netjsonconfig documentation

Release 0.6.2

Federico Capoano

Sep 11, 2017

Contents

Netjsonconfig is part of the OpenWISP project. **netjsonconfig** is a python library that converts NetJSON *DeviceConfiguration* objects into real router configurations that can be installed on systems like OpenWRT, LEDE or OpenWisp Firmware.

Its main features are:

- OpenWRT / LEDE support
- OpenWisp Firmware support
- OpenVPN support
- Possibility to support more firmwares via custom backends
- Based on the NetJSON RFC
- **Validation** based on JSON-Schema
- **Templates**: store common configurations in templates
- **Multiple template inheritance**: reduce repetition to the minimum
- **File inclusion**: easy inclusion of arbitrary files in configuration packages
- **Variables**: reference variables in the configuration
- **Command line utility**: easy to use from shell scripts or from other programming languages

Contents:

CHAPTER 1

Setup

Install stable version from pypi

The easiest way to install *netjsonconfig* is via the python package index:

```
pip install netjsonconfig
```

Install development version

If you need to test the latest development version you can do it in two ways;

The first option is to install a tarball:

```
pip install https://github.com/openwisp/netjsonconfig/tarball/master
```

The second option is to install via pip using git (this will automatically clone the repo and store it on your hard dirve):

```
pip install -e git+git://github.com/openwisp/netjsonconfig#egg=netjsonconfig
```

Install git fork for contributing

If you want to contribute see *Fork repo and install your fork*.

CHAPTER 2

Basic concepts

Before starting, let's quickly introduce the main concepts used in netjsonconfig:

- *NetJSON configuration dictionary*: python dictionary representing the configuration of a router
- *Backend*: python class used to convert the *configuration dictionary* to the format used natively by a router firmware and vice versa
- *Schema*: each backend has a JSON-Schema which defines the useful configuration options that the backend is able to process
- *Validation*: the configuration is validated against its JSON-Schema before being processed by the backend
- *Template*: common configuration options shared among routers (eg: VPNs, SSID) which can be passed to backends
- *Multiple template inheritance*: possibility to inherit common configuration options from more than one template
- *Context (configuration variables)*: variables that can be referenced from the *configuration dictionary*

NetJSON configuration dictionary

netjsonconfig is an implementation of the NetJSON format, more specifically the `DeviceConfiguration` object, therefore to understand the configuration format that the library uses to generate the final router configurations it is essential to read at least the relevant DeviceConfiguration section in the NetJSON RFC.

Here it is a simple *NetJSON DeviceConfiguration* object represented with a python dictionary:

```
{
    "type": "DeviceConfiguration",
    "general": {
        "hostname": "RouterA"
    },
    "interfaces": [
        {
            "name": "eth0",
```

```
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

The previous example describes a device named `RouterA` which has a single network interface named `eth0` with a statically assigned ip address `192.168.1.1/24` (CIDR notation).

Because netjsonconfig deals only with `DeviceConfiguration` objects, the `type` attribute can be omitted.

The previous configuration object therefore can be shortened to:

```
{
    "general": {
        "hostname": "RouterA"
    },
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

From now on we will use the term *configuration dictionary* to refer to *NetJSON DeviceConfiguration* objects.

Backend

A backend is a python class used to convert the *configuration dictionary* to the format used natively by the router (forward conversion, from NetJSON to native) and vice versa (backward conversion, from native to NetJSON), each supported firmware or opearting system will have its own backend and third parties can write their own custom backends.

The current implemented backends are:

- *OpenWrt*
- *OpenWisp* (based on the `OpenWrt` backend)
- *OpenVpn* (custom backend implementing only OpenVPN configuration)

Example initialization of `OpenWrt` backend:

```
from netjsonconfig import OpenWrt

ipv6_router = OpenWrt({
    "interfaces": [
        {
            "name": "eth0.1",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "fd87::1",
                    "mask": 128,
                    "proto": "static",
                    "family": "ipv6"
                }
            ]
        }
    ]
})
```

Each backend will implement **parsers**, **renderers** and **converters** to accomplish its configuration generation or parsing goals.

The process is best explained with the following diagram:

Converters take care of converting between *NetJSON* and the intermediate data structure (and vice versa).

Renderers take care of rendering the intermediate data structure to the native format.

Parsers perform the opposite operation of `Renderers`: they take care of parsing native format and build the intermediate data structure.

Schema

Each backend has a JSON-Schema, all the backends have a schema which is derived from the same parent schema, defined in `netjsonconfig.backends.schema` (view source).

Since different backends may support different features each backend may extend its schema by adding custom definitions.

Validation

All the backends have a `validate` method which is called automatically before trying to process the configuration.

If the passed configuration violates the schema the `validate` method will raise a `ValidationError`.

An instance of validation error has two public attributes:

- `message`: a human readable message explaining the error
- `details`: a reference to the instance of `jsonschema.exceptions.ValidationError` which contains more details about what has gone wrong; for a complete reference see the python-jsonschema documentation

You may call the `validate` method in your application arbitrarily, eg: before trying to save the *configuration dictionary* into a database.

Template

If you have devices with very similar *configuration dictionaries* you can store the shared blocks in one or more reusable templates which will be used as a base to build the final configuration.

Combining different templates

Let's illustrate this with a practical example, we have two devices:

- Router1
- Router2

Both devices have an `eth0` interface in DHCP mode; *Router2* additionally has an `eth1` interface with a statically assigned ipv4 address.

The two routers can be represented with the following code:

```
from netjsonconfig import OpenWrt

router1 = OpenWrt({
    "general": {"hostname": "Router1"}
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ]
        }
    ]
})

router2 = OpenWrt({
    "general": {"hostname": "Router2"},
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ]
        },
        {
            "name": "eth1",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
```

```
                }
            ]
        }
    ]
})
```

The two *configuration dictionaries* share the same settings for the `eth0` interface, therefore we can make the `eth0` settings our template and refactor the previous code as follows:

```
from netjsonconfig import OpenWrt

dhcp_template = {
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ]
        }
    ]
}

router1 = OpenWrt(config={"general": {"hostname": "Router1"}},
                  templates=[dhcp_template])

router2_config = {
    "general": {"hostname": "Router2"},
    "interfaces": [
        {
            "name": "eth1",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
router2 = OpenWrt(router2_config, templates=[dhcp_template])
```

Overriding a template

In many occasions you may want to define a general template which can be overridden in some specific occasions.

A common use case is to define a general radio template and override its channel on certain access points:

```
from netjsonconfig import OpenWrt

general_radio_template = {
```

```
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "protocol": "802.11n",
            "driver": "mac80211",
            "channel": 0,  # zero means "auto"
            "channel_width": 20,
            "country": "US",
            "disabled": False
        }
    ]
}

specific_radio_config = {
    "radios": [
        {
            "name": "radio0",
            "channel": 10,
        }
    ]
}

router1 = OpenWrt(config=specific_radio_config,
                  templates=[general_radio_template])

print(router1.render())
```

Will generater the following output:

```
package wireless

config wifi-device 'radio0'
        option channel '10'
        option country 'US'
        option disabled '0'
        option htmode 'HT20'
        option hwmode '11g'
        option phy 'phy0'
        option type 'mac80211'
```

Multiple template inheritance

You might have noticed that the `templates` argument is a list; that's because it's possible to pass multiple templates that will be added one on top of the other to build the resulting *configuration dictionary*, allowing to reduce or even eliminate repetitions.

Note: When using multiple templates, their order is important. Templates that are specified afterwards override the ones that come first.

To understand this, read the section *Multiple overrides*.

