Up a Monitor Alarm by Pointing to an Alarm Template

In order to make use of an alarm template, you must know its ID. To this end, first retrieve all alarm templates as described in the section and note the `id` value of the relevant template. Suppose this ID is "3". You can then refer to that template as follows:

```python
# Request settings
url = '%s/accounts/%s/monitors/' % (args.ncc_url, args.account)


# JSON content
json_data = json.dumps({
    'name': 'Monitor from template with alarm template',
    'input_values': {
```

Supply monitor input values here as in the previous example.

```python
    'started': True,        # Set this to False in order not to start the monitor
    'template_id': 1,       # Reference to _monitor_ template
    'alarm_configs': [{
        'template': 2,      # Reference to _test_ template
```

(Some optional parameters are omitted here.)

```python
    }]
})


# Create monitor
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
```

```
    'Content-Type': 'application/json',
})
```

Setting Up a Monitor Alarm by Configuring It Directly

Alternatively, you can set up an alarm for a monitor by supplying its entire configuration when creating the monitor, without referring to an alarm template. This is done as shown in the following example.

```
# Request settings
url = '%s/accounts/%s/monitors/' % (args.ncc_url, args.account)

# JSON content
json_data = json.dumps({
    'name': 'Monitor from template with alarm config.',
    'input_values': {
```

Supply monitor input values here.

```
    },
    'template_id': 1,
    'alarm_configs': [{
        'email': 1,
        'snmp': 1,
        'thr_es_critical': 7,
        'thr_es_critical_clear': 6,
        'thr_es_major': 5,
        'thr_es_major_clear': 4,
        'thr_es_minor': 3,
        'thr_es_minor_clear': 2,
        'thr_es_warning': 1,
        'thr_es_warning_clear': 0,
        'window_size': 60,
        'interval': 3600,
        'action': 'Action text',
        'send_only_once': False,
        'no_data_timeout': 1800,
        'snmp_trap_per_stream': False,
        'no_data_severity': 1  # CLEAR = 0 | WARNING = 1 | MINOR = 2 | MAJOR = 3 | CRITICAL = 4
    }]
```

```
})

# Create monitor
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

Starting and Stopping a Monitor

If the monitor was not configured to start at creation time (`started` set to False), you need to apply a PUT or PATCH operation to start it (the two operations are equivalent). Below, the PATCH operation is shown.
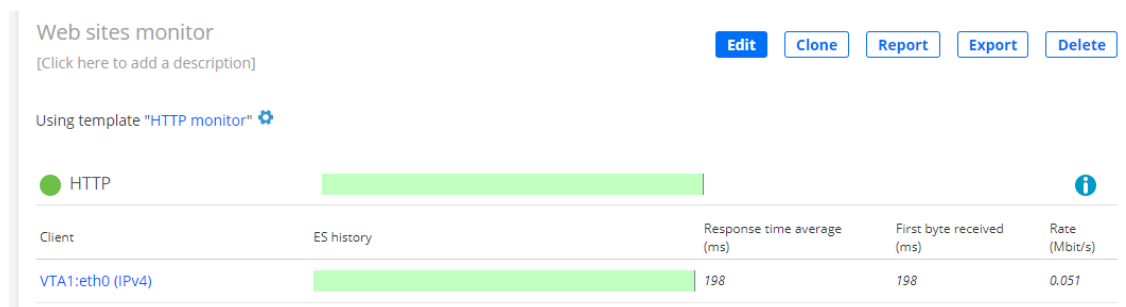
```
# Parameter settings for monitor
json_data = json.dumps({
    "started": True
}

# Request settings
url = '%s/accounts/%s/monitors/%s/' % (args.ncc_url, args.account, args.monitor_id)

# Start monitor
response = requests.patch(url=url, headers={'API-Token': args.token})
```

The monitor is now started:

To stop the monitor, use the same operation but with `started` set to False:

```
# Parameter settings for monitor
json_data = json.dumps({
    "started": False
}
```

Retrieving SLA Status and Data Metrics for a Monitor

Here is how to retrieve the SLA status and comprehensive data metrics for a monitor. This operation also fetches the complete configuration of the monitor.

By default, the SLA status is returned for each of the following time intervals: last 15 minutes, last hour, and last 24 hours. You can specify a different time interval, replacing the default ones, by including the `start` and `end` parameters in a query string at the end of the URL. The time is given in UTC (ISO 8601) as specified in IETF RFC 3339. An example is given below.

This operation can also return detailed data metrics for each task performed by the monitor. You turn on this feature by setting `with_detailed_metrics` to true (by default, this flag is set to false). The detailed data metrics are found under `tasks` > `streams` > `metrics` and are given for successive time intervals whose length are determined by the `resolution` parameter. The default and maximum resolution is 10 seconds. The resolution value entered is converted into one of the resolutions available: 10 seconds, 1 minute, 5 minutes, 20 minutes, or 4 hours.

Averaged metrics are returned by default. You can turn these off by setting `with_metrics_avg` to false in the query string. Average metrics are by default computed for the last 15 minutes and are found in `tasks` > `streams` > `metrics_avg`. If you specify a different time interval by `start` and `end`, averaged metrics will be returned for that interval instead.

The output also includes monitor logs.

Example (with default resolution 10 seconds for detailed data metrics):

```
# Detailed metrics
with_detailed_metrics = 'true'


# Request settings
url = '%s/accounts/%s/monitors/%s/?with_detailed_metrics=%s' % (args.ncc_url, args.account,
args.monitor_id, with_detailed_metrics)
```

```
# Get monitor
response = requests.get(url=url, headers={'API-Token': TOKEN})
```

The output will be similar to the following:

```
{
  "description": "This is a multisession TCP monitor",
  "gui_url": "https://localhost/dev/monitoring/11/?start=1507127679&end=1507128579",
  "id": 18,
  "last_15_minutes": {
    "end_time": "2017-10-04T14:49:39",
    "sla": 99.67,
    "sla_status": "acceptable",
    "start_time": "2017-10-04T14:34:39"
  },
  "last_24_hours": {
    "end_time": "2017-10-04T14:49:39",
    "sla": 99.98,
    "sla_status": "good",
    "start_time": "2017-10-03T14:49:39"
  },
  "last_hour": {
    "end_time": "2017-10-04T14:49:39",
    "sla": 99.86,
    "sla_status": "acceptable",
    "start_time": "2017-10-04T13:49:39"
  },
  "name": "MultiTCPMonitor",
  "report_url": "https://localhost/dev/monitoring/11/report_builder/?
start=1507127679&end=1507128579",
  "started": true,
  "tags": [],
  "tasks": [
    {
      "name": "MultiTCP",
      "task_type": "multitcp",
      "config": {
        "acceptable_sla": 99.5,
        "client": {
          "ip_version": 4,
          "name": "eth1",
          "preferred_ip": null,
```

```
        "test_agent": 2
    },
    "connections": 100,
    "delay": 0,
    "direction": "down",
    "down_es_connected": 10,
    "down_es_rate": 0.2,
    "down_es_max_rate": 0.2,
    "good_sla": 99.95,
    "port": 5000,
    "server": {
      "ip_version": 4,
      "name": "eth1",
      "preferred_ip": null,
      "test_agent": 1
    },
    "up_es_connected": 10,
    "up_es_rate": 0.2,
    "up_es_max_rate": 0.2
  }
]
"streams": [
  {
    "direction": "down",
    "gui_url": "https://localhost/dev/results/27/rrd_graph/?start=1507127679&end=1507128579",
    "id": 27,
    "is_owner": true,
    "metrics": [
      [
        "2017-10-04T13:34:39",
        "2017-10-04T13:34:49",
        "10",
        ... (remaining metrics omitted here)
      ]
      [
        ... (metrics for next 10 second interval)
      ]
      ... (further batches of metrics for successive 10 second intervals)
    ]
    "metrics_avg": {
      "start_time": "2017-10-04T14:34:39",
      "end_time": "2017-10-04T14:49:39",
      "active": 10,
```

