Jetconf

Oct 14, 2019

Guides

Jetconf is an implementation of the RESTCONF protocol written in Python 3.

CHAPTER 1

Installation

1.1 Requirements

Jetconf requires **Python 3.5** or **newer**:

```
$ apt-get install python3
$ apt-get install python3-pip
```

Other requirements should be installed automatically during installation.

1.2 Stable version - PyPI

Stable version is the most actual package version provided by Python Package Index (PyPI):

```
$ python3 -m pip install jetconf
```

1.3 Latest version - GitHub

Latest version is the most actual source code available in the Jetconf GitHub repository. It is the `master` branch.

To install Jetconf from source:

```
$ git clone https://github.com/CZ-NIC/jetconf.git
$ cd jetconf
$ pip install -r requirements.txt
$ python3 -m pip install .
```

CHAPTER 2

Sample jukebox-jetconf backend

`jukebox-jetconf` is an sample backend project created for Jetconf. It is very useful as template for start developing a new Jetconf backend.

2.1 Installation

Clone backend project from repository:

```
$ git clone https://github.com/CZ-NIC/jukebox-jetconf
```

Install backend package:

```
$ cd jukebox-jetconf
$ pip install .
```

Now the backend package should be installed.

2.2 Configuration

In the `data` directory of Jetconf repository there are some example files.

- `jetconf@.service`: simple systemd integration
- `example-config.yaml`: configuration file configured to working with *jukebox* backend and other files in *data* directory
- `doc-root`: default root directory for Jetconf HTTP server
- `ca.pem`: example generated self-signed Certification Authority certificate

server certificate:

- `server_localhost.crt` : example generated Jetconf server certificate

- `server_localhost.key`: key for *server_localhost.crt* certificate

client certificates:

- `example-client.pem`: basic client certificate
- `example-client_curl.pem`: client certificate for usage with cURL
- `example-client_browser.pfx`: client certificate in PKCS #12 format for usage with browser
- `pfx_passwd`: password for *example-client_browser.pfx* certificate

> **Warning:** Certificates provided with Jetconf are only generated to test or try Jetconf. Never use these certificates in final application.

Easiest way to run Jetconf with jukebox backend is to clone full Jetconf repository and start working in `data` directory:

```
$ git clone https://github.com/CZ-NIC/jetconf.git
$ cd jetconf/data
```

Paths in `example-config.conf` must be updated. If backend is installed and paths in configuration file are configured, Jetconf can be run.

Set up all on your own:

- *Configuration options*
- *Generating SSL Certificates*

CHAPTER 3

Run Jetconf

3.1 command line

All logging information will be displayed in terminal:

```
$ jetconf -c <path_to_config_file.yaml>
```

3.2 systemd

In `data` directory there is a simple `systemd` service file for Jetconf. To allow running Jetconf using systemd, this file needs to be copied to `/etc/systemd/system/`:

```
$ cp jetconf@.service /etc/systemd/system/jetconf@.service
```

Change the user in `/etc/systemd/system/jetconf@.service` to yours or create new `jetconf` user.

Move `.yaml` config file to `/etc/jetconf`. It must be named like `config-backend_name.yaml`. For example, configuration file for *jukebox* backend will be `config-jukebox.yaml`. It is nice to use Jetconf backend's name without *jetconf* suffix.

```
$ cp example-config.yaml /etc/jetconf/config-jukebox.yaml
```

Last, Jetconf service can be started in format `jetconf@backend_name.service`. For `jukebox` backend from above:

```
$ systemctl start jetconf@jukebox.service
```

CHAPTER 4

Generating SSL Certificates

This tutorial explains how to generate self-signed certificates for the Jetconf server and clients using OpenSSL. Example certificates can be found in `data` subdirectory.

> **Warning:** Self-signed certificates are of course not considered trustworthy by web browsers and operating systems, so they are only suitable for testing.

Two bash scripts to help generate SSL certificates are placed in `/utils/cert_gen` directory

- `gen_server_cert.sh` is used once for generating the server certificate.
- `gen_client_cert.sh` is used repeatedly for creating client certificates.

Their usage is described below.