Multiple overrides

Here's a more complex example involving multiple overrides:

```
from netjsonconfig import OpenWrt

general_radio_template = {
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "protocol": "802.11n",
            "driver": "mac80211",
            "channel": 0,  # zero means "auto"
            "channel_width": 20,
            "country": "00",  # world
            "disabled": False
        }
    ]
}

united_states_radio_template = {
    "radios": [
        {
            "name": "radio0",
            "country": "US"
        }
    ]
}

specific_radio_config = {
    "radios": [
        {
            "name": "radio0",
            "channel": 10,
        }
    ]
}

router1 = OpenWrt(config=specific_radio_config,
                  templates=[general_radio_template,
                             united_states_radio_template])

print(router1.render())
```

Will generater the following output:

```
package wireless

config wifi-device 'radio0'
        option channel '10'
        option country 'US'
        option disabled '0'
        option htmode 'HT20'
        option hwmode '11g'
        option phy 'phy0'
        option type 'mac80211'
```

Implementation details

The functions used under the hood to merge configurations and templates are `netjsonconfig.utils.merge_config` and `netjsonconfig.utils.merge_list`:

`netjsonconfig.utils.`**`merge_config`**(*template*, *config*, *list_identifiers=None*)

Merges `config` on top of `template`.

Conflicting keys are handled in the following way:

- simple values (eg: `str`, `int`, `float`, ecc) in `config` will overwrite the ones in `template`
- values of type `list` in both `config` and `template` will be merged using to the `merge_list` function
- values of type `dict` will be merged recursively

Parameters

- **`template`** – template `dict`
- **`config`** – config `dict`
- **`list_identifiers`** – `list` or `None`

Returns merged `dict`

`netjsonconfig.utils.`**`merge_list`**(*list1*, *list2*, *identifiers=None*)

Merges `list2` on top of `list1`.

If both lists contain dictionaries which have keys specified in `identifiers` which have equal values, those dicts will be merged (dicts in `list2` will override dicts in `list1`). The remaining elements will be summed in order to create a list which contains elements of both lists.

Parameters

- **`list1`** – `list` from template
- **`list2`** – `list` from config
- **`identifiers`** – `list` or `None`

Returns merged `list`

Context (configuration variables)

Without variables, many bits of configuration cannot be stored in templates, because some parameters are unique to the device, think about things like a *UUID* or a public ip address.

With this feature it is possible to reference variables in the *configuration dictionary*, these variables will be evaluated when the configuration is rendered/generated.

Here's an example from the real world, pay attention to the two variables, `{{ UUID }}` and `{{ KEY }}`:

```
from netjsonconfig import OpenWrt

openwisp_config_template = {
    "openwisp": [
        {
            "config_name": "controller",
            "config_value": "http",
            "url": "http://controller.examplewifiservice.com",
```

```
            "interval": "60",
            "verify_ssl": "1",
            "uuid": "{{ UUID }}",
            "key": "{{ KEY }}"
        }
    ]
}

context = {
    'UUID': '9d9032b2-da18-4d47-a414-1f7f605479e6',
    'KEY': 'xk7OzA1qN6h1Ggxy8UH5NI8kQnbuLxsE'
}

router1 = OpenWrt(config={"general": {"hostname": "Router1"}},
                  templates=[openwisp_config_template],
                  context=context)
```

Let's see the result with:

```
>>> print(router1.render())
package system

config system
        option hostname 'Router1'
        option timezone 'UTC'
        option zonename 'UTC'

package openwisp

config controller 'http'
        option interval '60'
        option key 'xk7OzA1qN6h1Ggxy8UH5NI8kQnbuLxsE'
        option url 'http://controller.examplewifiservice.com'
        option uuid '9d9032b2-da18-4d47-a414-1f7f605479e6'
        option verify_ssl '1'
```

Warning: When using variables, keep in mind the following rules:

- variables must be written in the form of `{{ var_name }}` or `{{var_name}}`;
- variable names can contain only alphanumeric characters and underscores;
- unrecognized variables will be ignored;

Project goals

If you are interested in this topic you can read more about the *Goals and Motivations* of this project.

Support

See OpenWISP Support Channels.

License

This software is licensed under the terms of the GPLv3 license, for more information, please see full LICENSE file.

CHAPTER 3

AirOS Backend

The `AirOs` backend allows to generate AirOS v8.3 compatible configurations.

Warning: This backend is in experimental stage: it may have bugs and it will receive backward incompatible updates during the first 6 months of development (starting from September 2017). Early feedback and contributions are very welcome and will help to stabilize the backend faster.

Intermediate representation

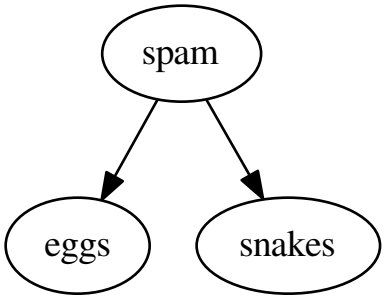
The intermediate representation is the output of the a **converter**, it is backend specific and is built as a tree structure made from python builtins values.

A tree is a *acyclic, directional graph* with an element called *root*.

The root of our tree is stored in the first element of a tuple, along with the root's direct sons as a list:

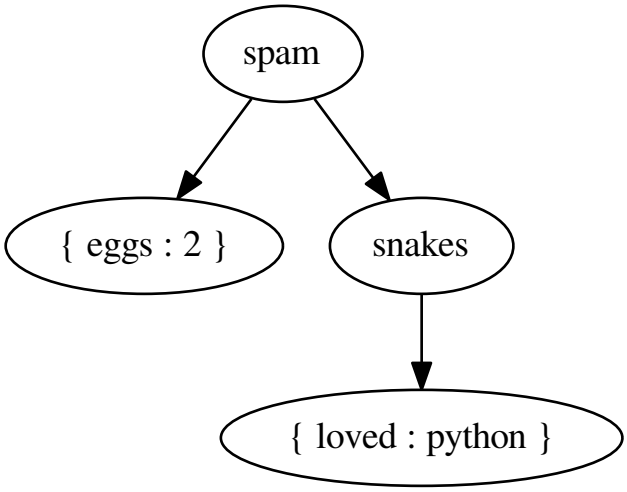
```
tree = (root, direct_sons)
```

As an example here we present the tree *('spam', ['eggs', 'snakes'])*



As a son may be a carrier of a value so we store it in a dictionary instead of adding a *leaf* with another level of recursion.

As an example here we present the tree *('spam', [{ 'eggs': 2 }, { 'snakes' : { 'loved' : 'python' }}])*:



This tree could be tranlated to a configuration file for AirOS that looks like this:

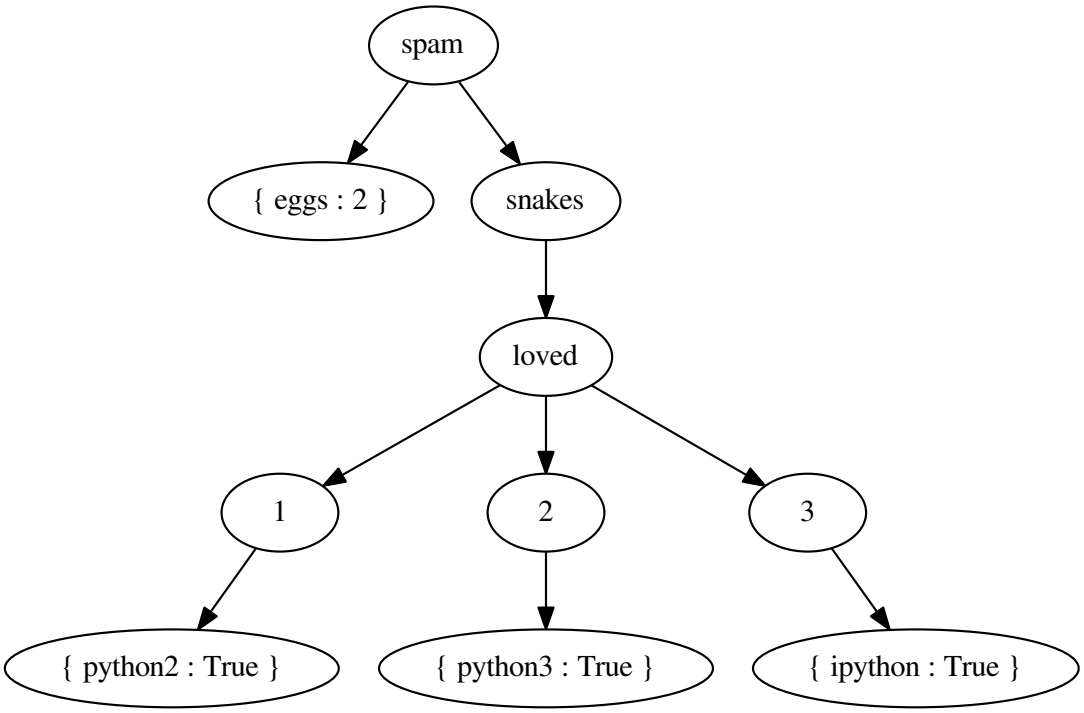
```
spam.eggs=2
spam.snakes.loved=python
```

So our tree representation is based on the simple assumption that a *leaf* is a dictionary without nested values and nested values in a dictionary creates a father-son relationship.

