ntc$_{r}osetta_{c}onf Documentation$

Release 0.0.1

David Barroso

Jul 31, 2019

Contents

`ntc-rosetta-conf` is a RESTCONF interface for ntc-rosetta. This RESTCONF interface allows you to manipulate a candidate and running databases using supported models by `ntc-rosetta` and it also exposes a few RPC endpoints to translate, parse and merge configurations.

CHAPTER 1

Contents

1.1 Intro

`ntc-rosetta-conf` is a RESTCONF interface for ntc-rosetta. This RESTCONF interface allows you to manipulate a candidate and running databases using supported models by `ntc-rosetta` and it also exposes a few RPC endpoints to translate, parse and merge configurations.

1.1.1 Installing

This python package is available through pip so you can install with the following command:

```
pip install ntc-rosetta-conf
```

1.1.2 Starting the restconf interface

```
$ ntc-rosetta-conf serve \
  --datamodel openconfig \
  --pid-file /tmp/ntc-rosetta-conf-demo.pid \
  --log-level debug \
  --data-file data.json \
  --port 8443 \
  --ssl-crt pki_auto_generated_dir/server_ca/certs/rtr00.lab.local.crt \
  --ssl-key pki_auto_generated_dir/server_ca/keys/rtr00.lab.local.key \
  --ca-crt pki_auto_generated_dir/client_ca/certs/client_ca.crt
```

1.1.3 Consuming the restconf interface

```
$ curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    https://localhost:8443/restconf/data/openconfig-interfaces:interfaces
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "description": "an interface description",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth1",
                "config": {
                    "name": "eth1",
                    "description": "another interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
$ curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @docs/tutorials/4_translate/translate_running.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native"

interface eth0
   description an interface description
   exit
!
interface eth1
   description another interface
   exit
!
```

1.2 CLI

To start `ntc-rosetta-cpnf` you can use its command line:

```
$ ntc-rosetta-conf serve --help
Usage: ntc-rosetta-conf serve [OPTIONS]

Options:
  --datamodel [openconfig|ntc]    Datamodel to use
  --pid-file TEXT                 PID file
  --log-level [debug|info|warning|error]
                                  Logging level
  --data-file TEXT                Path to json file to load data from and save
                                  on commit
  --listen-on-localhost-only      Listen on localhost only
  --port INTEGER                  Port to listen to
```

(continues on next page)

(continued from previous page)

```
  --ssl-crt TEXT                SSL Certificate for the webserver
  --ssl-key TEXT                Private key for the webserver
  --ca-crt TEXT                 CA certificate used to sign client
                                certificates
  --help                        Show this message and exit.
```

1.3 Tutorials

1.3.1 Jetconf/Restconf basics

In this demo we are going to see how jetconf works from an end user perspective and some of its capabilities.

Starting things up

Run:

```
make start-dev-containers
```

Now, let's export some environment variables we need:

[1]:
```
export USER_CERT=../../tests/certs/test_user_curl.pem
export BASE_URL=https://ntc-rosetta-conf:8443
```

Basics

Jetconf adheres to RFC8040, with the exception of having support for candidate/running configuration database (which we will explore later).

Methods supported

Path	Child type	Method	Use
container	list	POST	Creates a list element (doesn't fail if exists, but will fail on commit)
container	leaf	POST	Creates a leaf (fails if exists)
container	container	POST	Creates container (fails if exists)
container	N/A	PUT	Replaces the container object targetted in the path
container	N/A	DELETE	Deletes container targetted in the path
leaf	N/A	PUT	Replaces existing leaf
leaf	N/A	DELETE	Deletes existing leaf
list	N/A	PUT	Replaces existing element
list	N/A	DELETE	Deletes existing element

It's important to understand the difference between `path` and `child` in this context, which only makes sense when the path points to a container and the method is a POST. The path is basically the URL you are querying, which might end in a container, list element or leaf. For instance:

- container - `/interfaces` or `/interfaces/interface=eth0`
- list - `/interfaces/interfaces`

- leaf - `/interfaces/interfaces=eth0/name`

The child type is basically the object type you are POSTing. When working with containers the object can either be a child object, which can be of either type, or iselt. We will see this more clearly as we progress with examples for each method.

Adding operations and hooks

Finally, users can define their own operations. We will see examples of that in a later notebook when we explore napalm's integration. Hooks can also be defined; hooks are action that can be attached to actions like "creating an interface", "removing a vlan", "updating a particular object or field", etc. . .

Examples

Now, let's explore the methods supported to deal with objects by example.

Here we are targing a container (`openconfig-interfaces:interfaces`) but the object inside will contain a list element (`openconfig-interfaces:interface: {...}`). You can use this to create new list elements:

[2]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```


[3]:
```
cat 1_jetconf_basics/add_interface_eth0.json
```


```
{
        "openconfig-interfaces:interface": {
                "name": "eth0",
                "config": {
                        "name": "eth0",
                        "description": "a test interface",
                        "type": "iana-if-type:ethernetCsmacd"
                }
        }
}
```


[4]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @1_jetconf_basics/add_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[5]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
```


(continues on next page)

(continued from previous page)

```
                "config": {
                    "name": "eth0",
                    "description": "a test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```

Now we are targetting a different container but the object inside is a leaf. You can use this to create new leaves in a container. For instance, let's add the mtu field to the configuration, which is missing:

[6]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "description": "a test interface",
        "type": "iana-if-type:ethernetCsmacd"
    }
}
```

[7]:
```
cat 1_jetconf_basics/add_mtu.json
```

```
{
        "openconfig-interfaces:mtu": 9000
}
```

[8]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @1_jetconf_basics/add_mtu.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

[9]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "description": "a test interface",
        "type": "iana-if-type:ethernetCsmacd",
        "mtu": 9000
    }
}
```

You can POST a container object in a container object to create it. For instance, let's create the `hold-time` container under the interface itself, which is missing:

[10]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/hold-time
```

```
{
    "ietf-restconf:errors": {
        "error": [
            {
                "error-type": "protocol",
                "error-tag": "invalid-value",
                "error-path": "/openconfig-interfaces:interfaces/interface/0",
                "error-message": "NonexistentInstance: member 'hold-time'"
            }
        ]
    }
}
```


[11]:
```
cat 1_jetconf_basics/add_hold_time.json
```