Installing OpenSSL

To start with, check that OpenSSL is installed. If not, it should be available as a package for your operating system:

```
$ apt-get install openssl
```

4.1 Certification Authority

The generated server and client certificates have to be signed by a Certification Authority (CA). For testing purposes, though, a self-signed CA-like certificate will do.

> **Warning:** For production uses, a trusted CA should always be used.

The easiest, but least secure, way is to use the pre-generated CA-like certificate and private key from the files `ca.pem` and `ca.key` available from the `utils/cert_gen` directory.

Alternatively, the CA-like certificate and key can be generated using the procedure below.

4.1.1 Generate your own CA-like certificate

Make or move to your working directory:

```
$ mkdir my_ca_cert
$ cd my_ca_cert
```

Generate `ca.key`. see genrsa:

```
$ openssl genrsa -out ca.key 2048
```

Generate `ca.pem` certificate. see x509:

```
$ openssl req -x509 -new -nodes -key ca.key -sha256 -days 1024 -out ca.pem
```

Some parameters of the certificate have to be filled in. They are not terribly important for testing purposes. For example:

```
Country Name (2 letter code) [AU]:CZ
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example CA
Organizational Unit Name (eg, section) []:exca.cz
Common Name (e.g. server FQDN or YOUR name) []:mail@exca.cz
Email Address []:mail@exca.cz
```

4.2 Server Certificate

To generate a new server certificate for JetConf that will be accepted even by the more pedantic web browsers like Chrome, just run the provided `gen_server_cert.sh` script.

The script can be used in one of the two following ways.

The command will generate a new server private key along with the certificate:

```
$ ./gen_server_cert.sh <out_file_suffix> <domain/ip>
```

In this case, the name of the private key file passed to the script as the <*server_key*> argument:

```
$ ./gen_server_cert.sh <out_file_suffix> <domain/ip> <server_key>
```

The script autodetects if the certificate is being issued for a domain name or an IP address <*domain/ip*>, and sets the appropriate SAN value.

For example, this command will create a certificate named `server_example.crt` for `example.com` domain with new private key `server_example.key`:

```
$ ./gen_server_cert.sh example example.com
```

If you want this certificate to be accepted by your web browser, the issuing CA's certificate needs to be imported to your browser.

Warning: It is strongly recommended to do not import the provided CA's certificate `ca.pem` to your production browser, as its private key is publicly known. If you do so, someone could perform a MITM attack to any connection with an SSL-protected website.

4.3 Client Certificate

The `gen_client_cert.sh` script is intended for generating client certificates signed by the previously created CA-like certificate.

The script is used simply as follows:

```
$ ./gen_client_cert.sh <email_address>
```

The issued certificate will use the email address passed in the argument is used as the `emailAddress DN` and `commonName` parameter of the client certificate. Also, the email address identifies the client to the JetConf server by default.

For example, the command:

```
$ ./gen_client_cert.sh joe@example.net
```

will generate the following files:

- `joe@example.net.pem` - the client certificate
- `joe@example.net.key` - the client private key
- `joe@example.net_curl.pem` - the previous 2 files combined and protected by a password. Some utilities, such as curl, expect the client certificate in this format.
- `joe@example.net.pfx` - *PKCS#12* format for browsers. The password is the email address, i.e. `joe@example.net` in this case.

CHAPTER 5

Architecture

Jetconf is an implementation of the RESTCONF protocol for remote management of network devices and services.

YANG 1.1 data modelling language is also fully supported.

Jetconf is written in Python 3 language and available as open source software under the terms of the GNU GPLv3 license.

5.1 Requirements and Restrictions

Jetconf is a compliant RESTCONF implementation supporting all mandatory features.

Although it is written in Python, it should be fast enough to support large configuration databases with moderate rate of changes. A typical use can may be an authoritative TLD name server in which Jetconf covers both server management and domain provisioning.

Jetconf supports only the **JSON** data encoding, i.e. media types with the `+json` structured syntax suffix, such as `application/yang.data+json`.

Jetconf supports only HTTP/2 transport. Entity tags (ETag headers) can be generated for all data resources, whereas timestamps (Last-Modified headers) are supported for all container-like resources, i.e. not for individual leaf and leaf-list instances.

5.2 Datastore

Jetconf uses YANGSON library, which is responsible for *storage*, *validation* and *manipulation* with YANG data. This library utilizes an in-memory persistent structure called *"Zipper"* where the YANG data are kept in.