Instead when the configuration requires that the son values must be prefixed from a number, e.g. *vlan.1.devname=eth0* we store a list of dictionaries.

```
(
     'spam',
     [
         {
             'eggs' : 2,
         },
         {
             'snakes' : {
                 'loved' : [
                     {
                         'python2' : True,
                     },
                     {
                         'python3' : True,
                     },
                     {
                         'ipython' : True,
                     }
                  ],
              },
         }
     ]
)
```

And the resulting tree is:



And the configuration is:

```
spam.eggs=2
spam.snakes.loved.1.python2=true
spam.snakes.loved.2.python3=true
spam.snakes.loved.2.ipython=true
```

The process by which we can go from the intermediate representation from the output configuration is called flattening, you can find more in the next section.

Flattening

To avoid at all cost a recursive logic in the template we flatten the intermediate representation to something that has a *namespace* a *key* and a *value*.

The objective is to go from a python *NetJSON configuration dictionary* that we get from loading a NetJSON to the AirOS configuration.

An input *NetJSON configuration dictionary* is just a python dictionary, e.g.:

```
#python
{
     'interfaces' : [
          {
               'name' : 'eth0.1',
               'type' : 'ethernet',
               'comment' : 'management vlan'
               'comment' : 'management'
          },
          {
               'name' : 'eth0.2',
               'type' : 'ethernet',
               'comment' : 'traffic'
          }
     ]
}
```

And this must be converted to an appropiate AirOS configuration which looks like this:

```
vlan.1.comment=management
vlan.1.devname=eth0
vlan.1.id=1
vlan.1.status=enabled
vlan.2.comment=management
vlan.2.devname=eth0
vlan.2.id=2
vlan.2.status=enabled
vlan.status=enabled
```

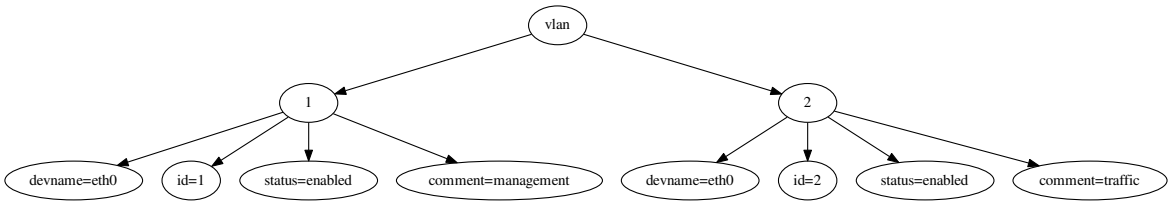
To do this we must convert the *NetJSON configuration dictionary* into something that resembles the target text, the output configuration.

```
(
     # namespace
     'vlan',
     #options
     [
          {
```

```
            # key : value
            '1.devname' : 'eth0',
            '1.id' : '1'
            '1.status' : 'enabled',
            '1.comment' : 'management'
        },
        {
            '2.devname' : 'eth0',
            '2.id' : '2'
            '2.status' : 'enabled',
            '2.comment' : 'traffic'
        }
    ]
)
```

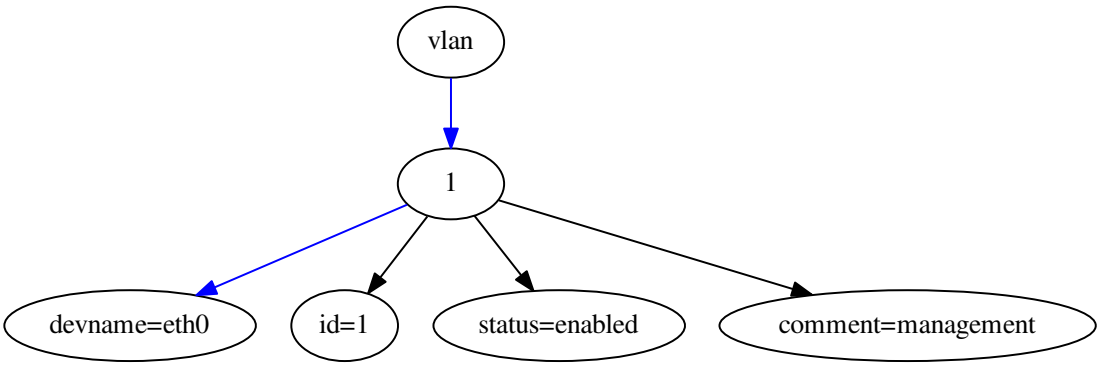
And to do that we get rid of the multiple indentation levels by flattening the tree structure.

The tree associated with the previous NetJSON example is this:



And by exploring depth first we get to read a line of the configuration at a time.

E.g. following the blue line from the *vlan* root to the first *leaf* we have the configuration *vlan.1.devname=eth0*



Tools

AirOS is shipped with proprietary tools that can parse the configuration file and upgrade the antenna.

cfgmtd

This tool can write and read data to the memory that persist between reboots.

ubntcfg

This tool can parse the configuration and creates the init scripts that configure the device

rc scripts

This are not commands but a collection of scripts that orchestrate the configuration process. As they are stored on the antenna they can be modified to obtain different behaviours.

- update scripts are stored in */usr/local/rc.d*
- module list is stored in */etc/startup.list*

The update process is orchestrated by the */usr/local/rc.d/rc.do.softrestart* script.

Process

AirOS mantains the device configuration in two files, both can be found in */tmp*.

- */tmp/system.cfg* the target configuration
- */tmp/running.cfg* the running configuration

If we want to upgrade the device configuration with our file we can overwrite the target configuration and runt the commands *cfgmtd -w* and */usr/local/rc.d/rc.do.softrestart save*

Full transcript of the update processs

```
cp /path/to/my/config.cfg /tmp/system.cfg
# writes the configuration to the persistent memory
cfgmtd -w /tmp/system.cfg
# initiate the configuration update
/usr/local/rc.d/rc.do.softrestart save
```

Initialization

`AirOs.__init__`(*config=None*, *native=None*, *templates=None*, *context=None*)

Parameters

- **`config`** – `dict` containing a valid **NetJSON** configuration dictionary
- **`native`** – `str` or file object representing a native configuration that will be parsed and converted to a **NetJSON** configuration dictionary
- **`templates`** – `list` containing **NetJSON** configuration dictionaries that will be used as a base for the main config
- **`context`** – `dict` containing configuration variables

Raises **`TypeError`** – raised if `config` is not of type `dict` or if `templates` is not of type `list`

Initialization example:

```
from netjsonconfig import AirOs

router = AirOs({
    "general": {
        "hostname": "MasterAntenna"
    }
})
```

If you are unsure about the meaning of the initalization parameters, read about the following basic concepts:

- *NetJSON configuration dictionary*
- *Template*
- *Context (configuration variables)*

Render method

`AirOs.`**`render`**(*files=True*)

Converts the configuration dictionary into the corresponding configuration format

Parameters **`files`** – whether to include "additional files" in the output or not; defaults to `True`

Returns string with output

Generate method

`AirOs.`**`generate`**()

Returns a `BytesIO` instance representing the configuration file

Returns in-memory configuration file, instance of `BytesIO`

Write method

`AirOs.`**`write`**(*name*, *path='./'*)

JSON method

`AirOs.`**`json`**(*validate=True*, **args*, ***kwargs*)

returns a string formatted as **NetJSON DeviceConfiguration**; performs validation before returning output;

`*args` and `*kwargs` will be passed to `json.dumps`;

Returns string

Extending the backend

Please see the *Intermediate representation* page for extending converters and adding functionalities to this backend

The configuration upgrade process

Please see the *Tools* page for information about the process and tools that upgrades the configuration on the device

Converters with defaults

NetSJON does not map explicitly to various section of the AirOS device configuration. For those section we have provided default values that should work both in `bridge` and `router` mode.