```
{
        "openconfig-interfaces:hold-time": {}
}
```


[12]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @1_jetconf_basics/add_hold_time.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0
```

[13]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/hold-time
```


```
{
    "openconfig-interfaces:hold-time": {}
}
```


Note: In this example we added an empty container, but you can add an already populated one

When doing a PUT in a container, the object in the paylod is itself. You can use this to do replace the container:

[14]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```


```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "description": "a test interface",
        "type": "iana-if-type:ethernetCsmacd",
        "mtu": 9000
    }
}
```


[15]:
```
cat 1_jetconf_basics/change_config.json
```


```
{
        "openconfig-interfaces:config": {
                "name": "eth0",
                "description": "a new interface description",
                "type": "iana-if-type:ethernetCsmacd"
        }
}
```

[16]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @1_jetconf_basics/change_config.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

[17]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```


```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "description": "a new interface description",
        "type": "iana-if-type:ethernetCsmacd"
    }
}
```


Use it to delete a container:

[18]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0
```


```
{
    "openconfig-interfaces:interface": [
        {
            "name": "eth0",
            "hold-time": {},
            "config": {
                "name": "eth0",
                "description": "a new interface description",
                "type": "iana-if-type:ethernetCsmacd"
            }
        }
    ]
}
```


[19]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X DELETE \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/hold-time
```

[20]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0
```


```
{
    "openconfig-interfaces:interface": [
        {
            "name": "eth0",
            "config": {
                "name": "eth0",
                "description": "a new interface description",
                "type": "iana-if-type:ethernetCsmacd"
            }
        }
    ]
}
```


You can use it to change a configuration element:

[21]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config/
↪description
```

```
{
    "openconfig-interfaces:description": "a new interface description"
}
```

[22]:
```
cat 1_jetconf_basics/change_description.json
```

```
{
    "openconfig-interfaces:description": "yet another changed description"
}
```

[23]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @1_jetconf_basics/change_description.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config/
↪description
```

[24]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config/
↪description
```

```
{
    "openconfig-interfaces:description": "yet another changed description"
}
```

This is useful to remove configuration elements:

[25]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "type": "iana-if-type:ethernetCsmacd",
        "description": "yet another changed description"
    }
}
```

[26]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X DELETE \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config/
↪description
```

[27]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config
```

```
{
    "openconfig-interfaces:config": {
        "name": "eth0",
        "type": "iana-if-type:ethernetCsmacd"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

You can use this to replace the entire list:

[28]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface
```

```
{
    "openconfig-interfaces:interface": [
        {
            "name": "eth0",
            "config": {
                "name": "eth0",
                "type": "iana-if-type:ethernetCsmacd"
            }
        }
    ]
}
```

[29]:
```
cat 1_jetconf_basics/replace_interfaces.json
```

```
{
        "openconfig-interfaces:interface": [
                {
                        "name": "eth1",
                        "config": {
                                "name": "eth1",
                                "type": "iana-if-type:ethernetCsmacd"
                        }
                },
                {
                        "name": "eth2",
                        "config": {
                                "name": "eth2",
                                "type": "iana-if-type:ethernetCsmacd"
                        }
                }
        ]
}
```

[30]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @1_jetconf_basics/replace_interfaces.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface
```

[31]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface
```

```
{
    "openconfig-interfaces:interface": [
        {
            "name": "eth1",
            "config": {
                "name": "eth1",
```

(continues on next page)

(continued from previous page)


```
                "type": "iana-if-type:ethernetCsmacd"
            }
        },
        {
            "name": "eth2",
            "config": {
                "name": "eth2",
                "type": "iana-if-type:ethernetCsmacd"
            }
        }
    ]
}
```


You can use this to delete the entire list (although you will need to reinitialize it with a POST, much better to replace the list with an empty one instead):

[32]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth1",
                "config": {
                    "name": "eth1",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth2",
                "config": {
                    "name": "eth2",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```


[33]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X DELETE \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface
```

[34]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {}
}
```


[35]:
```
# ignore me, this discards the changes so the notebook can be rerun
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-reset
```

```
{
    "status": "OK"
}
```

1.3.2 Candidate/Running config database

In this demo we are going to see how jetconf let's you manage the

Starting things up

Run:

```
make start-dev-containers
```

Now, let's export some environment variables we need:

[1]:
```
export USER_CERT=../../tests/certs/test_user_curl.pem
export BASE_URL=https://ntc-rosetta-conf:8443
```

Basics

Even though it's not part of the specification, jetconf implements a netconf-like workflow where changes are made against a candidate database and commited in a single transaction into the running database. There are a few notes to be made here.

Even though changes are always made against the candidate database, the `GET` method can be used against the running configuration. To support this the URL target will be slightly different:

- Use `$BASE_URL/restconf/data/...` to target the candidate database.
- Use `$BASE_URL/restconf_running/data/...` to target the running database.

To operate the candidata database, jetconf supports the following operational endpoints:

Path	Method	Use
/restconf/operations/jetconf:conf-reset	POST	Discard candidate configuration
/restconf/operations/jetconf:conf-commit	POST	commit candidate configuration
/restconf/operations/jetconf:get-schema-digest	POST	retrieve supported schema
/restconf/operations/jetconf:conf-status	POST	retrieve status of the configuration

Let's see a few examples, let's start by adding an interface:

[2]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```

[3]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @2_candidate_config/add_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

Now, let's verify the candidate database:

[4]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "description": "a test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```


The changes are there, let's look at the running database:

[5]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```


The changes are not yet present there, let's do a couple of other changes; let's add another interface:

[6]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @2_candidate_config/add_interface_eth1.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

Now, let's verify the different configuration databases:

[7]:
```
# candidate:devices
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
```


(continues on next page)

(continued from previous page)


```
                    "description": "a test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth1",
                "config": {
                    "name": "eth1",
                    "description": "another test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```


[8]:
```
# running:devices
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```


Pretty much what we expected. Now we can do three things:

- `/restconf/operations/jetconf:conf-reset` - Discard the changes
- `/restconf/operations/jetconf:conf-commit` - Commit the changes
- `restconf/operations/jetconf:conf-status` - Verify the status of the configuration

Let's start by verifying the status (should report that there is a transaction opened), commit the changes and then verify the status again (should report the transaction is not closed):

[9]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-status
```


```
{
    "status": "OK",
    "transaction-opened": true
}
```


[10]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "status": "OK",
    "conf-changed": true
}
```


[11]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-status
```

```
{
    "status": "OK",
    "transaction-opened": false
}
```


Now, let's verify the running config:

[12]:
```
# running:interfaces
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "description": "a test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth1",
                "config": {
                    "name": "eth1",
                    "description": "another test interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```


[13]:
```
# ignore me, this deletes the data so the notebook can be rerun
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @../../tests/data/interfaces_empty.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "status": "OK",
    "conf-changed": true
}
```


1.3.3 Errors

In this demo we are going to see how jetconf deals with errors in the data.

Starting things up

Run:

```
make build_container
docker run \
    -p 8443:8443 \
    -e ONLINE_MODE=0 \
    rosetta -c /rosetta/tests/config.yaml
```

Now, let's export some environment variables we need:

[1]:
```
export USER_CERT=../../tests/certs/test_user_curl.pem
export BASE_URL=https://ntc-rosetta-conf:8443
```

Types of errors

There are three type of errors you can encounter:

1. Simple schema validation errors - For instance, a leaf being assigned a wrong type. This are performed on each operation.
2. Complex schema validation errors - Wrong identity value or value. This are performed on commit only.
3. Semantic errors - A duplicated key in a list, a missing leaf-ref, etc. This are performed on commit only.

Simple schema validation errors

We are going to start trying to create a device with a number as name:

[2]:
```
cat 3_errors/add_interface_eth0_bad_description.json
```

```
{
        "openconfig-interfaces:interface": {
                "name": "eth0",
                "config": {
                        "name": "eth0",
                        "description": 0,
                        "type": "iana-if-type:ethernetCsmacd"
                }
        }
}
```

[3]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @3_errors/add_interface_eth0_bad_description.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

```
{
    "ietf-restconf:errors": {
        "error": [
            {
                "error-type": "protocol",
                "error-tag": "invalid-value",
                "error-path": "/openconfig-interfaces:interfaces/interface/1/config/
↪description",
                "error-message": "RawTypeError: expected string value"
            }
        ]
```

(continues on next page)

(continued from previous page)

```
    }
}
```

[4]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-reset
```


```
{
    "status": "OK"
}
```


Complex schema validation errors

Now, let's try creating an interface with a very large MTU:

[5]:
```
cat 3_errors/add_interface_eth0_large_mtu.json
```


```
{
        "openconfig-interfaces:interface": {
                "name": "eth0",
                "config": {
                        "name": "eth0",
                        "type": "iana-if-type:ethernetCsmacd",
                        "mtu": 5465464564564645
                }
        }
}
```


[6]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @3_errors/add_interface_eth0_large_mtu.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[7]:
```
# previous command succeeded, however, the commit will fail
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "ietf-restconf:errors": {
        "error": [
            {
                "error-type": "protocol",
                "error-tag": "operation-failed",
                "error-app-tag": "invalid-type",
                "error-path": "/openconfig-interfaces:interfaces/interface/0/config/
↪mtu",
                "error-message": "YangTypeError: expected uint16"
            }
        ]
    }
}
```

[8]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-reset
```


```
{
    "status": "OK"
}
```


1.3.4 Semantic errors

Now we are going to try to create a two devices with the same name. On commit it will complain the key is not unique (it doesn't matter if one of the element was previously commited or not):

[9]:
```
cat 3_errors/add_interface_eth0.json
```


```
{
        "openconfig-interfaces:interface": {
                "name": "eth0",
                "config": {
                        "name": "eth0",
                        "type": "iana-if-type:ethernetCsmacd"
                }
        }
}
```


[10]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @3_errors/add_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[11]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @3_errors/add_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[12]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
```


(continues on next page)

(continued from previous page)


```
        ]
    }
}
```


[13]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "ietf-restconf:errors": {
        "error": [
            {
                "error-type": "protocol",
                "error-tag": "invalid-value",
                "error-app-tag": "non-unique-key",
                "error-path": "/openconfig-interfaces:interfaces/interface",
                "error-message": "SemanticError: 'eth0'"
            }
        ]
    }
}
```


[14]:
```
# ignore me, this discards the changes so the notebook can be rerun
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-reset
```


```
{
    "status": "OK"
}
```


1.3.5 Using rosetta-conf to generate native configuration

Starting things up

Run:

```
make start-dev-containers
```

Now, let's export some environment variables we need:

[1]:
```
export USER_CERT=../../tests/certs/test_user_curl.pem
export BASE_URL=https://ntc-rosetta-conf:8443
```

Configuration

First we need to configure the platform of the device, this configuration is part of the model.

[2]:
```
cat 4_translate/configuration.json
```


```
{
    "ntc-rosetta-conf:device": {
        "config": {
```


(continues on next page)

(continued from previous page)

```
            "platform": "ios"
        }
    }
}
```

[3]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/configuration.json \
    $BASE_URL/restconf/data/
```

[4]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```

```
{
    "status": "OK",
    "conf-changed": true
}
```

[5]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/ntc-rosetta-conf:device
```

```
{
    "ntc-rosetta-conf:device": {
        "config": {
            "platform": "ntc-rosetta-conf:ios"
        }
    }
}
```

Translating

The first thing we can do is translate the data into native format. Let's start by adding a couple of interfaces:

[6]:
```
cat 4_translate/add_interface_eth0.json
```

```
{
    "openconfig-interfaces:interface": {
        "name": "eth0",
        "config": {
            "name": "eth0",
            "description": "an interface description",
            "type": "iana-if-type:ethernetCsmacd"
        }
    }
}
```

[7]:
```
cat 4_translate/add_interface_eth1.json
```

```
{
    "openconfig-interfaces:interface": {
        "name": "eth1",
        "config": {
            "name": "eth1",
```