Jetconf also provides an option to serialize data into `.json` file on each commit, which ensures that all configuration data will be persistent among server startups.

Additionally, the datastore can have an access control module associated with it. If so, every read/write operation will be verified with this ACM.

5.3 Access Control

The current version of Jetconf implements NACM access control system, which enables to specify fine-grained access permissions to particular data resources.

The NACM data can only be edited by privileged users in startup *Configuration options*.

5.4 Jetconf Server Loop

1. The client opens a secure TLS connection.
2. The client is authenticated via a client certificate. The certificate of the CA that issued the client certificate needs to be specified in the configuration file. The *e-mail* or *commonName* field obtained from the client certificate is henceforth used as the username, in particular for access control. If the client cannot be authenticated, for example because his certificate has expired or because it was not issued by correct CA, the connection is terminated.
3. The server waits for an incoming client request.
4. A received request is parsed and handed over to the appropriate component. If the media type specified is not supported (in particular, is not `+json`), `415 Unsupported Media Type` is sent, If the message is otherwise invalid, `400 Bad Request` is sent.
5. The NACM data are queried to determine which groups the user is a member of.
6. Depending on the type of the request (read, write or RPC operation invocation) and the Request-URI, the required permissions are determined, and the NACM database is checked to verify that the user possess all of them. If not, `403 Forbidden` is sent.
7. If the request is an RPC operation, it is invoked and an appropriate reply or error message generated.
8. If the request is a read operation, the corresponding data are retrieved from the datastore and formatted into a reply, or an error status code is returned.
9. If the request is a write operation, the changes are applied using a persistent structure API (so that the original unmodified configuration remains available). The new configuration is passed to the *YANGSON* library for validation. If the validation succeeds, the new configuration is written to non-volatile memory, and passed to server instrumentation that applies the necessary changes. An appropriate response or error code is generated and sent.
10. After finishing one of the steps 7, 8 or 9, the server returns to step 3.

5.5 Python Modules

- `rest_server`: a module providing the HTTP/2 and user authentication functionality for REST operations,
- `http_handlers`: handlers connecting HTTP requests to datastore operations,
- `data`: datastore implementation,
- `nacm`: basic NACM implementation,
- `config`: a module for reading and parsing the config file,
- `helpers`: static helper classes shared across modules,
- `op_internal`: implementation of Jetconf internal RPCs,
- `errors`: definition of exceptions used in Jetconf.

CHAPTER 6

Configuration options

Jetconf configuration is set as `.conf` text file in YAML format loaded by Jetconf on startup. Jetconf configuration has two types sections, *common* sections and *application-specific* sections.

6.1 Common sections

Common sections are configuring core Jetconf settings available in any running same version of Jetconf. It do not depend on the Jeconf backend package.

6.1.1 GLOBAL

Example

```
GLOBAL:
    TIMEZONE: "Europe/Prague"
    LOGFILE: "-"
    PIDFILE: "/tmp/jetconf.pid"
    PERSISTENT_CHANGES: true
    LOG_LEVEL: "info"
    LOG_DBG_MODULES: ["usr_conf_data_handlers", "data"]
```

(continues on next page)

(continued from previous page)

```
    YANG_LIB_DIR: "yang-data/"
    DATA_JSON_FILE: "data.json"
    VALIDATE_TRANSACTIONS: true
    CLIENT_CN: false
    BACKEND_PACKAGE: "jetconf_jukebox"
```

Options

TIMEZONE:

Default: `"GMT"`

A timezone of the Jetconf server. This is necessary because all timestamps returned in HTTP response headers need to be returned in GMT.

LOGFILE:

Default: `"-"`

A location of Jetconf's log file. This can be either a `path` on the filesystem or a `-`. If configured as a `-`, Jetconf server will run in foreground and all logging information will be written to stdout (suitable for testing).

PIDFILE:

Default: `"/tmp/jetconf.pid"`

A location of Jetconf's process ID file.

PERSISTENT_CHANGES:

Default: `true`

This option specifies if the changes commited to datastore will also be synchronized to the filesystem (*JSON* file defined by the `DATA_JSON_FILE` option). It should be set to true in most cases, but can be turned off for i.e. testing purposes. If turned off, the Jetconf datastore will contain exactly the same initial data at every startup.