The list of "defaulted" converters follows:

- Discovery
- Dhcpc
  - `dhcpc.devname` defaults to `br0`
- Dyndns
- Httpd
- Igmpproxy
- Iptables
  - `iptables.sys.mgmt.devname` defaults to `br0`
- Netconf
  - the first interface with a `gateway` specified is the management interface in `bridge` mode
  - the first interface with a `gateway` specified is the `wan` interface in `router` mode
- Pwdog
- Radio
  - most of the configuration for the radio interface is taken from a PowerBeam `PBE-5AC-400`
- Syslog
- System
- Telnetd
- Tshaper
- Unms
- Update
- Upnpd

General settings

From the `general` property we can configure the contact and the location for a device using the `contact` and `location` properties.

The following snippet specify both contact and location:

```
{
    "type": "DeviceConfiguration",
    "general": {
        "contact": "user@example.com",
        "location": "Up on the roof"
    }
}
```

Network interface

From the `interfaces` key we can configure the device network interfaces.

AirOS supports the following types of interfaces

- **network interfaces**: may be of type `ethernet`
- **wirelesss interfaces**: must be of type `wireless`
- **bridge interfaces**: must be of type `bridge`

A network interface can be designed to be the management interfaces by setting the `role` key to `mlan` on the address chosen.

As an example here is a snippet that set the vlan `eth0.2` to be the management interface on the address `192.168.1.20`

```
{
    "interfaces": [
        {
            "name": "eth0.2",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.20",
                    "family": "ipv4",
                    "role": "mlan",
                    "mask": 24,
                    "proto": "static"
                }
            ]
        }
    ]
}
```

Ethernet

The `ethernet` interface can be configured to allow auto-negotiation and flow control with the properties `autoneg` and `flowcontrol`

As an example here is a snippet that enables both auto-negotiation and flow control

```
{
    "interfaces": [
        {
            "type": "ethernet",
            "name": "eth0",
```

```
                "autoneg": true,
                "flowcontrol": true
            }
        ]
}
```

Role

Interfaces can be assigned a `role` to mimic the web interfaces features.

As an example setting the `role` property of an address to `mlan` will add the role `mlan` to the interface configuration and set it as the management interface.

Warning: Not setting a management interface will lock you out from the web interface

Here is the snippet to set the role to `mlan`

```
{
    "interfaces": [
        {
            "type": "ethernet",
            "name": "eth0",
            "addresses": [
                {
                    "family": "ipv4",
                    "proto": "static",
                    "address": "192.168.1.1",
                    "role": "mlan"
                }
            ]
        }
    ]
}
```

This is the list of roles available for a device in `bridge` mode:

- `mlan` for the management interface

This is the list of roles available for a device in `router` mode:

- `wan` for the wan interface
- `lan` for the lan interface

GUI

As an extension to NetJSON you can use the `gui` key to set the language of the interface

The default values for this key are reported below

```
{
    "type": "DeviceConfiguration",
    "gui": {
        "language": "en_US",
```

```
    }
}
```

Netmode

AirOS v8.3 can operate in `bridge` and `router` mode (but defaults to `bridge`) and this can be specified with the `netmode` property.

```
{
    "type": "DeviceConfiguration",
    "netmode": "bridge"
}
```

NTP servers

This is an extension to the NetJSON specification.

By setting the key `ntp` property in your input you can provide the configuration for the ntp client running on the device.

```
{
    "type": "DeviceConfiguration",
    "ntp": {
        "enabled": true,
        "server": [
            "0.ubnt.pool.ntp.org"
        ]
    }
}
```

For the lazy one we provide these defaults

```
{
    "type": "DeviceConfiguration",
    "ntp": {
        "enabled": true,
        "server": [
            "0.pool.ntp.org",
            "1.pool.ntp.org",
            "2.pool.ntp.org",
            "3.pool.ntp.org"
        ]
    }
}
```

Radio

The following properties of a `Radio Object` are used during the conversion, the others have been set to safe defaults.

- `name`

Ssh

We can specify the configuration for the ssh server on the antenna using the `sshd` property.

This snippet shows how to configure the ssh server with the default values.

```
{
    "type": "DeviceConfiguration",
    "sshd": {
        "port": 22,
        "enabled": true,
        "password_auth": true
    }
}
```

And this shows how to set the authorized ssh public keys

```
{
    "type": "DeviceConfiguration",
    "sshd": {
        "keys": [
            {
                "type": "ssh-rsa",
                "key": "ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAADAQABAAABAQDBEEhdDJIbHVHIXQQ8dzH3pfmIbZjlrcIV+YkZM//
↪ezQtINTUbqolCXFsETVVwbCH6d8Pi1v1lCDgILbkOOivTIKUgG8/84yI4VLCH03CAd55IG7IFZe9e6ThT4/
↪MryH8zXKGAq5rnQSW90ashZaOEH0wNTOhkZmQ/QhduJcarevH4iZPrq5eM/ClCXzkF0I/
↪EWN89xKRrjMB09WmuYOT48n5Es08iJxwQ1gKfjk84Fy+hwMKVtOssfBGuYMBWByJwuvW5xCH3H6eVr1GhkBRrlTy6KAkc9kfAs
↪jAS2hr6kAh6cxapKENHxoAdJNvMEpdU11v6PMoOtIb edoput@hypnotoad",
                "comment": "my shh key",
                "enabled": true
            }
        ]
    }
}
```

Users

We can specify the user password as a blob divided into `salt` and `hash`.

From the antenna configuration take the user section.

```
users.status=enabled
users.1.status=enabled
users.1.name=ubnt
users.1.password=$1$yRo1tmtC$EcdoRX.JnD4VaEYgghgWg1
```

In the line `users.1.password=$1$yRo1tmtC$EcdoRX.JnD4VaEYgghgWg1` there are both the salt and the password hash in the format `$ algorithm $ salt $ hash`, e.g in the previous block `algorithm=1`, `salt=yRo1tmtC` and `hash=EcdoRX.JnD4VaEYgghgWg1`.

To specify the password in NetJSON use the `user` property.

```
{
    "type": "DeviceConfiguration",
    "user": {
```

```
        "name": "ubnt",
        "passsword": "EcdoRX.JnD4VaEYgghgWg1",
        "salt": "yRo1tmtC"
    }
}
```

WPA2

AirOS v8.3 supports both WPA2 personal (PSK+CCMP) and WPA2 enterprise (EAP+CCMP) as an authentication protocol. The only ciphers available is CCMP.

As an antenna only has one wireless network available only the first wireless interface will be used during the generation.

As an example here is a snippet that set the authentication protocol to WPA2 personal

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "mode": "station",
                "radio": "ath0",
                "ssid": "ap-ssid-example",
                "encryption": {
                    "protocol": "wpa2_personal",
                    "key": "changeme"
                }
            }
        }
    ]
}
```

And another that set the authentication protocol to WPA2 enterprise

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "mode": "station",
                "radio": "ath0",
                "ssid": "ap-ssid-example",
                "encryption": {
                    "protocol": "wpa2_enterprise",
                    "identity": "my-identity",
                    "password": "changeme",
                }
            }
        }
    ]
```

CHAPTER 4

OpenWRT Backend

The `OpenWrt` backend allows to generate OpenWRT compatible configurations.

Note: This backend purposely generates only named UCI blocks.

UCI stands for Unified Configuration Interface and it is the default configuration system installed on OpenWRT and its fork LEDE.