(continues on next page)

(continued from previous page)


```
            "description": "another interface",
            "type": "iana-if-type:ethernetCsmacd"
        }
    }
}
```


[8]:

```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/add_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[9]:

```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/add_interface_eth1.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

[10]:

```
# candidate database
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "eth0",
                "config": {
                    "name": "eth0",
                    "description": "an interface description",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            },
            {
                "name": "eth1",
                "config": {
                    "name": "eth1",
                    "description": "another interface",
                    "type": "iana-if-type:ethernetCsmacd"
                }
            }
        ]
    }
}
```


[11]:

```
# running database
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```


Now we can translate either the “candidate” or “running” databases into native configuration:

[12]:
```
cat 4_translate/translate_candidate.json
```

```
{
    "ntc-rosetta-conf:input": {
        "database": "candidate"
    }
}
```

[13]:
```
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_candidate.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native"
```

```
interface eth0
   description an interface description
   exit
!
interface eth1
   description another interface
   exit
!
```

[14]:
```
# running was empty so no data there
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_running.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native"
```

[15]:
```
# now we are happy with it we can commit the configuration
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```

```
{
    "status": "OK",
    "conf-changed": true
}
```

More changes

We can now apply more changes to the candidate database and keep comparing the candidate and running databases of the device using their native representation:

[16]:
```
cat 4_translate/change_interface_eth0.json
```

```
{
    "openconfig-interfaces:description": "a changed description"
}
```

[17]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @4_translate/change_interface_eth0.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces/interface=eth0/config/
↪description
```

[18]:
```
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_candidate.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native"
```

```
interface eth0
   description a changed description
   exit
!
interface eth1
   description another interface
   exit
!
```

[19]:
```
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_running.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native"
```

```
interface eth0
   description an interface description
   exit
!
interface eth1
   description another interface
   exit
!
```

At this point you could grab both "native" results and diff them:

[20]:
```
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_candidate.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native" > /tmp/
↪candidate
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/translate_running.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:translate | jq -r ".native" > /tmp/
↪running
diff -W 100 --side-by-side /tmp/candidate /tmp/running || echo -n
```

```
interface eth0                                  interface eth0
   description a changed description          |    description an interface␣
↪description
   exit                                            exit
!                                               !
interface eth1                                  interface eth1
   description another interface                   description another interface
   exit                                            exit
!                                               !
```

Merging

Alterntatively, you can call the merge endpoint and get a list of commands that will make the configurations converge:

[21]:
```
curl -s --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @4_translate/merge.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:merge | jq -r ".native"
```

```
interface eth0
   description a changed description
   exit
!
```

[22]:
```
# ignore me, this deletes the data so the notebook can be rerun
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-reset
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @../../tests/data/interfaces_empty.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X DELETE \
    $BASE_URL/restconf/data/ntc-rosetta-conf:device
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "status": "OK"
}{
    "status": "OK",
    "conf-changed": true
}
```


1.3.6 Parsing

Starting things up

Run:

```
make start-dev-containers
```

Now, let's export some environment variables we need:

[1]:
```
export USER_CERT=../../tests/certs/test_user_curl.pem
export BASE_URL=https://ntc-rosetta-conf:8443
```

Configuration

First we need to configure the platform of the device, this configuration is part of the model.

[2]:
```
cat 5_parse/configuration.json
```

```
{
    "ntc-rosetta-conf:device": {
        "config": {
```

(continues on next page)

(continued from previous page)


```
            "platform": "ios"
        }
    }
}
```


[3]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @5_parse/configuration.json \
    $BASE_URL/restconf/data/
```

[4]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```


```
{
    "status": "OK",
    "conf-changed": true
}
```


[5]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/ntc-rosetta-conf:device
```


```
{
    "ntc-rosetta-conf:device": {
        "config": {
            "platform": "ntc-rosetta-conf:ios"
        }
    }
}
```


Parsing

Now let's see how we can parse native configuration, let's start by checking configuration is empty:

[6]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
     $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```


```
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```


Now we need to a json object with the following structure:

```
{
    "ntc-rosetta-conf:input": {
        "validate": true,
        "native": "string with configuration in native format (ios_style/xml/etc)"
    }
}
```

[7]:
```
cat 5_parse/parse.json
```

```
{
    "ntc-rosetta-conf:input": {
        "validate": true,
        "native": "interface GigabitEthernet0\n   description an interface␣
↪description\n   exit\n!\ninterface GigabitEthernet1\n   description another␣
↪interface\n   exit\n!\n"
    }
}
```

Now we call the RPC `ntc-rosetta-conf:parse`:

[8]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    -d @5_parse/parse.json \
    $BASE_URL/restconf/operations/ntc-rosetta-conf:parse
```

```
{
    "result": "ntc-rosetta-config:success"
}
```

Configuration is parsed and loaded into the running database:

[9]:
```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X GET \
    $BASE_URL/restconf_running/data/openconfig-interfaces:interfaces
```

```
{
    "openconfig-interfaces:interfaces": {
        "interface": [
            {
                "name": "GigabitEthernet0",
                "config": {
                    "name": "GigabitEthernet0",
                    "type": "iana-if-type:ethernetCsmacd",
                    "description": "an interface description",
                    "enabled": true
                }
            },
            {
                "name": "GigabitEthernet1",
                "config": {
                    "name": "GigabitEthernet1",
                    "type": "iana-if-type:ethernetCsmacd",
                    "description": "another interface",
                    "enabled": true
                }
            }
        ]
    }
}
```

[10]:
```
# ignore me, this deletes the data so the notebook can be rerun
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X PUT \
    -d @../../tests/data/interfaces_empty.json \
    $BASE_URL/restconf/data/openconfig-interfaces:interfaces
```

(continues on next page)

(continued from previous page)

```
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X DELETE \
    $BASE_URL/restconf/data/ntc-rosetta-conf:device
curl --http2 -k --cert-type PEM -E $USER_CERT \
    -X POST \
    $BASE_URL/restconf/operations/jetconf:conf-commit
```