LOG_LEVEL:

Default: `"info"`

Defines the Jetconf's log verbosity. Possible values are: `debug`, `info`, `warning` and `error`.

LOG_DBG_MODULES:

Default: `[*]`

When `LOG_LEVEL` is set to `debug`, this options defines list of Python modules which will write out debugging information. This is useful to prevent flooding the log with debugging messages from irrelevant modules. I.e. when debugging `"usr_conf_data_handlers"` module, you may not be interested with debug information from the `"nacm"`. Can be set to wildcard `*`.

YANG_LIB_DIR:

Default: `"yang-data/"`

Specifies the location of **YANG library**. This is the directory containing `*.yang` files, it must also contain the `"yang-library-data.json"` file with configuration and description of all present YANG modules.

DATA_JSON_FILE:

Default: `"data.json"`

A path to JSON file containing the datastore data. This file will be loaded at Jetconf startup. If `PERSISTENT_CHANGES` option is set to true, all changes made to the datastore will be also stored to this file.

VALIDATE_TRANSACTIONS:

Default: `true`

This option defines if the datastore data should be validated according to YANG data model after a transaction is commited. It should be set to true except for testing and debugging purposes.

CLIENT_CN:

Default: `false`

If enabled, Jetconf will use `commonName` to identify users. By default Jetconf is using `emailAddress` to identify users.

BACKEND_PACKAGE:

Default: `"jetconf_jukebox"`

This option selects the package with backend bindings that Jetconf will use. An exact name of the Python package has to be specified here, and also the package has to be installed in Python's environment.

6.1.2 HTTP_SERVER

Example

```
HTTP_SERVER:
    DOC_ROOT: "doc-root"
    DOC_DEFAULT_NAME: "index.html"
    API_ROOT: "/restconf"
    API_ROOT_STAGING: "/restconf_staging"
    SERVER_NAME: "jetconf-h2"
    UPLOAD_SIZE_LIMIT: 1
    LISTEN_LOCALHOST_ONLY: false
    PORT: 8443
    DISABLE_SSL: false
    SERVER_SSL_CERT: "server.crt"
    SERVER_SSL_PRIVKEY: "server.key"
    CA_CERT: "ca.pem"
    DBG_DISABLE_CERTS: false
```

Options

DOC_ROOT:

Default: `"doc-root"`

A root directory where regular files will be placed. All HTTP GET requests outside `API_ROOT` are considered as requests for regular files on filesystem.

DOC_DEFAULT_NAME:

Default: `"index.html"`

A default filename in DOC_ROOT and its subdirectories.

API_ROOT:

Default: `"/restconf"`

Defines the base URI of RESTCONF data. All requests for resources inside API_ROOT will be considered as RESTCONF requests. It is usually not needed to change this value. Example: `"/restconf" -> https://localhost/restconf/ns:some_resouce`

API_ROOT_STAGING:

Default: `/restconf_staging`

Same as above, except this is for staging data (data edited by user, but not commited yet).

SERVER_NAME:

Default: `"jetconf-h2"`

A value returned in `"Server: "` header of HTTP response.

UPLOAD_SIZE_LIMIT:

Default: `1`

A maximum size of incoming data in `PUT` or `POST` body (in **megabytes**), which the server can handle.

LISTEN_LOCALHOST_ONLY:

Default: `false`

If set to `true`, the Jetconf HTTP server will only accept incoming connections from *localhost*.

PORT:

Default: `8443`

The TCP port of Jetconf server.

DISABLE_SSL:

Default: `false`

If enabled, the user authentication system based on client certificates will be turned off and user data will be parsed from HTTP headers. For instance, this change allows you to run Jetconf behind a load balancer where the TLS connection is terminated and http request is forwarded to Jetconf server with relevant headers. Can be combined with `DBG_DISABLE_CERT`.

SERVER_SSL_CERT:

Default: `"server.crt"`

The location of server SSL certificate in PEM format.

SERVER_SSL_PRIVKEY:

Default: `"server.key"`

The location of server SSL private key in PEM format.

CA_CERT:

Default: `"ca.pem"`

The location of certification authority certificate, which is used for issuing client certificates.