```
        "connected": 10,
        "disconnected": 0,
        "es": 0.33,
        "es_connected": 0.33,
        "es_rate": 0.00,
        "rate": 12.49,
        "sla": 99.67,
        "sla_status": "acceptable"
      },
      "metrics_headers": [
        "start_time",
        "end_time",
        ... (rest omitted)
      ]
      "metrics_headers_display": [
        "Start time",
        "End time",
        ... (rest omitted)
      ]
      "name": "VTA1:eth1 (IPv4) (server) -> VTA2:eth1 (IPv4) (client)"
    }
  ],
  "last_3_logs": [
    {
      "level": "warning",
      "message": "Monitoring log message"
    }
  ]
}
```

Here is how to specify "start" and "end" times and the time resolution of detailed data metrics in a query string:

```
# Custom time interval
start = '2017-10-04T12:00:00.000000Z'
end = '2017-10-04T12:30:00.000000Z'

# Detailed metrics
with_detailed_metrics = 'true'

# Time resolution of metrics
resolution = '60'
```

```
# Request settings
url = '%s/accounts/%s/monitors/%s/?start=%s&end=%s&with_detailed_metrics=%s&resolution=%s' %
(args.ncc_url, args.account, args.monitor_id, start, end, with_detailed_metrics, resolution)
```

You can also retrieve all monitors with their SLAs in one go. However, in this case, no detailed data metrics are included in the export (the `tasks` > `streams` item is omitted). This is to limit the size of the output if the number of monitors is large.

```
# Request settings
url = '%s/accounts/%s/monitors/' % (args.ncc_url, args.account)

# Get monitor and its SLA
response = requests.get(url=url, headers={'API-Token': args.token})
```

Generating a PDF Report on a Monitor

You can generate a PDF report on a monitor directly from the REST API. The report has the same format as that generated from the Control Center GUI.

By default, the report covers the last 15 minutes. You can specify a different time interval by including the `start` and `end` parameters in a query string at the end of the URL. The time is given in UTC (ISO 8601) as specified in IETF RFC 3339.

In addition, the following options can be included in the query string:

- `worst_num`: For each task in a monitor, you can specify how many measurement results to show, ranked by the number of errored seconds with the worst on top. The scope of a measurement result is task-dependent; to give one example, for HTTP it is the result obtained for one client. The default number is 30.

- `graphs`: Include graphs in the report.

Example:

```
# Include graphs
graphs = 'true'

# Request settings
url = '%s/accounts/%s/monitors/%s/pdf_report?graphs=%s' % (args.ncc_url, args.account,
```

```
  args.monitor_id, graphs)


  # Get monitor
  response = requests.get(url=url, headers={'API-Token': args.token})


  print 'Status code: %s' % response.status_code
  print json.dumps(response.json(), indent=4)
```

Examples: Tests

IN THIS SECTION

This section assumes that Test Agents (as many as are required for the tests) have been created according to "Creating and Deploying a New Test Agent" on page 31.

Overview of Test Orchestration

Before you can create and run a test through the REST API, you must have a template on which to base the test defined in Control Center, as explained in the chapter "Test and Monitor Templates" on page 28. All parameters specified in that template as "Template input" then need to be assigned values when you create a test from it in the REST API.

Creating and Running a Test

The template we will use for our test in this example is an HTTP test template.

To inspect that template through the REST API, we retrieve a list of all test templates in the account:

```
# Request settings
# NOTE: User is able to pass additional parameters as a query string ?limit=100&offset=111:
# limit: Changes number of elements returned from API
# offset: Changes element from which results will be returned
url = '%s/accounts/%s/test_templates/%s' % (args.ncc_url, args.account, args.query_params)

# Get list of test templates
response = requests.get(url=url, headers={'API-Token': args.token})
```

If our HTTP template is the only test template defined, the response will look like this:

```
{
  "count": 1,
  "items": [
    {
      "inputs": {
        "clients": {
          "input_type": "interface_list"
        },
        "url": {
          "input_type": "string"
        }
      },
      "description": "This is a template for HTTP tests",
      "name": "HTTP_test",
      "id": 1
    }
  ],
  "next": null,
  "limit": 10,
  "offset": 0,
  "previous": null
}
```

The HTTP test in this template has two mandatory `inputs` left to be defined at runtime: `clients` (list of Test Agent interfaces playing the role of clients) and `url` (the URL to fetch using HTTP). The parameter names are those defined as **Variable name** in Control Center. Here, they are simply lowercase versions of the Control Center display names ("clients" vs. "Clients", etc.).

If there are multiple templates, and you want to inspect just a single template with a known ID, you can do that as follows:

```
# Request settings
url = '%s/accounts/%s/test_templates/%s/' % (args.ncc_url, args.account, args.monitor_id)

# Get test template
response = requests.get(url=url, headers={'API-Token': args.token})
```

We now create and run the HTTP test using the POST operation for tests.

Below is code supplying the required parameter settings for a test based on the HTTP test template. Depending on the structure of the template, the details here will of course vary. Another example with a slightly more complex test template is found in the section .

```python
# Request settings
url = '%s/accounts/%s/tests/' % (args.ncc_url, args.account)

# Parameter settings for test
json_data = json.dumps({
    "name": "REST API initiated HTTP test",
    "description": "This is an HTTP test initiated from the REST API",
    "input_values": {
        "clients": {
            "input_type": "interface_list",
            "value": [{
                "test_agent_id": 1,
                "interface": "eth0",
                "ip_version": 4
            }]
        },
        "url": {
            "input_type": "string",
            "value": "example.com"
        },
    "status": "scheduled",
    "template_id": 1
})

# Create and start test
response = requests.post(url=url, data=json_data, headers={
    'API-Token': TOKEN,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})
```

The execution of the test is displayed in Control Center:

Control Center will also respond to the REST API command with the ID of the test. In this example, the test ID is 47:

```
Status code: 201
{
  "template_id": 1,
  "description": "",
  "name": "HTTP_test",
  "id": 47
}
```

The test ID can also be found in the URL for the test in the Control Center web GUI. In this example, that URL is `https://<host IP>/<account>/testing/47/`.

Example with a Different Test Template

Here is another example of a test template: one for UDP, taking as input a server, a list of clients, and a UDP port number. In the Paragon Active Assurance GUI, this UDP template looks as follows:

To supply the inputs to this template, we can use the code below. Here, we have overridden the default value of port. If the default value (5000) is kept, the port section can be omitted from input_values.

```
# Request settings
url = '%s/accounts/%s/tests/' % (args.ncc_url, args.account)

# Parameter settings for test
json_data = json.dumps({
    "name": "REST API initiated UDP test",
    "description": "This is a UDP test initiated from the REST API",
    "input_values": {
        "clients": {
            "input_type": "interface_list",
            "value": [{
                "test_agent_id": 1,
                "interface": "eth0",
                "ip_version": 4
            }]
        },
        "server": {
```

```
            "input_type": "interface",
            "value": {
                "test_agent_id": 2,
                "interface": "eth0",
                "ip_version": 4
            }
        },
        "port": {
            "input_type": "integer",
            "value": "5050"
        }
    },
    "status": "scheduled",
    "template_id": 2
})


# Create and start test
response = requests.post(url=url, data=json_data, headers={
    'API-Token': TOKEN,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})


print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Retrieving Test Results

You retrieve results for a test by pointing to the test ID. This also fetches the full configuration of the test.

The basic test results consist of a passed/failed outcome for each test step and for the test as a whole.

By default, for tests where this is relevant, the test results also include averaged metrics taken over the full duration of the test. The average metrics are found in `tasks > streams > metrics_avg`. You can turn these average metrics off by setting `with_metrics_avg` to false in the query string.

Optionally, this operation can return detailed (second-by-second) data metrics for each task performed by the test (again, for tests which produce such data). You turn on this feature by setting `with_detailed_metrics` to true. By default, this flag is set to false. The detailed data metrics are found under `tasks > streams > metrics`.

There is yet another setting `with_other_results` which, if set to true, causes additional test results to be returned for the Path trace task type (routes and reroute events).

Example 1: TWAMP Test

A TWAMP test is an example of a test that produces metrics continuously.