```
{
    "status": "OK",
    "conf-changed": true
}
```

1.3.7 Mutual TLS authentication

`ntc-rosetta-conf` utilizes mutual TLS (mTLS) for authentication purposes. In this document we are going to quickly see how mutual TLS authentication works in the context of `ntc-rosetta-conf`

Intro

This introduction mTLS doesn't aim to be an in-depth explanation of how it works, there are plenty of resources out there that aim to do that. Here we are going to just introduce the basic concepts so we have a basic understanding to operate `ntc-rosetta-conf`

The basic idea of mTLS is that both the client and the server will validate the other party by looking at their certificate. This is done by checking that the certificate authority that signed the other party is (a) the one we expect, (b) cert hasn't expired and (c) cert hasn't been revoked. Let's see an illustration:

```
+---------------------------------------------+ +--------------------------------------
↪------+
|                client CA                  | |                server CA              ␣
↪       |
|(certificate authority to sign client certs)| |(certificate authority to sign server␣
↪certs)|
+---------------------------------------------+ +--------------------------------------
↪------+
         _____/       |       \_____                     _____/       |       \_____
        |             |             |                   |             |             |
    +----------+ +----------+ +----------+          +----------+ +----------+ +----------+
    | client1 | | client2 | | client3 |          | server1 | | server2 | | server3 |
    +----------+ +----------+ +----------+          +----------+ +----------+ +----------+


                              mTLS Authentication Overview
       ==============================================================================
Client| 1. Start tls handshake                                                     ␣
↪|Server
      | ---------------------------------------------------------------------------> |
      |                    A. Send server certificate and request one to the client |
      | <--------------------------------------------------------------------------- |
      | 2. Validate server cert is valid by checking it was signed by "server CA"   |
      | 3. Send client certificate                                                   |
      | ---------------------------------------------------------------------------> |
      |   B. Validate client cert is valid by checking it was signed by "client CA" |
       ==============================================================================
```

Note: This is gross oversimplification of how TLS and mTLS work. Refer to more in-depth guides/documentation if you are interested in knowing how it works.

Tutorial

Let's test the theory. Before we start, if you want to follow this tutorial you need:

1. Docker. Alternatively, you can install easypki and use it outside the docker container.
2. Create folder `pki_auto_generated_dir` in your current path: `mkdir pki_auto_generated_dir`

Clients

First, we need to provide our users with client certificates, so let's create a couple.

Creating the client CA

To generate client certificates we are going to need a CA. Ideally, such CA would be an intermediate CA signed by a valid CA but as this is dev we are going to just create a root CA:

```
$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --ca \
      --filename client_ca \
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      ntc-rosetta-conf-client-authority
```

Now, you can find under `pki_auto_generated_dir/client_ca/{certs,keys}` our CA cert and key:

```
$ ls pki_auto_generated_dir/client_ca/{certs,keys}
pki_auto_generated_dir/client_ca/certs:
client_ca.crt

pki_auto_generated_dir/client_ca/keys:
client_ca.key
```

Creating client certificates

Now that we have a CA to generate client certificates let's create a couple for Jane and John:

```
$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --client \
      --ca-name "client_ca" \
```

(continues on next page)

(continued from previous page)

```
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      --email "jane@networktocode.com" \
      jane@networktocode.com

$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --client \
      --ca-name "client_ca" \
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      --email "john@networktocode.com" \
      john@networktocode.com
```

Generated certs and keys will be under the same `client_ca` folder from before:

```
$ls pki_auto_generated_dir/client_ca/{certs,keys}
pki_auto_generated_dir/client_ca/certs:
client_ca.crt  jane@networktocode.com.crt  john@networktocode.com.crt

pki_auto_generated_dir/client_ca/keys:
client_ca.key  jane@networktocode.com.key  john@networktocode.com.key
```

Server certificates

Now we are going to need a server certificate for each instance of `ntc-rosetta-conf`.

Server CA

As with the client CA ideally you'd use an intermediate CA signed by a valid CA but, as this is dev, we are going to create our own root CA:

```
$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --ca \
      --filename server_ca \
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      ntc-rosetta-conf-server-authority
```

This time, we will find certs and keys under `pki_auto_generated_dir/server_ca/{certs,keys}`:

```
$ ls pki_auto_generated_dir/server_ca/{certs,keys}
pki_auto_generated_dir/server_ca/certs:
server_ca.crt

pki_auto_generated_dir/server_ca/keys:
server_ca.key
```

Creating server certificates

Now it's time to generate the certificates for our `ntc-rosetta-conf` instances:

```
$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --ca-name "server_ca" \
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      rtr00.lab.local

$ docker run -v $PWD:/certs -w /certs \
   creatdevsolutions/easypki create \
      --ca-name "server_ca" \
      --expire 365 \
      --private-key-size 2048 \
      --organization "NTC" \
      --organizational-unit "Eng" \
      --locality "Stockholm" \
      --country "Sweden" \
      --province "Stockholm" \
      rtr01.lab.local
```

Certs and keys will be under the same path as the server CA:

```
$ ls pki_auto_generated_dir/server_ca/{certs,keys}
pki_auto_generated_dir/server_ca/certs:
rtr00.lab.local.crt  rtr01.lab.local.crt  server_ca.crt

pki_auto_generated_dir/server_ca/keys:
rtr00.lab.local.key  rtr01.lab.local.key  server_ca.key
```

Starting ntc-rosetta-conf

Now that everything is ready, let's start `ntc-rosetta-conf`. Note the options for `--ssl-crt` (server cert), `--ssl-key` (server key) and `--ca-crt` (client CA cert):

```
$ ntc-rosetta-conf serve \
   --datamodel openconfig \
   --pid-file /tmp/ntc-rosetta-conf-demo.pid \
```

(continues on next page)

(continued from previous page)