Initialization

`OpenWrt.__init__`(*config=None*, *native=None*, *templates=None*, *context=None*)

Parameters

- **`config`** – `dict` containing a valid **NetJSON** configuration dictionary
- **`native`** – `str` or file object representing a native configuration that will be parsed and converted to a **NetJSON** configuration dictionary
- **`templates`** – `list` containing **NetJSON** configuration dictionaries that will be used as a base for the main config
- **`context`** – `dict` containing configuration variables

Raises **`TypeError`** – raised if `config` is not of type `dict` or if `templates` is not of type `list`

If you are unsure about the meaning of the initalization parameters, read about the following basic concepts:

- *NetJSON configuration dictionary*
- *Backend*
- *Template*
- *Context (configuration variables)*

Initialization example (forward conversion):

```
from netjsonconfig import OpenWrt

router = OpenWrt({
    "general": {
        "hostname": "HomeRouter"
    }
})
```

Initialization example (backward conversion):

```
from netjsonconfig import OpenWrt

router = OpenWrt(native=open('./openwrt-config.tar.gz'))
```

Render method

`OpenWrt.`**`render`**(*files=True*)

Converts the configuration dictionary into the corresponding configuration format

Parameters **`files`** – whether to include "additional files" in the output or not; defaults to `True`

Returns string with output

Code example:

```
from netjsonconfig import OpenWrt

o = OpenWrt({
    "interfaces": [
        {
            "name": "eth0.1",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.2",
                    "gateway": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                },
                {
                    "address": "192.168.2.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                },
                {
                    "address": "fd87::2",
                    "gateway": "fd87::1",
                    "mask": 64,
                    "proto": "static",
                    "family": "ipv6"
                }
            ]
        }
```

```
    ]
})
print(o.render())
```

Will return the following output:

```
package network

config interface 'eth0_1'
        option gateway '192.168.1.1'
        option ifname 'eth0.1'
        option ip6addr 'fd87::2/64'
        option ip6gw 'fd87::1'
        list ipaddr '192.168.1.2/24'
        list ipaddr '192.168.2.1/24'
        option proto 'static'
```

Generate method

`OpenWrt.generate()`

Returns a `BytesIO` instance representing an in-memory tar.gz archive containing the native router configuration.

Returns in-memory tar.gz archive, instance of `BytesIO`

Example:

```
>>> import tarfile
>>> from netjsonconfig import OpenWrt
>>>
>>> o = OpenWrt({
...     "interfaces": [
...         {
...             "name": "eth0",
...             "type": "ethernet",
...             "addresses": [
...                 {
...                     "proto": "dhcp",
...                     "family": "ipv4"
...                 }
...             ]
...         }
...     ]
... })
>>> stream = o.generate()
>>> print(stream)
<_io.BytesIO object at 0x7fd2287fb410>
>>> tar = tarfile.open(fileobj=stream, mode='r:gz')
>>> print(tar.getmembers())
[<TarInfo 'etc/config/network' at 0x7fd228790250>]
```

As you can see from this example, the `generate` method does not write to disk, but returns an instance of `io.BytesIO` which contains a tar.gz file object with the following file structure:

```
/etc/config/network
```

The configuration archive can then be written to disk, served via HTTP or uploaded directly on the OpenWRT router where it can be finally "restored" with `sysupgrade`:

```
sysupgrade -r <archive>
```

Note that `sysupgrade -r` does not apply the configuration, to do this you have to reload the services manually or reboot the router.

Note: the `generate` method intentionally sets the timestamp of the tar.gz archive and its members to `0` in order to facilitate comparing two different archives: setting the timestamp would infact cause the checksum to be different each time even when contents of the archive are identical.

Write method

`OpenWrt.`**`write`**(*name*, *path='./'*)

Like `generate` but writes to disk.

Parameters

- **name** – file name, the tar.gz extension will be added automatically
- **path** – directory where the file will be written to, defaults to `./`

Returns None

Example:

```
>>> import tarfile
>>> from netjsonconfig import OpenWrt
>>>
>>> o = OpenWrt({
...     "interfaces": [
...         {
...             "name": "eth0",
...             "type": "ethernet",
...             "addresses": [
...                 {
...                     "proto": "dhcp",
...                     "family": "ipv4"
...                 }
...             ]
...         }
...     ]
... })
>>> o.write('dhcp-router', path='/tmp/')
```

Will write the configuration archive in `/tmp/dhcp-router.tar.gz`.

Parse method

`OpenWrt.`**`parse`**(*native*)

Parses a native configuration and converts it to a NetJSON configuration dictionary

This method is automatically called when initializing the backend with the `native` argument:

```
from netjsonconfig import OpenWrt

router = OpenWrt(native=open('./openwrt-config.tar.gz'))
```

The argument passed to `native` can be a string containing a dump obtained via `uci export`, or a file object (real file or `BytesIO` instance) representing a configuration archive in tar.gz format typically used in OpenWRT/LEDE.

JSON method

`OpenWrt.`**`json`**(*validate=True*, **args*, ***kwargs*)

returns a string formatted as **NetJSON DeviceConfiguration**; performs validation before returning output;

`*args` and `*kwargs` will be passed to `json.dumps`;

Returns string

Code example:

```
>>> from netjsonconfig import OpenWrt
>>>
>>> router = OpenWrt({
...     "general": {
...         "hostname": "HomeRouter"
...     }
... })
>>> print(router.json(indent=4))
{
    "type": "DeviceConfiguration",
    "general": {
        "hostname": "HomeRouter"
    }
}
```

General settings

The general settings reside in the `general` key of the *configuration dictionary*, which follows the NetJSON General object definition (see the link for the detailed specification).

Currently only the `hostname` option is processed by this backend.

General object extensions

In addition to the default *NetJSON General object options*, the `OpenWrt` backend also supports the following custom options:

key name	type	function
`timezone`	string	one of the allowed timezone values (first element of each tuple)

General settings example

The following *configuration dictionary*:

```
{
    "general": {
        "hostname": "routerA",
        "timezone": "UTC",
        "ula_prefix": "fd8e:f40a:6701::/48"
    }
}
```

Will be rendered as follows:

```
package system

config system 'system'
        option hostname 'routerA'
        option timezone 'UTC'
        option zonename 'UTC'

package network

config globals 'globals'
        option ula_prefix 'fd8e:f40a:6701::/48'
```

Network interfaces

The network interface settings reside in the `interfaces` key of the *configuration dictionary*, which must contain a list of NetJSON interface objects (see the link for the detailed specification).

There are 3 main type of interfaces:

- **network interfaces**: may be of type `ethernet`, `virtual`, `loopback` or `other`
- **wireless interfaces**: must be of type `wireless`
- **bridge interfaces**: must be of type `bridge`

Interface object extensions

In addition to the default *NetJSON Interface object options*, the `OpenWrt` backend also supports the following custom options for every type of interface:

key name	type	allowed values
`network`	string	logical interface name (UCI specific)

In the following sections some examples of the most common use cases are shown.

Loopback interface example

The following *configuration dictionary*:

```
{
    "interfaces": [
        {
            "name": "lo",
            "type": "loopback",
            "addresses": [
```

```
                {
                    "address": "127.0.0.1",
                    "mask": 8,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'lo'
        option ifname 'lo'
        option ipaddr '127.0.0.1'
        option netmask '255.0.0.0'
        option proto 'static'
```

Dualstack (IPv4 & IPv6)

The following *configuration dictionary*:

```
{
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "family": "ipv4",
                    "proto": "static",
                    "address": "10.27.251.1",
                    "mask": 24
                },
                {
                    "family": "ipv6",
                    "proto": "static",
                    "address": "fdb4:5f35:e8fd::1",
                    "mask": 48
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'eth0'
    option ifname 'eth0'
    option ip6addr 'fdb4:5f35:e8fd::1/48'
    option ipaddr '10.27.251.1'
```

```
    option netmask '255.255.255.0'
    option proto 'static'
```

DNS servers and search domains

DNS servers can be set using `dns_servers`, while search domains can be set using `dns_search`.