Below is Python code for getting the results from a TWAMP test:

```python
# Request settings
url = '%s/accounts/%s/tests/%s/' % (args.ncc_url, args.account, args.test_id)

# Get test
response = requests.get(url=url, headers={'API-Token': args.token})
```

The output will look something like this:

```json
{
  "description": "",
  "end_time": "2017-09-03T20:05:04.000000Z",
  "gui_url": "https://<Control Center host and port>/demo/testing/48/",
  "name": "TWAMP_test",
  "report_url": "https://<Control Center host and port>/demo/testing/48/report_builder/",
  "start_time": "2017-09-03T20:04:27.000000Z",
  "status": "passed",
  "steps": [
    {
      "description": "",
      "end_time": "2017-09-03T20:05:03.000000Z",
      "id": 48,
      "name": "Test from template",
      "start_time": "2017-09-03T20:04:27.000000Z",
      "status": "passed",
      "status_message": "",
      "tasks": [
        {
          "name": "",
          "streams": [
            {
              "gui_url": "https://10.0.157.46/dev/results/72/rrd_graph/?
```

```
start=1526898065&end=1526898965",
          "id": 72,
          "is_owner": true,
          "metrics": [
            [
              "2017-09-03T20:04:27",
              "2017-09-03T20:04:27",
              4.4356,
              1.7191,
              9.9563,
              23.5623,
              21.8432,
              391,
              21.09,
              93,
              66,
              null,
              null,
              null,
              null,
              48.07,
              212,
              189,
              null,
              null,
              null,
              null,
              100,
              100,
              null,
              null,
              0,
              null,
              null
            ],
            [
              "2017-09-03T20:04:27",
              "2017-09-03T20:04:28",
              <metrics omitted>
            ],
            <remaining metrics omitted>
          ],
          "metrics_avg": {
```

```
      "davg": 10.167887,
      "davg_far": null,
      "davg_near": null,
      "dmax": 24.934801,
      "dmax_far": null,
      "dmax_near": null,
      "dmin": 1.143761,
      "dmin_far": null,
      "dmin_near": null,
      "dv": 23.791039,
      "dv_far": null,
      "dv_near": null,
      "end_time": "2018-05-21T02:01:40",
      "es": 1,
      "es_delay": null,
      "es_dscp": 0,
      "es_dv": null,
      "es_loss": 1,
      "es_ses": null,
      "loss_far": 18.584512,
      "loss_near": 51.791573,
      "lost_far": 81.910714,
      "lost_near": 228.285714,
      "miso_far": 60.5,
      "miso_near": 206.267857,
      "rate": 4.507413,
      "recv": 397.339286,
      "sla": 0,
      "sla_status": "unknown",
      "start_time": "2018-05-21T02:00:40",
      "uas": null
    },
    "metrics_headers": [
      "start_time",
      "end_time",
      "rate",
      "dmin",
      "davg",
      "dmax",
      "dv",
      "recv",
      "loss_far",
      "lost_far",
```

```
              "miso_far",
              "dmin_far",
              "davg_far",
              "dmax_far",
              "dv_far",
              "loss_near",
              "lost_near",
              "miso_near",
              "dmin_near",
              "davg_near",
              "dmax_near",
              "dv_near",
              "es",
              "es_loss",
              "es_delay",
              "es_dv",
              "es_dscp",
              "es_ses",
              "uas"
            ],
            "metrics_headers_display": [
              "Start time",
              "End time",
              "Rate (Mbit/s)",
              "Min round-trip delay (ms)",
              "Average round-trip delay (ms)",
              "Max round-trip delay (ms)",
              "Average round-trip DV (ms)",
              "Received packets",
              "Far-end loss (%)",
              "Far-end lost packets",
              "Far-end misorders",
              "Min far-end delay (ms)",
              "Average far-end delay (ms)",
              "Max far-end delay (ms)",
              "Far-end DV (ms)",
              "Near-end loss (%)",
              "Near-end lost packets",
              "Near-end misorders",
              "Min near-end delay (ms)",
              "Average near-end delay (ms)",
              "Max near-end delay (ms)",
              "Near-end DV (ms)",
```

```
                    "ES (%)",
                    "ES loss (%)",
                    "ES delay (%)",
                    "ES DV (%)",
                    "ES DSCP (%)",
                    "SES (%)",
                    "Unavailable seconds (%)"
                ],
                "reflector": 18,
                "sender": {
                  "ip_version": 4,
                  "name": "eth1",
                  "preferred_ip": null,
                  "test_agent": 1,
                  "test_agent_name": "VTA1"
                }
              }
            ],
            "task_type": "twamp",
          }
        ]
      }
    ]
  }
}
```