DBG_DISABLE_CERTS:

Default: `false`

If enabled, the user authentication system based on client certificates will be turned off and every incoming connection will default to "test-user" username. This should never be turned on in real environment, it is only intended for testing and benchmarking purposes (no HTTP/2 benchmarking tools support client certificates at this moment). Can be combined with `DISABLE_SSL`.

6.1.3 NACM

Example

```
NACM:
    ENABLED: true
    ALLOWED_USERS: ["superuser@example.com", "admin@example.com"]
```

Options

ENABLED:

Default: `true`

If set to false, NACM rules will not be applied.

ALLOWED_USERS:

Default: `[]`

A list of superusers allowed to edit NACM data. By default no superuser is specified.

6.2 Application-specific sections

Application-specific sections are configuring additional Jetconf settings available in specific implementation Jetconf. Depends on Jeconf backend package. Typically it configures Jetconf backend settings, that have to be defined by backend developer.

For instance, configuration required by knot-jetconf backend package.

```
KNOT:
    SOCKET: "/tmp/knot.sock"
```

SOCKET:

Default: `"/tmp/knot.sock"`

A path to KnotDNS control socket.

CHAPTER 7

Backend API

As there can be various use-case scenarios for Jetconf, bindings to a user application are not part of Jetconf server itself, but instead they are implemented in a separate package, so called *"Jetconf backend"*.

The basic idea of Jetconf's backend architecture is that every node of the YANG schema (i.e. container, list, leaf-list) can have a custom handler object assigned to it. When a specific event affecting this node occurs , like configuration data being rewritten or RESCONF operation is called, an appropriate member function of this node handler is invoked.

As there are some major differences between YANG configuration data, state data and RPCs, the architecture of corresponding node handlers in Jetconf also has to follow these differences.

7.1 Backend package architecture

Every backend package for Jetconf server has to provide implementation of following modules.

- `usr_conf_data_handlers` (Handlers for configuration data)
- `usr_state_data_handlers` (Handlers for state data)
- `usr_op_handlers` (Handlers for RESTCONF operations - RPCs)

- `us_action_handlers` (Handlers for RESTCONF actions - operation on node)
- `usr_datastore` (Datastore initialization and save/load functions can be customized here)
- `usr_init` (Jetconf initialization)

In addition to this, backend package can also contain any other resources if necessary. When you consider writing a custom backend, looking at the very basic demo package jukebox-jetconf is a good way to start.

7.2 Handler inheritance

Because some data models can be quite large, it would be difficult to manually assign handler objects to all schema nodes. Because of this, for configuration and state data handlers, Jetconf offers a feature called **Handler inheritance**.

If a node without its own handler is edited, Jetconf finds a nearest parent node which has the handler assigned and then it calls its `replace` or `replace_item` method. It's up to backend developer's decision where to place handler objects, a more fine-grained placement will usually mean better performance (less data rewriting), at the cost of more work.

7.3 usr_init

Useful for code that has to be executed on the startup or on the end of Jetconf backend.

```
def jc_startup():

    # execute code on startup

def jc_end():

    # execute code on end
```

7.4 usr_datastore

Basic `usr_datastore` module without any customization.

```
from jetconf.data import JsonDatastore

class UserDatastore(JsonDatastore):
    pass
```

Customizing `load()` and `save()` functions

```
from jetconf.data import JsonDatastore


class UserDatastore(JsonDatastore):

    def load(self):

        # load method can be customized here

    def save(self):
```

(continues on next page)

(continued from previous page)

```
        # save method can be customized here
```

7.5 usr_conf_data_handlers

The main purpose of configuration data handlers is to project all changes performed on a particular data node, like creation, modification or deletion, to the user application.

A configuration node handler is implemented by creating a custom class which inherits from either `ConfDataObjectHandler` or `ConfDataListHandler` base class depending on the type of YANG node. The former must be used when implementing a handler for `Container` or `Leaf` data nodes, while the latter is used for list-like types, specifically `List` and `Leaf-List`.