If specified, these values will be automatically added in every interface which has at least one static ip address; interfaces which have no ip address configured or are using dynamic ip address configuration won't get the `dns` option in the UCI output, eg:

```
{
    "dns_servers": ["10.11.12.13", "8.8.8.8"],
    "dns_search": ["openwisp.org", "netjson.org"],
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "address": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        },
        # the following interface has DHCP enabled
        # and it won't contain the dns setting
        {
            "name": "eth1",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ]
        },
        # the following VLAN interface won't get
        # the dns nor the dns_search settings
        {
            "name": "eth1.31",
            "type": "ethernet"
        }
    ]
}
```

Will return the following UCI output:

```
package network

config interface 'eth0'
        option dns '10.11.12.13 8.8.8.8'
        option dns_search 'openwisp.org netjson.org'
        option ifname 'eth0'
```

```
        option ipaddr '192.168.1.1'
        option netmask '255.255.255.0'
        option proto 'static'

config interface 'eth1'
        option dns_search 'openwisp.org netjson.org'
        option ifname 'eth1'
        option proto 'dhcp'

config interface 'eth1_31'
        option ifname 'eth1.31'
        option proto 'none'
```

DHCP ipv6 ethernet interface

The following *configuration dictionary*:

```
{
    "interfaces": [
        {
            "name": "eth0",
            "network": "lan",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv6"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'lan'
        option ifname 'eth0'
        option proto 'dchpv6'
```

Using different protocols

OpenWRT and LEDE support many protocols (pppoe, pppoa, pptp, l2tp, ecc) and the list of supported protocols evolves over time.

OpenWISP and netjsonconfig try to stay out of your way by leaving you maximum flexibility to use any protocol and any configuration option you may need, just set `type` to `other`, then proceed by setting *proto* and any other configuration option according to your needs, see the example below.

PPPoE proto example

The following configuration dictionary:

```
{
    "interfaces": [
        {
            "type": "other",
            "name": "eth0",
            "network": "wan",
            "proto": "pppoe",
            "username": "<username>",
            "password": "<password>"
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'wan'
        option ifname 'eth0'
        option password '<password>'
        option proto 'ppoe'
        option username '<username>'
```

Bridge settings

Interfaces of type `bridge` can contain a few options that are specific for network bridges:

- `bridge_members`: interfaces that are members of the bridge
- `stp`: spanning tree protocol

The `OpenWrt` backend NetJSON extensions for bridge interfaces:

key name	type	default	allowed values
`igmp_snooping`	boolean	`True`	sets the `multicast_snooping` kernel setting for a bridge

Bridge interface example

The following *configuration dictionary*:

```
{
    "interfaces": [
        {
            "name": "eth0.1",
            "network": "lan",
            "type": "ethernet"
        },
        {
            "name": "eth0.2",
            "network": "wan",
            "type": "ethernet"
        },
        {
            "name": "lan_bridge",  # will be named "br-lan_bridge" by OpenWRT
            "type": "bridge",
```

```
            "stp": True,  # enable spanning tree protocol
            "igmp_snooping": True,  # enable imgp snooping
            "bridge_members": [
                "eth0.1",
                "eth0.2"
            ],
            "addresses": [
                {
                    "address": "172.17.0.2",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'lan'
        option ifname 'eth0.1'
        option proto 'none'

config interface 'wan'
        option ifname 'eth0.2'
        option proto 'none'

config interface 'lan_bridge'
        option ifname 'eth0.1 eth0.2'
        option igmp_snooping '1'
        option ipaddr '172.17.0.2'
        option netmask '255.255.255.0'
        option proto 'static'
        option stp '1'
        option type 'bridge'
```

Wireless settings

Interfaces of type `wireless` may contain a lot of different combination of settings to configure wireless connectivity: from simple access points, to 802.1x authentication, 802.11s mesh networks, adhoc mesh networks, WDS repeaters and much more.

The `OpenWrt` backend NetJSON extensions for wireless interfaces:

key name	type	default	allowed values
`network`	array	`[]`	attached networks; if left blank will be automatically determined

Some extensions are applicable only when `mode` is `access_point`:

key name	type	default	allowed values
`wmm`	boolean	`True`	enables WMM (802.11e) support
`isolate`	boolean	`False`	isolate wireless clients from one another
`macfilter`	string	`disable`	ACL policy, accepts: “disable”, “allow” and “deny”
`maclist`	array	`[]`	mac addresses filtered according to macfilter policy

These extensions must be used the `wireless` object of a wireless interface eg:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "myWiFi",
                # OpenWrt backend NetJSON extensions
                "wmm": True,
                "isolate": True
            }
        }
    ]
}
```

The same applies for custom configuration options not included in the `OpenWrt` backend schema:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "myWiFi",
                # custom configuration options not defined
                # in the OpenWrt backend schema
                "beacon_int": 200,
                "noscan": True,
                "custom1": "made-up-for-example-purposes",
            }
        }
    ]
}
```

In the following sections some examples of the most common use cases are shown.

Wireless access point

The following *configuration dictionary* represent one of the most common wireless access point configuration:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
```

```
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "myWiFi",
                "wmm": True,  # 802.11e
                "isolate": True  # client isolation
            }
        }
    ]
}
```

UCI output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option ifname 'wlan0'
        option isolate '1'
        option mode 'ap'
        option network 'wlan0'
        option ssid 'myWiFi'
        option wmm '1'
```

Note: the `network` option of the `wifi-iface` directive is filled in automatically but can be overridden if needed by setting the `network` option in the `wireless` section of the *configuration dictionary*. The next example shows how to do this.

Wireless attached to a different network

In some cases you might want to attach a wireless interface to a different network, for example, you might want to attach a wireless interface to a bridge:

```
{
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet"
        },
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "wifi service",
                # the wireless interface will be attached to the "lan" network
                "network": ["lan"]
```

```
            }
        },
        {
            "name": "lan",  # the bridge will be named br-lan by OpenWRT
            "type": "bridge",
            "bridge_members": [
                "eth0",
                "wlan0"
            ],
            "addresses": [
                {
                    "address": "192.168.0.2",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config interface 'eth0'
        option ifname 'eth0'
        option proto 'none'

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

config interface 'lan'
        option ifname 'eth0 wlan0'
        option ipaddr '192.168.0.2'
        option netmask '255.255.255.0'
        option proto 'static'
        option type 'bridge'

package wireless

config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option ifname 'wlan0'
        option mode 'ap'
        option network 'lan'
        option ssid 'wifi service'
```

Wireless access point with macfilter ACL

The `OpenWrt` backend supports a custom NetJSON extension for wireless access point interfaces: `macfilter` (read more about `macfilter` and `maclist` on the OpenWRT documentation for Wireless configuration).

In the following example we ban two mac addresses from connecting to a wireless access point:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "MyWifiAP",
                "macfilter": "deny",
                "maclist": [
                    "E8:94:F6:33:8C:1D",
                    "42:6c:8f:95:0f:00"
                ]
            }
        }
    ]
}
```

UCI output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option ifname 'wlan0'
        option macfilter 'deny'
        list maclist 'E8:94:F6:33:8C:1D'
        list maclist '42:6c:8f:95:0f:00'
        option mode 'ap'
        option network 'wlan0'
        option ssid 'MyWifiAP'
```

Wireless mesh (802.11s) example

Setting up **802.11s** interfaces is fairly simple, in the following example we bridge `eth0` with `mesh0`, the latter being a layer2 802.11s wireless interface.

Note: in 802.11s mesh mode the `ssid` property is not required, while `mesh_id` is mandatory.

```
{
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet"
        },
        {
            "name": "mesh0",
```

```
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "802.11s",
                "mesh_id": "ninux",
                "network": ["lan"]
            }
        },
        {
            "name": "lan",
            "type": "bridge",
            "bridge_members": ["eth0", "mesh0"],
            "addresses": [
                {
                    "address": "192.168.0.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

UCI output:

```
package network

config interface 'eth0'
        option ifname 'eth0'
        option proto 'none'

config interface 'mesh0'
        option ifname 'mesh0'
        option proto 'none'

config interface 'lan'
        option ifname 'eth0 mesh0'
        option ipaddr '192.168.0.1'
        option netmask '255.255.255.0'
        option proto 'static'
        option type 'bridge'

package wireless

config wifi-iface 'wifi_mesh0'
        option device 'radio0'
        option ifname 'mesh0'
        option mesh_id 'ninux'
        option mode 'mesh'
        option network 'lan'
```

Wireless mesh (adhoc) example

In wireless adhoc mode, the `bssid` property is required.

The following example:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "ssid": "freifunk",
                "mode": "adhoc",
                "bssid": "02:b8:c0:00:00:00"
            }
        }
    ]
}
```