```
   --log-level debug \
   --data-file data.json \
   --port 8443 \
   --ssl-crt pki_auto_generated_dir/server_ca/certs/rtr00.lab.local.crt \
   --ssl-key pki_auto_generated_dir/server_ca/keys/rtr00.lab.local.key \
   --ca-crt pki_auto_generated_dir/client_ca/certs/client_ca.crt
2019-07-17 11:54:59,599 INFO     NTC Rosetta Conf version TBD
2019-07-17 11:54:59,601 INFO     Using config:
GLOBAL:
  DATA_JSON_FILE: data.json
  LOGFILE: '-'
  LOG_DBG_MODULES:
  - '*'
  LOG_LEVEL: debug
  PERSISTENT_CHANGES: true
  PIDFILE: /tmp/ntc-rosetta-conf-demo.pid
  TIMEZONE: GMT
  VALIDATE_TRANSACTIONS: true
  YANG_LIB_DIR: asda
HTTP_SERVER:
  API_ROOT: /restconf
  API_ROOT_RUNNING: /restconf_running
  CA_CERT: pki_auto_generated_dir/client_ca/certs/client_ca.crt
  DBG_DISABLE_CERTS: false
  DOC_DEFAULT_NAME: index.html
  DOC_ROOT: doc-root
  LISTEN_LOCALHOST_ONLY: false
  PORT: 8443
  SERVER_NAME: jetconf-h2
  SERVER_SSL_CERT: pki_auto_generated_dir/server_ca/certs/rtr00.lab.local.crt
  SERVER_SSL_PRIVKEY: pki_auto_generated_dir/server_ca/keys/rtr00.lab.local.key
  UPLOAD_SIZE_LIMIT: 1
NACM:
  ALLOWED_USERS: []
  ENABLED: true

2019-07-17 11:55:00,571 INFO     Server started on ('::', 8443, 0, 0)
```

Client

Now we can use curl to query `ntc-rosetta-conf`. Let's start by trying to use curl without using any client cert:

```
$ curl \
      --cacert pki_auto_generated_dir/server_ca/certs/server_ca.crt \
      -X GET \
      https://rtr00.lab.local:8443/restconf/data/openconfig-interfaces:interfaces
curl: (56) OpenSSL SSL_read: SSL_ERROR_SYSCALL, errno 104
```

We got an SSL error. This is because the handshake failed as the server requested a client certificate. Let's try passing our client cert and key this time:

```
$ curl \
   --cacert pki_auto_generated_dir/server_ca/certs/server_ca.crt \
   --cert pki_auto_generated_dir/client_ca/certs/jane@networktocode.com.crt \
   --key pki_auto_generated_dir/client_ca/keys/jane@networktocode.com.key \
```

(continues on next page)

(continued from previous page)

```
    -X GET \
    https://rtr00.lab.local:8443/restconf/data/openconfig-interfaces:interfaces
{
    "openconfig-interfaces:interfaces": {
        "interface": []
    }
}
```

Now we managed to authenticate ourselves with the server.

Note: make sure that `rtr00.lab.local` resolves the correct IP. You can do that for the sake of testing by editing `/etc/hosts`.

Note: You probably noticed the line `--cacert pki_auto_generated_dir/server_ca/certs/server_ca.crt`. We need that option because we are using self-signed certificates.

1.4 Architecture

This document shows a reference architecture for `ntc-rosetta-conf`. Note that you are free to deploy it in any way that works for you though.

First thing to bear in mind when deploying `ntc-rosetta-conf` is that each instance represents a single device. This means that if you have 100 routers you will need 100 instances. This might sound a bit cumbersome but it has the advantage you limit your blast radius in case an instance fails for some reason and it also helps scaling out the solution by spreading the instances across many servers.

To avoid having to run all those instances manually, having to manage all the different ports to avoid collisions and having to remember which port belongs to which router, we recommend running this behind some dockerized solution and behind a load balancer.

A continuation you can see an example of such type of deployment using `docker-compose` and `haproxy`.

1.4.1 haproxy

Let's start looking at the configuration file:

```
global
    maxconn 2048
    ulimit-n 51200
    tune.ssl.default-dh-param 2048

defaults
    timeout connect 5000
    timeout client 50000
    timeout server 50000
    option http-server-close
    option httpclose
    mode http
    balance roundrobin
```

(continues on next page)

(continued from previous page)

```
frontend https-in
    mode http
    # listen on port 65443 and enable mTLS and http/2
    bind 0.0.0.0:65443 ssl crt /etc/haproxy/rosetta.pem ca-file /etc/haproxy/ca.pem␣
↪verify optional alpn h2

    # forward SSL headers to rosetta
    http-request set-header X-SSL                       %[ssl_fc]
    http-request set-header X-SSL-Client-Verify         %[ssl_c_verify]
    http-request set-header X-SSL-Client-DN             %{+Q}[ssl_c_s_dn]
    http-request set-header X-SSL-Client-CN             %{+Q}[ssl_c_s_dn(cn)]
    http-request set-header X-SSL-Issuer                %{+Q}[ssl_c_i_dn]
    http-request set-header X-SSL-Client-Not-Before     %{+Q}[ssl_c_notbefore]
    http-request set-header X-SSL-Client-Not-After      %{+Q}[ssl_c_notafter]

    # configure rules to forward requests to the different instances of rosetta
    use_backend rtr00 if { path -i -m beg /rtr00 }
    use_backend rtr01 if { path -i -m beg /rtr01 }


backend rtr00
    mode http

    # remove /rtr0x from the url
    reqrep ^([^\ ]*\ /)rtr00[/]?(.*)     \1\2
    server rtr00 172.21.33.100:8443 proto h2

backend rtr01
    mode http

    # remove /rtr0x from the url
    reqrep ^([^\ ]*\ /)rtr01[/]?(.*)     \1\2
    server rtr00 172.21.33.101:8443 proto h2
```

Let's try to summarize what's going on:

1. First we have some globals and defaults, we can ignore those.
2. Next we define a `frontend`, this is what we are going to consume from the outside. The frontend is going to be responsible of terminating TLS and enforcing mTLS and forwarding SSL headers to the different instances instances of `ntc-rosetta-conf`. Finally, the `frontend` is going to look at the URL path, look for `/rtr0{0,1}` and forward the requests to the corresponding instance of `ntc-rosetta-conf`.
3. Finally, we are going to define a backend per instance of `ntc-rosetta-conf`. In this example we have two of them. The backend needs to specify how to connect to it and also it needs to remove the `/rtr0x` bit from the URL as that's not part of our service.

1.4.2 docker-compose

`docker-compose` is going to be responsible of instantiating the loadbalancer and both instances of `ntc-rosetta-conf`. There isn't a lot of magic here. Just mount the volumes with the configuration for haproxy, the data directories for each instance of `ntc-rosetta-conf` and disable ssl on them as it will be terminated on the loadbalancer:

```
---
version: '2.2'
```

(continues on next page)

(continued from previous page)

```
services:
    loadbalancer:
        image: haproxy:2.0-alpine
        volumes:
            - ./haproxy:/etc/haproxy
        command: [
            "haproxy",
            "-f", "/etc/haproxy/haproxy.cfg",
        ]
        ports:
            - 65443:65443
        networks:
            net1:
                ipv4_address: 172.21.33.10
                ipv6_address: 2001:db8:33::10

    rtr00:
        build:
            context: ../../..
            dockerfile: Dockerfile
            args:
                PYTHON: 3.6
        networks:
            net1:
                ipv4_address: 172.21.33.100
                ipv6_address: 2001:db8:33::100
        volumes:
            - ./data/rtr00:/data
        command: [
            "ntc-rosetta-conf",
            "serve",
            "--datamodel", "openconfig",
            "--pid-file", "/tmp/ntc-rosetta-conf-demo.pid",
            "--log-level", "debug",
            "--data-file", "/data/data.json",
            "--port", "8443",
            "--disable-ssl",
        ]

    rtr01:
        build:
            context: ../../..
            dockerfile: Dockerfile
            args:
                PYTHON: 3.6
        networks:
            net1:
                ipv4_address: 172.21.33.101
                ipv6_address: 2001:db8:33::101
        volumes:
            - ./data/rtr01:/data
        command: [
            "ntc-rosetta-conf",
            "serve",
            "--datamodel", "openconfig",
            "--pid-file", "/tmp/ntc-rosetta-conf-demo.pid",
```

(continues on next page)

(continued from previous page)

```
            "--log-level", "debug",
            "--data-file", "/data/data.json",
            "--port", "8443",
            "--disable-ssl",
        ]