7.5.1 ConfDataObjectHandler:

Attributes:

```
self.ds             # type: jetconf.data.BaseDatastore
                    # Can be used for accessing the datastore content from handler␣
↪functions

self.schema_path    # type: str
                    # Contains the YANG schema path to which this handler object is␣
↪registered (as string)

self.schema_node    # type: yangson.schemanode.SchemaNode
                    # Contains the YANG schema path to which this handler object is␣
↪registered (parsed)
```

Arguments:

```
ii:         # type: yangson.instance.InstanceRoute
            # Contains parsed instance identifier of the data node. Useful for␣
↪determining list keys if this data node is a child of some list node.
ch:         # type: jetconf.data.DataChange
            # Can be used for accessing additional edit information, like HTTP input␣
↪data if needed
```

Handlers derived from this base class has to implement the following interface:

```
from jetconf.handler_base import ConfDataObjectHandler
from yangson.instance import InstanceRoute
from jetconf.data import BaseDatastore, DataChange


class MyConfDataHandler(ConfDataObjectHandler):
    def create(self, ii: InstanceRoute, ch: DataChange):

        # Called when a new node is created

    def replace(self, ii: InstanceRoute, ch: DataChange):

        # Called when the node is being rewritten by new data
```

(continues on next page)

(continued from previous page)

```
    def delete(self, ii: InstanceRoute, ch: DataChange):

        # Called when the node is deleted
```

7.5.2 ConfDataListHandler:

Attributes:

```
self.ds             # type: jetconf.data.BaseDatastore
                    # Can be used for accessing the datastore content from handler␣
↪functions

self.schema_path    # type: str
                    # Contains the YANG schema path to which this handler object is␣
↪registered (as string)

self.schema_node    # type: yangson.schemanode.SchemaNode
                    # Contains the YANG schema path to which this handler object is␣
↪registered (parsed)
```

Arguments:

```
ii:     # type: yangson.instance.InstanceRoute
        # Contains parsed instance identifier of the data node. Useful for␣
↪determining list keys if this data node is a child of some list node.

ch:     # type: jetconf.data.DataChange
        # Can be used for accessing additional edit information, like HTTP input data␣
↪if needed
```

Handlers derived from this base class has to implement the following interface:

```
from jetconf.handler_base import ConfDataListHandler
from yangson.instance import InstanceRoute
from jetconf.data import BaseDatastore, DataChange


class MyConfDataHandler(ConfDataListHandler):
    def create_item(self, ii: InstanceRoute, ch: DataChange):

        # Called when a new item is added to the list or leaf-list

    def replace_item(self, ii: InstanceRoute, ch: DataChange):

        # Called when specific list item is being rewritten

    def delete_item(self, ii: InstanceRoute, ch: DataChange):

        # Called when an item is being deleted from the list
```

7.5.3 Handler registration

Assignation of handler objects to the specific data nodes is done via registering them in `jetconf.handler_list.CONF_DATA_HANDLES` handler list. Every `usr_conf_data_handlers` backend module must implement the global function `register_conf_handlers`, where the instantiation and registration of handler objects is done. This function is called on Jetconf startup after datastore initialization and has the following signature.

```
def register_conf_handlers(ds: BaseDatastore):

    ds.handlers.conf.register(MyConfHandler(ds, "/ns:schema-path/to-desired-node"))
```

7.6 usr_state_data_handlers

YANG state data, in contrast to the configuration data, represents more of a current state of the backend application. This means that they are not actually stored in Jetconf's datastore, but instead they has to be generated on the go. Generation of state data is the purpose of state data handlers.

A state data handler has to acquire actual state data from backend application and generate data content of the node where it's assigned. The output data are formatted in Python's representation of *JSON* (using lists, dicts etc.) and their structure must be compliant with the standardized JSON encoding of YANG data (RFC7951).

A state node handler is implemented by creating a custom class which inherits from either `StateDataContainerHandler` or `StateDataListHandler`, depending on the YANG node type. This is similar to he configuration data handlers.