Example 2: DSCP Remapping Test

A DSCP remapping test is one that does not produce continuous metrics, but rather a single set of results at the end. It cannot run concurrently with anything else. The format of the output for this test is indicated below. (The Python code for retrieving the test results is the same except for the test ID.)

```
{
  "description": "",
  "end_time": "2019-01-04T07:19:53.000000Z",
  "gui_url": "https://10.0.157.46/dev/testing/154/",
  "id": 154,
  "name": "DSCP",
  "report_url": "https://10.0.157.46/dev/testing/154/report_builder/",
  "start_time": "2019-01-04T07:19:49.000000Z",
```

```
"status": "passed",
"steps": [
  {
    "description": "",
    "end_time": "2019-01-04T07:19:52.000000Z",
    "id": 154,
    "name": "Step 1",
    "start_time": "2019-01-04T07:19:49.000000Z",
    "status": "passed",
    "status_message": "",
    "step_type": "exclusive_task_container",
    "tasks": [
      {
        "end_time": "2019-01-04T07:19:52",
        "log": [
          {
            "level": "debug",
            "message": "Starting script",
            "time": "2019-01-04T07:19:49"
          },
          {
            "level": "debug",
            "message": "Sending from VTA1:eth1 (IPv4) to VTA2:eth1 (IPv4)",
            "time": "2019-01-04T07:19:49"
          },
          {
            "level": "debug",
            "message": "Sending packets",
            "time": "2019-01-04T07:19:50"
          },
          {
            "level": "info",
            "message": "Passed: ",
            "time": "2019-01-04T07:19:52"
          }
        ],
        "name": "Step 1",
        "results": {
          "create": [
            {
              "id": 269,
              "type": "Table"
            }
```

```json
        ],
        "update": {
          "269": {
            "columns": [
              {
                "name": "Sent DSCP",
                "type": "string"
              },
              {
                "name": "Received DSCP",
                "type": "string"
              },
              {
                "name": "Test result",
                "type": "string"
              }
            ],
            "rows": [
              [
                "cs0 (0)",
                "cs0 (0)",
                "Passed"
              ],
              [
                "cs1 (8)",
                "cs1 (8)",
                "Passed"
              ],
              [
                "af11 (10)",
                "af11 (10)",
                "Passed"
              ],
              <... (DSCPs omitted)>
              [
                "cs7 (56)",
                "cs7 (56)",
                "Passed"
              ]
            ],
            "title": "DSCP remapping results"
          }
        }
```

```
      },
      "start_time": "2019-01-04T07:19:49",
      "status": "passed",
      "status_message": "",
      "task_type": "dscp_remapping"
    }
  ]
 }
]
}
```

Generating a PDF Report on a Test

You can generate a PDF report on a test directly from the REST API. The report has the same format as that generated from the Control Center GUI.

By default, the report covers the last 15 minutes of the test. You can specify a different time interval by including the `start` and `end` parameters in a query string at the end of the URL. The time is given in UTC (ISO 8601) as specified in IETF RFC 3339.

In addition, the following options can be included in the query string:

- `worst_num`: For each task in a test, you can specify how many measurement results to show, ranked by the number of errored seconds with the worst on top. The scope of a measurement result is task-dependent; to give one example, for HTTP it is the result obtained for one client. The default number is 30.

- `graphs`: Include graphs in the report.

Example:

```
# Include graphs
graphs = 'true'

# Request settings
url = '%s/accounts/%s/tests/%s/pdf_report?graphs=%s' % (args.ncc_url, args.account,
args.test_id, graphs)

# Get test
response = requests.get(url=url, headers={'API-Token': args.token})
```

```
print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Tags

Tags defined in Paragon Active Assurance can be applied to:

- monitors

- monitor templates

- Test Agents

- TWAMP reflectors.

For example, you can tag a monitor with the same tag as a subset of Test Agents that are going to run the monitor. This feature is particularly helpful if you have a large number of monitors and templates defined.

If you have set up an alarm with SNMP traps for a monitor, then the SNMP traps will be assigned the same tags as the monitor, if any.

Creating Tags

Below we show how to create a tag with name and color as defined by the JSON data.