networks:
    net1:
        driver: bridge
        enable_ipv6: true
        ipam:
            config:
            - subnet: 172.21.33.0/24
            - subnet: 2001:db8:33::/64
```

After everything is up now you should be able to access each particular instance via `/rtr00` and `/rtr01` respectively. For instance; `https://rosetta:65443/rtr00/restconf/data/openconfig-interfaces:interfaces`

This can look like it's going to be a lot if you have hundreds or thousands of devices but as you probably figured already these two configuration files are very easy to template and automate.

1.5 YANG models

`ntc-rosetta-conf` supports same YANG models that `ntc-rosetta` does so refer to its documentation for details on those.

In addition, RESTCONF operations require the following models:

1.5.1 ietf-yang-library

This module contains monitoring information about the YANG modules and submodules that are used within a YANG-based server. Copyright (c) 2016 IETF Trust and the persons identified as authors of the code. All rights reserved. Redistribution and use in source and binary forms, with or without modification, is permitted pursuant to, and subject to the license terms contained in, the Simplified BSD License set forth in Section 4.c of the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info). This version of this YANG module is part of RFC 7895; see the RFC itself for full legal notices.

Types

revision-identifier

Represents a specific date in YYYY-MM-DD format.

type: `string`

pattern: `\d{4}-\d{2}-\d{2}`

Data nodes

/modules-state

Contains YANG module monitoring information.

nodetype: `container`

/modules-state/module-set-id

Contains a server-specific identifier representing the current set of modules and submodules. The server MUST change the value of this leaf if the information represented by the 'module' list instances has changed.

nodetype: `leaf`

Type: `string`

/modules-state/module

Each entry represents one revision of one module currently supported by the server.

nodetype: `list`

/modules-state/module/name

The YANG module or submodule name.

nodetype: `leaf` (list key)

Type: `yang:yang-identifier`

/modules-state/module/revision

The YANG module or submodule revision date. A zero-length string is used if no revision statement is present in the YANG module or submodule.

nodetype: `leaf` (list key)

Type: `union`

- **Type**: `revision-identifier`
- **Type**: `string`

/modules-state/module/schema

Contains a URL that represents the YANG schema resource for this module or submodule. This leaf will only be present if there is a URL available for retrieval of the schema for this entry.

nodetype: `leaf`

Type: `inet:uri`

/modules-state/module/namespace

The XML namespace identifier for this module.

nodetype: `leaf`

Type: `inet:uri`

/modules-state/module/feature

List of YANG feature names from this module that are supported by the server, regardless of whether they are defined in the module or any included submodule.

nodetype: `leaf-list`

Type: `yang:yang-identifier`

/modules-state/module/deviation

List of YANG deviation module names and revisions used by this server to modify the conformance of the module associated with this entry. Note that the same module can be used for deviations for multiple modules, so the same entry MAY appear within multiple 'module' entries. The deviation module MUST be present in the 'module' list, with the same name and revision values. The 'conformance-type' value will be 'implement' for the deviation module.

nodetype: `list`

/modules-state/module/deviation/name

The YANG module or submodule name.

nodetype: `leaf` (list key)

Type: `yang:yang-identifier`

/modules-state/module/deviation/revision

The YANG module or submodule revision date. A zero-length string is used if no revision statement is present in the YANG module or submodule.

nodetype: `leaf` (list key)

Type: `union`

- **Type**: `revision-identifier`
- **Type**: `string`

/modules-state/module/conformance-type

Indicates the type of conformance the server is claiming for the YANG module identified by this entry.

nodetype: `leaf`

Type: `enumeration`

- `implement`: Indicates that the server implements one or more

protocol-accessible objects defined in the YANG module identified in this entry. This includes deviation statements defined in the module. For YANG version 1.1 modules, there is at most one module entry with conformance type 'implement' for a particular module name, since YANG 1.1 requires that, at most, one revision of a module is implemented. For YANG version 1 modules, there SHOULD NOT be more than one module entry for a particular module name.

- `import`: Indicates that the server imports reusable definitions

from the specified revision of the module but does not implement any protocol-accessible objects from this revision. Multiple module entries for the same module name MAY exist. This can occur if multiple modules import the same module but specify different revision dates in the import statements.

/modules-state/module/submodule

Each entry represents one submodule within the parent module.

nodetype: `list`

/modules-state/module/submodule/name

The YANG module or submodule name.

nodetype: `leaf` (list key)

Type: `yang:yang-identifier`

/modules-state/module/submodule/revision

The YANG module or submodule revision date. A zero-length string is used if no revision statement is present in the YANG module or submodule.

nodetype: `leaf` (list key)

Type: `union`

- **Type**: `revision-identifier`
- **Type**: `string`

/modules-state/module/submodule/schema

Contains a URL that represents the YANG schema resource for this module or submodule. This leaf will only be present if there is a URL available for retrieval of the schema for this entry.

nodetype: `leaf`

Type: `inet:uri`

/yang-library-change

Generated when the set of modules and submodules supported by the server has changed.

nodetype: `notification`

/yang-library-change/module-set-id

Contains the module-set-id value representing the set of modules and submodules supported at the server at the time the notification is generated.

nodetype: `leaf`

Type: `leafref`

- **path reference**: `/modules-state/module-set-id`

1.5.2 ntc-rosetta-conf

This module describes all the operations that ntc-rosetta-conf can perform.

Identities

base: *RESULT*

Base identity for results

success

base identity: RESULT

Operation succeeded

error

base identity: RESULT

Operation failed

base: *PLATFORM*

Base identity for device's platform

ios

base identity: PLATFORM

IOS device

junos

base identity: PLATFORM

Junos device

Data nodes

/device

Top-level container for device configuration and state

nodetype: `container`

/device/config

Top-level container for device configuration

nodetype: `container`

/device/config/platform

Device platform. Some RPC methods will require you to set this

nodetype: `leaf`

Type: `identityref`

- **base**: `PLATFORM`

/ping

Ping the server

nodetype: `rpc`

/ping/input

nodetype: `input`

/ping/output

nodetype: `output`

/ping/output/result

Whether the operation succeeded or not

nodetype: `leaf`

Type: `identityref`

- **base**: `ntc-rosetta-conf:RESULT`

/ping/output/error-message

If the operation failed, message describing the error

nodetype: `leaf`

Type: `string`

/merge

Call ntc-rosetta merge method. Visit https://ntc-rosetta.readthedocs.io/en/latest/tutorials/ios_merging.html for details

nodetype: `rpc`

/merge/input

nodetype: `input`

/merge/input/replace

Replace argument for the given operation

nodetype: `leaf`

Type: `boolean`

/merge/output

nodetype: `output`

/merge/output/result

Whether the operation succeeded or not

nodetype: `leaf`

Type: `identityref`

- **base**: `ntc-rosetta-conf:RESULT`

/merge/output/error-message

If the operation failed, message describing the error

nodetype: `leaf`

Type: `string`

/merge/output/native

Configuration in native format

nodetype: `leaf`

Type: `string`

/parse

Call ntc-rosetta parse method. Visit https://ntc-rosetta.readthedocs.io/en/latest/tutorials/ios_parsing.html for detauls. Loads the parsed object into the candidate database.

nodetype: `rpc`

/parse/input

nodetype: `input`

/parse/input/native

Native confguration to parse

nodetype: `leaf`

Type: `string`

/parse/input/validate

Valiadte the configuration prior to load it into the candidate databbase

nodetype: `leaf`

Type: `boolean`

/parse/output

nodetype: `output`

/parse/output/result

Whether the operation succeeded or not

nodetype: `leaf`

Type: `identityref`

- **base**: `ntc-rosetta-conf:RESULT`

/parse/output/error-message

If the operation failed, message describing the error

nodetype: `leaf`

Type: `string`

/translate

Call ntc-rosetta parse method. Visit https://ntc-rosetta.readthedocs.io/en/latest/tutorials/ios_translate.html

nodetype: `rpc`

/translate/input

nodetype: `input`

/translate/input/database

Database to translate

nodetype: `leaf`

Type: `enumeration`

/translate/input/replace

Replace argument for the given operation

nodetype: `leaf`

Type: `boolean`

/translate/output

nodetype: `output`

/translate/output/result

Whether the operation succeeded or not

nodetype: `leaf`

Type: `identityref`

- **base**: `ntc-rosetta-conf:RESULT`

/translate/output/error-message

If the operation failed, message describing the error

nodetype: `leaf`

Type: `string`

/translate/output/native

Configuration in native format

nodetype: `leaf`

Type: `string`

1.6 CONTRIBUTING

All contributions are welcome and even though there are no strict or formal requirements to contribute there are some basic rules contributors need to follow:

1. Make sure your contribution is original and it doesn't violate anybody's copyright
2. Make sure tests pass
3. Make sure your contribution is tested

Below you can find more information depending on what you are trying to contribute, in case of doubt don't hesitate to open a PR with your contribution and ask for help.

1.6.1 Running tests

To run tests you need `docker` and `GNU Make`. If you meet the requirements all you need to do is execute `make tests`. All the tests will run inside docker containers you don't need to worry about the environment.

1.6.2 Adding documentation

If you want to contribute documentation you need to be sligthly familiar with sphinx as that's the framework used in this project (and most python projects) to build the documentation.

In addition, if you want to contribute with tutorials or code examples you need to be familiar with jupyter. The advantage of using jupyter notebooks over just plain text is that notebooks can be tested. This means code examples and tutorials will be tested by the CI and ensure they stay relevant and work.

The easiest way of working with jupyter is by executing `make jupyter` in your local machine and pointing your browser to http://localhost:8888/notebooks/docs. If you are adding a new notebook don't forget to add it to sphinx's documentation.

1.6.3 Coding Style

We use black to format the code.

1.6.4 Adding new features

New features need to come with tests and a tutorial in the form of a jupyter notebook so it can be tested.

1.6.5 mypy

We use mypy to bring static typing to our code. This adds some complexity but results in cleaner, less error-prone and more understandable code.

1.7 CHANGELOG

1.7.1 0.0.1

- Initial release

Indices and tables

- genindex
- modindex
- search