7.6.1 StateDataContainerHandler

Attributes:

```
self.ds             # type: jetconf.data.BaseDatastore
                    # Can be used for accessing the datastore content from handler␣
↪functions

self.data_model     # type: yangson.datamodel.DataModel
                    # Reference to the current data model object

self.sch_pth        # type: str
                    # YANG schema path to which this handler object is registered (as␣
↪string)

self.schema_node    # type: yangson.schemanode.DataNode
                    # Reference to the Yangson schema node object
```

```
from yangson.instance import InstanceRoute
from jetconf.handler_base import StateDataContainerHandler
from jetconf.data import BaseDatastore

class MyStateDataHandler(StateDataContainerHandler):
    def generate_node(self, node_ii: InstanceRoute, username: str, staging: bool)

        # This method has to generate content of the state data node

        return generated_content
```

7.6.2 StateDataListHandler

Attributes:

```
self.ds             # type: jetconf.data.BaseDatastore
                    # Can be used for accessing the datastore content from handler␣
↪functions

self.data_model     # type: yangson.datamodel.DataModel
                    # Reference to the current data model object

self.sch_pth        # type: str
                    # YANG schema path to which this handler object is registered (as␣
↪string)

self.schema_node    # type: yangson.schemanode.DataNode
                    # Reference to the Yangson schema node object
```

Methods:

```
from yangson.instance import InstanceRoute
from jetconf.helpers import JsonNodeT
from jetconf.handler_base import StateDataListHandler
from jetconf.data import BaseDatastore

class MyStateDataHandler(StateDataListHandler):
    def generate_list(self, node_ii: InstanceRoute, username: str, staging: bool) ->␣
↪JsonNodeT:

        # This method has to generate entire list

        return generated_content

    def generate_list(self, node_ii: InstanceRoute, username: str, staging: bool) ->␣
↪JsonNodeT:

        # Generates only one specific item of the list. The list key(s) of the item␣
↪which needs to be generated can be resolved by processing the instance identifier␣
↪passed in 'node_ii' argument.

        return generated_content
```

7.6.3 Handler registration

Assignation of state data handler objects to the specific data nodes is done via registering them in `jetconf.handler_list.STATE_DATA_HANDLERS` handler list. This is similar to the configuration data. Every `usr_state_data_handlers` backend module must implement the global function `register_state_handlers`, where the instantiation and registration of handler objects is done. This function is called on Jetconf startup after datastore initialization and has the following signature:

```
def register_state_handlers(ds: BaseDatastore):

    ds.handlers.state.register(MyStateDataHandler(ds, "/ns:schema-path/to/state/node
↪"))
```

7.7 usr_op_handlers

Handlers for RESTCONF operations.

Arguments:

```
input_args:         # type: JSON
                    # Operation input arguments with structure defined by YANG model

username:           # type: jetconf.data.BaseDatastore
                    # Name of the user who invoked the operation
```

An operation handlers are implemented by adding a custom method to the class `OpHandlersContainer`. Finally, this class is instantiated and its methods are assigned to specific operation names.

```
from yangson.instance import InstanceRoute
from jetconf.helpers import JsonNodeT
from jetconf.data import BaseDatastore

class OpHandlersContainer:
    def __init__(self, ds: BaseDatastore):
        self.ds = ds

    def my_op_handler(self, input_args: JsonNodeT, username: str) -> JsonNodeT:

        # RPC operation Body

        # Operation output data as defined by YANG data model
        # output is not mandatory
        return output_data
```

7.7.1 Handler registration

Every `usr_op_handlers` backend module must implement the global function `register_op_handlers`, where the class `OpHandlersContainer` is instantiated and its methods are tied to individual operations. This function with following signature is called on Jetconf startup after datastore initialization.

```
def register_op_handlers(ds: BaseDatastore):

    op_handlers_obj = OpHandlersContainer(ds)
    ds.handlers.op.register(op_handlers_obj.my_op_handler, "ns:operation")
```

7.8 us_action_handlers

Handlers for RESTCONF actions.

Arguments:

```
ii:     # type: yangson.instance.InstanceRoute
       # Contains parsed instance identifier of the data node. Useful for determining␣
↪list keys if this data node is a child of some list node.

input_args:         # type: JSON
```

(continues on next page)

(continued from previous page)

```
                    # Operation input arguments with structure defined by YANG model

username:           # type: jetconf.data.BaseDatastore
                    # Name of the user who invoked the operation
```

An action handlers are implemented by adding a custom method to the class `ActionHandlersContainer`. Finally, this class is instantiated and its methods are assigned to specific action names and node path.

```
from yangson.instance import InstanceRoute
from jetconf.helpers import JsonNodeT
from jetconf.data import BaseDatastore

class ActionHandlersContainer:
    def __init__(self, ds: BaseDatastore):
        self.ds = ds

    def my_action_handler(self, ii: InstanceRoute, input_args: JsonNodeT, username:␣
↪str) -> JsonNodeT:

        # Action Body

        # Action output data as defined by YANG data model
        # output is not mandatory
        return output_data
```