```
# Request settings
url = '%s/accounts/%s/tags/' % (args.ncc_url, args.account)
test_agent_url = '%s/accounts/%s/test_agents/2/' % (args.ncc_url, args.account)

json_data = json.dumps({
  'name': 'Tag',
  'color': '#000'
})

# Create tag
response = requests.post(url=url, data=json_data, headers={
  'API-Token': args.token,
  'Accept': 'application/json; indent=4',
  'Content-Type': 'application/json',
})

print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Assigning a Tag

Here is how to assign the tags found under `tags` to the items found under `resources`. In this case, a single tag is assigned to a sole Test Agent.

The other resource types are: `monitor`, `monitor_template`, `test_template`, `twamp_reflector`.

```
# Request settings
url = '%s/accounts/%s/tags/assign/' % (args.ncc_url, args.account)

json_data = json.dumps({
  'resources': [
      {
          'type': 'test_agent', 'ids': [2]
      }
```

```
    ],
    'tags': [
        10
    ],
})


# Assign tags
response = requests.post(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})


print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Updating a Tag

This code example shows how to update the label and color of a tag. The tags remains assigned to the same resources as before.

```
# Request settings
url = '%s/accounts/%s/tags/%s/' % (args.ncc_url, args.account, args.tag_id)

json_data = json.dumps({
    'name': 'New tag name',
    'color': '#eee',
})


# Update tag
response = requests.put(url=url, data=json_data, headers={
    'API-Token': args.token,
    'Accept': 'application/json; indent=4',
    'Content-Type': 'application/json',
})


print 'Status code: %s' % response.status_code
print json.dumps(response.json(), indent=4)
```

Unassigning a Tag

Below we unassign the tag "1" from the Test Agent with id = 2.

```python
# Request settings
url = '%s/accounts/%s/tags/unassign/' % (args.ncc_url, args.account)

json_data = json.dumps({
'resources': [
    {
        'type': 'test_agent', 'ids': [2]
    }
],
'tags': [
    1
]
})


# Unassign tag
response = requests.post(url=url, data=json_data, headers={
'API-Token': args.token,
'Accept': 'application/json; indent=4',
'Content-Type': 'application/json',
})

print 'Status code: %s' % response.status_code
if response.status_code != NO_CONTENT:
  print json.dumps(response.json(), indent=4)
```

Deleting a Tag

In order to delete a tag altogether from Control Center, the following code is used.

```python
# Request settings
url = '%s/accounts/%s/tags/%s/' % (args.ncc_url, args.account, args.tag_id)

# Delete tag
response = requests.delete(url=url, headers={'API-Token': args.token})
```

```
print 'Status code: %s' % response.status_code
# delete does not return any content
if response.status_code != NO_CONTENT:
    print json.dumps(response.json(), indent=4)
```

Troubleshooting

IN THIS SECTION

-

Troubleshooting

Problem: Executing actions through the REST API browser fails
Solution:

- Start by verifying that the curl example found in the API browser works. Note that you might have to run it using the `--insecure` flag, namely if you are using the default snakeoil SSL certificates; see the section on SSL certificate configuration in the Installation Guide, chapter Service Configuration.

- If the curl example works fine, the reason that the actions fail is that you are not surfing to the web page configured as `SITE_URL` in `/etc/netrounds/netrounds.conf`. That exact same host name must be used; otherwise your browser will consider your action as a cross-site scripting attempt. Example:

  - Suppose that `SITE_URL=https://mynetrounds.local` and that this resolves to the IP 192.168.1.123.

  - Correct URL to navigate to: `https://mynetrounds.local/rest/`

  - Wrong URL: `https://192.168.1.123/rest/`

  - Wrong URL: `https://other.host.resolving.to.same.ip/rest/`