7.8.1 Handler registration

Every `usr_action_handlers` backend module must implement the global function `register_action_handlers`, where the class `ActionHandlersContainer` is instantiated and its methods are tied to individual actions. This function with following signature is called on Jetconf startup after datastore initialization.

```
def register_action_handlers(ds: BaseDatastore):
    act_handlers_obj = ActionHandlersContainer(ds)
    ds.handlers.action.register(act_handlers_obj.my_action_handler, "/ns:schema-path/
↪to/action/node")
```

CHAPTER 8

For Developers

Warning: It is highly recommended to set up a virtual environment for Jetconf development. The following procedure uses the `venv` module for this purpose (it is included in the standard Python library since version 3.3).

8.1 Development Environment

1. Install the latest stable **Python3** version.
2. Clone the Jetconf project in a directory of your choice:

```
$ git clone https://github.com/CZ-NIC/jetconf.git
```

3. Create the virtual environment:

```
$ python3 -m venv jetconf
```

4. Activate the virtual environment:

```
$ cd jetconf
$ source bin/activate
```

5. Install required standard packages inside the virtual environment:

```
$ make install-deps
```

If you are prompted to upgrade `pip`, you can do that, too.

When you are inside the virtual environment, the shell prompt should change to something like:

```
(jetconf) $
```

To leave the virtual environment, just do:

```
$ deactivate
```

Tip: The virtual environment can be entered anytime later by executing step 4. The steps preceding it need to be performed just once.

The setup described above has a few consequences that have to be kept in mind:

- Any project files that need to go to `bin` (executable Python scripts),‘‘include‘‘ or `lib` have to be added as exceptions to *.gitignore*, for example:

  ```
  !bin/jetconf
  ```

- After adding a new Python module dependency, it is necessary to run:

  ```
  $ make deps
  ```

and commit the new content of `requirements.txt`.

8.2 Run from source

For development purposes, Jetconf can also be started directly from git repository with `run.py` script:

```
$ ./run.py -c <path_to_config_file.yaml>
```

CHAPTER 9

Release Notes

9.1 0.3.6

9.1.1 Added

- Root Resource Discovery: https://tools.ietf.org/html/rfc8040#section-3.1
- DISABLE_SSL and CLIENT_CN options: https://github.com/CZ-NIC/jetconf/pull/8
- RESTCONF actions: https://tools.ietf.org/html/rfc8040#section-3.6
- simple systemd unit: https://github.com/CZ-NIC/jetconf/blob/master/data/jetconf%40.service

CHAPTER 10

Jetconf Backends

- jukebox-jetconf
- knot-jetconf

CHAPTER 11

Jetconf Clients

Useful links:

- *Generating SSL Certificates*
- *Configuration options*

11.1 cURL

- cURL
- cURL GitHub

A Swiss-knife tool for HTTP/2.

11.1.1 View data in a terminal with cURL

User's certificate with `_curl` suffix in `.pem` format is needed.

After this command you should get some data from Jetconf server in json. Do not forget to set `<path_to_pem_cert>` and `<jetconf server ip address>`:

```
$ curl --http2 -k --cert-type PEM -E <path_to_pem_cert> -X GET https://<jetconf_
↪server_ip_address>:8443/restconf/data
```

If `DISABLE_SSL` and `CLIENT_CN` are both set to `true`, the following command can be used. `<username>` is sent in HTTP header:

```
$ curl --http2-prior-knowledge -H "X-SSL-Client-CN: <username>" -X GET http://
↪<jetconf_server_ip_address>:8443/restconf/data
```

11.2 Jetscreen

- Jetscreen Page
- Jetscreen Source

A prototype of an interactive graphical Jetconf client written in Angular 2. Works only with the JetConf implementation.

11.2.1 View data with Jetscreen

User's certificate in `.pfx` format must be imported to the browser.

1. Open public Jetscreen Page
2. Enter your Jetconf server URL and press *enter* or click the *Reset* button. You may be prompted to select a user certificate.
3. Top-level data containers should then appear.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search