Will result in:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option bssid '02:b8:c0:00:00:00'
        option device 'radio0'
        option ifname 'wlan0'
        option mode 'adhoc'
        option network 'wlan0'
        option ssid 'freifunk'
```

WDS repeater example

In the following example we show how to configure a WDS station and repeat the signal:

```
{
    "interfaces": [
        # client
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "mode": "station",
                "radio": "radio0",
                "network": ["wds_bridge"],
                "ssid": "FreeRomaWifi",
                "bssid": "C0:4A:00:2D:05:FD",
                "wds": True
            }
        },
        # repeater access point
        {
            "name": "wlan1",
            "type": "wireless",
```

```
            "wireless": {
                "mode": "access_point",
                "radio": "radio1",
                "network": ["wds_bridge"],
                "ssid": "FreeRomaWifi"
            }
        },
        # WDS bridge
        {
            "name": "br-wds",
            "network": "wds_bridge",
            "type": "bridge",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4"
                }
            ],
            "bridge_members": [
                "wlan0",
                "wlan1",
            ]
        }
    ]
}
```

Will result in:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

config interface 'wlan1'
        option ifname 'wlan1'
        option proto 'none'

config interface 'br_wds'
        option ifname 'wlan0 wlan1'
        option network 'wds_bridge'
        option proto 'dhcp'
        option type 'bridge'

package wireless

config wifi-iface 'wifi_wlan0'
        option bssid 'C0:4A:00:2D:05:FD'
        option device 'radio0'
        option ifname 'wlan0'
        option mode 'sta'
        option network 'wds_bridge'
        option ssid 'FreeRomaWifi'
        option wds '1'

config wifi-iface 'wifi_wlan1'
        option device 'radio1'
        option ifname 'wlan1'
```

```
        option mode 'ap'
        option network 'wds_bridge'
        option ssid 'FreeRomaWifi'
```

WPA2 Personal (Pre-Shared Key)

The following example shows a typical wireless access point using *WPA2 Personal (Pre-Shared Key)* encryption:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "wpa2-personal",
                "encryption": {
                    "protocol": "wpa2_personal",
                    # possible cipher values are:
                    #   "auto", "tkip", "ccmp", and "tkip+ccmp"
                    "cipher": "tkip+ccmp",
                    "key": "passphrase012345"
                }
            }
        }
    ]
}
```

UCI output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option device 'radio0'
        option encryption 'psk2+tkip+ccmp'
        option ifname 'wlan0'
        option key 'passphrase012345'
        option mode 'ap'
        option network 'wlan0'
        option ssid 'wpa2-personal'
```

WPA2 Enterprise (802.1x) ap

The following example shows a typical wireless access point using *WPA2 Enterprise (802.1x)* security on **OpenWRT**, you can use this type of configuration for networks like eduroam:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "eduroam",
                "encryption": {
                    "protocol": "wpa2_enterprise",
                    "cipher": "auto",
                    "key": "radius_secret",
                    "server": "192.168.0.1",
                    "port": 1812,
                    "acct_server": "192.168.0.2",
                    "acct_port": 1813,
                    "nasid": "hostname"
                }
            }
        }
    ]
}
```

UCI Output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option acct_port '1813'
        option acct_server '192.168.0.2'
        option device 'radio0'
        option encryption 'wpa2'
        option ifname 'wlan0'
        option key 'radius_secret'
        option mode 'ap'
        option nasid 'hostname'
        option network 'wlan0'
        option port '1812'
        option server '192.168.0.1'
        option ssid 'eduroam'
```

WPA2 Enterprise (802.1x) client

WPA2 Enterprise (802.1x) client example:

```
{
    "interfaces": [
        {
            "name": "wlan0",
```

```
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "station",
                "ssid": "enterprise-client",
                "bssid": "00:26:b9:20:5f:09",
                "encryption": {
                    "protocol": "wpa2_enterprise",
                    "cipher": "auto",
                    "eap_type": "tls",
                    "identity": "test-identity",
                    "password": "test-password",
                }
            }
        }
    ]
}
```

UCI Output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        option bssid '00:26:b9:20:5f:09'
        option device 'radio0'
        option eap_type 'tls'
        option encryption 'wpa2'
        option identity 'test-identity'
        option ifname 'wlan0'
        option mode 'sta'
        option network 'wlan0'
        option password 'test-password'
        option ssid 'enterprise-client'
```

Radio settings

The radio settings reside in the `radio` key of the *configuration dictionary*, which must contain a list of NetJSON radio objects (see the link for the detailed specification).

Radio object extensions

In addition to the default *NetJSON Radio object options*, the `OpenWrt` backend also requires setting the following additional options for each radio in the list:

key name	type	allowed values
`driver`	string	mac80211, madwifi, ath5k, ath9k, broadcom
`protocol`	string	802.11a, 802.11b, 802.11g, 802.11n, 802.11ac

Radio example

The following *configuration dictionary*:

```
{
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "driver": "mac80211",
            "protocol": "802.11n",
            "channel": 11,
            "channel_width": 20,
            "tx_power": 5,
            "country": "IT"
        },
        {
            "name": "radio1",
            "phy": "phy1",
            "driver": "mac80211",
            "protocol": "802.11n",
            "channel": 36,
            "channel_width": 20,
            "tx_power": 4,
            "country": "IT"
        }
    ]
}
```

Will be rendered as follows:

```
package wireless

config wifi-device 'radio0'
        option channel '11'
        option country 'IT'
        option htmode 'HT20'
        option hwmode '11g'
        option phy 'phy0'
        option txpower '5'
        option type 'mac80211'

config wifi-device 'radio1'
        option channel '36'
        option country 'IT'
        option disabled '0'
        option htmode 'HT20'
        option hwmode '11a'
        option phy 'phy1'
        option txpower '4'
        option type 'mac80211'
```

Automatic channel selection example

If you need to use the "automatic channel selection" feature of OpenWRT, you must set the channel to `0` and, unless you are using neither **802.11n** nor **802.11ac**, you must set the `hwmode` property to tell OpenWRT which band to use (11g for 2.4 Ghz, 11a for 5 GHz).

The following example sets "automatic channel selection" for two radios, the first radio uses **802.11n** in the 2.4 GHz band, while the second uses **802.11ac** in the 5 GHz band.

```
{
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "driver": "mac80211",
            "protocol": "802.11n",
            "channel": 0,  # 0 stands for auto
            "hwmode": "11g",  # must set this explicitly, 11g means 2.4 GHz band
            "channel_width": 20
        },
        {
            "name": "radio1",
            "phy": "phy1",
            "driver": "mac80211",
            "protocol": "802.11ac",
            "channel": 0,  # 0 stands for auto
            "hwmode": "11a",  # must set this explicitly, 11a means 5 GHz band
            "channel_width": 80
        }
    ]
}
```

UCI output:

```
package wireless

config wifi-device 'radio0'
        option channel 'auto'
        option htmode 'HT20'
        option hwmode '11g'
        option phy 'phy0'
        option type 'mac80211'

config wifi-device 'radio1'
        option channel 'auto'
        option htmode 'VHT80'
        option hwmode '11a'
        option phy 'phy1'
        option type 'mac80211'
```

802.11ac example

In the following example we show how to configure an *802.11ac* capable radio:

```
{
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "driver": "mac80211",
            "protocol": "802.11ac",
            "channel": 36,
            "channel_width": 80,
```

```
        }
    ]
}
```

UCI output:

```
package wireless

config wifi-device 'radio0'
        option channel '36'
        option htmode 'VHT80'
        option hwmode '11a'
        option phy 'phy0'
        option type 'mac80211'
```

Static Routes

The static routes settings reside in the `routes` key of the *configuration dictionary*, which must contain a list of NetJSON Static Route objects (see the link for the detailed specification).

Static route object extensions

In addition to the default *NetJSON Route object options*, the `OpenWrt` backend also allows to define the following optional settings:

key name	type	default	description
`type`	string	`unicast`	unicast, local, broadcast, multicast, unreachable prohibit, blackhole, anycast
`mtu`	string	`None`	MTU for route, only numbers are allowed
`table`	string	`None`	Routing table id, only numbers are allowed
`onlink`	boolean	`False`	When enabled, gateway is on link even if the gateway does not match any interface prefix

Static route example

The following *configuration dictionary*:

```
{
    "routes": [
        {
            "device": "eth1",
            "destination": "192.168.4.1/24",
            "next": "192.168.2.2",
            "cost": 2,
            "source": "192.168.1.10",
            "table": "2",
            "onlink": True,
            "mtu": "1450"
        },
        {
            "device": "eth1",
            "destination": "fd89::1/128",
```

```
            "next": "fd88::1",
            "cost": 0,
        }
    ]
}
```

Will be rendered as follows:

```
package network

config route 'route1'
        option gateway '192.168.2.2'
        option interface 'eth1'
        option metric '2'
        option mtu '1450'
        option netmask '255.255.255.0'
        option onlink '1'
        option source '192.168.1.10'
        option table '2'
        option target '192.168.4.1'

config route6 'route2'
        option gateway 'fd88::1'
        option interface 'eth1'
        option metric '0'
        option target 'fd89::1/128'
```

Policy routing

The policy routing settings reside in the `ip_rule` key of the *configuration dictionary*, which is a custom NetJSON extension not present in the original NetJSON RFC.

The `ip_rule` key must contain a list of rules, each rule allows the following options:

key name	type
`in`	string
`out`	string
`src`	string
`tos`	string
`mark`	string
`invert`	boolean
`lookup`	string
`goto`	integer
`action`	string

For the function and meaning of each key consult the relevant OpenWrt documentation about rule directives.

Policy routing example

The following *configuration dictionary*:

```
{
    "ip_rules": [
        {
```

```
            "in": "eth0",
            "out": "eth1",
            "src": "192.168.1.0/24",
            "dest": "192.168.2.0/24",
            "tos": 2,
            "mark": "0x0/0x1",
            "invert": True,
            "lookup": "0",
            "action": "blackhole"
        },
        {
            "src": "192.168.1.0/24",
            "dest": "192.168.3.0/24",
            "goto": 0
        },
        {
            "in": "vpn",
            "dest": "fdca:1234::/64",
            "action": "prohibit"
        },
        {
            "in": "vpn",
            "src": "fdca:1235::/64",
            "action": "prohibit"
        }
    ]
}
```

Will be rendered as follows:

```
package network

config rule 'rule1'
        option action 'blackhole'
        option dest '192.168.2.0/24'
        option in 'eth0'
        option invert '1'
        option lookup '0'
        option mark '0x0/0x1'
        option out 'eth1'
        option src '192.168.1.0/24'
        option tos '2'

config rule 'rule2'
        option dest '192.168.3.0/24'
        option goto '0'
        option src '192.168.1.0/24'

config rule6 'rule3'
        option action 'prohibit'
        option dest 'fdca:1234::/64'
        option in 'vpn'

config rule6 'rule4'
        option action 'prohibit'
        option in 'vpn'
        option src 'fdca:1235::/64'
```

Programmable switch settings

The programmable switch settings reside in the `switch` key of the *configuration dictionary*, which is a custom NetJSON extension not present in the original NetJSON RFC.

The `switch` key must contain a list of dictionaries, all the following keys are required:

key name	type
`name`	string
`reset`	boolean
`enable_vlan`	boolean
`vlan`	list

The elements of the `vlan` list must be dictionaries, all the following keys are required:

key name	type
`device`	string
`reset`	boolean
`vlan`	integer
`ports`	string

For the function and meaning of each key consult the relevant OpenWrt documentation about switch directives.

Switch example

The following *configuration dictionary*:

```
{
    "switch": [
        {
            "name": "switch0",
            "reset": True,
            "enable_vlan": True,
            "vlan": [
                {
                    "device": "switch0",
                    "vlan": 1,
                    "ports": "0t 2 3 4 5"
                },
                {
                    "device": "switch0",
                    "vlan": 2,
                    "ports": "0t 1"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config switch 'switch0'
        option enable_vlan '1'
        option name 'switch0'
        option reset '1'
```

```
config switch_vlan 'switch0_vlan1'
        option device 'switch0'
        option ports '0t 2 3 4 5'
        option vid '1'
        option vlan '1'

config switch_vlan 'switch0_vlan2'
        option device 'switch0'
        option ports '0t 1'
        option vid '2'
        option vlan '2'
```

Overriding or disabling `vid` UCI option

The OpenWRT/LEDE UCI `vid` option of `switch_vlan` sections is automatically inferred from the `vlan` number, although it's possible to override it or disable it if needed:

```
{
    "switch": [
        {
            "name": "switch0",
            "reset": True,
            "enable_vlan": True,
            "vlan": [
                {
                    "device": "switch0",
                    "vlan": 1,
                    "vid": 110,  # manual override
                    "ports": "0t 2 3 4 5"
                },
                {
                    "device": "switch0",
                    "vlan": 2,
                    # ``None`` or empty string will remove
                    # ``vid`` output from the UCI result
                    "vid": None,
                    "ports": "0t 1"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

config switch 'switch0'
        option enable_vlan '1'
        option name 'switch0'
        option reset '1'

config switch_vlan 'switch0_vlan1'
        option device 'switch0'
        option ports '0t 2 3 4 5'
```

```
        option vid '110'
        option vlan '1'

config switch_vlan 'switch0_vlan2'
        option device 'switch0'
        option ports '0t 1'
        option vlan '2'
```

NTP settings

The Network Time Protocol settings reside in the `ntp` key of the *configuration dictionary*, which is a custom NetJSON extension not present in the original NetJSON RFC.

The `ntp` key must contain a dictionary, the allowed options are:

key name	type	function
`enabled`	boolean	ntp client enabled
`enable_server`	boolean	ntp server enabled
`server`	list	list of ntp servers

NTP settings example

The following *configuration dictionary*:

```
{
    "ntp": {
    "enabled": True,
    "enable_server": False,
    "server": [
        "0.openwrt.pool.ntp.org",
        "1.openwrt.pool.ntp.org",
        "2.openwrt.pool.ntp.org",
        "3.openwrt.pool.ntp.org"
    ]
}
```

Will be rendered as follows:

```
package system

config timeserver 'ntp'
        list server '0.openwrt.pool.ntp.org'
        list server '1.openwrt.pool.ntp.org'
        list server '2.openwrt.pool.ntp.org'
        list server '3.openwrt.pool.ntp.org'
        option enable_server '0'
        option enabled '1'
```

LED settings

The led settings reside in the `led` key of the *configuration dictionary*, which is a custom NetJSON extension not present in the original NetJSON RFC.

The `led` key must contain a list of dictionaries, the allowed options are:

key name	type
`name`	string
`default`	boolean
`dev`	string
`sysfs`	string
`trigger`	string
`delayoff`	integer
`delayon`	integer
`interval`	integer
`message`	string
`mode`	string

The required keys are:

- `name`
- `sysfs`
- `trigger`

For the function and meaning of each key consult the relevant OpenWrt documentation about led directives.

LED settings example

The following *configuration dictionary*:

```
{
    "led": [
        {
            "name": "USB1",
            "sysfs": "tp-link:green:usb1",
            "trigger": "usbdev",
            "dev": "1-1.1",
            "interval": 50
        },
        {
            "name": "USB2",
            "sysfs": "tp-link:green:usb2",
            "trigger": "usbdev",
            "dev": "1-1.2",
            "interval": 50
        },
        {
            "name": "WLAN2G",
            "sysfs": "tp-link:blue:wlan2g",
            "trigger": "phy0tpt"
        }
    ]
}
```

Will be rendered as follows:

```
package system

config led 'led_usb1'
        option dev '1-1.1'
```

```
        option interval '50'
        option name 'USB1'
        option sysfs 'tp-link:green:usb1'
        option trigger 'usbdev'

config led 'led_usb2'
        option dev '1-1.2'
        option interval '50'
        option name 'USB2'
        option sysfs 'tp-link:green:usb2'
        option trigger 'usbdev'

config led 'led_wlan2g'
        option name 'WLAN2G'
        option sysfs 'tp-link:blue:wlan2g'
        option trigger 'phy0tpt'
```

Including custom options

It is very easy to add configuration options that are not explicitly defined in the schema of the `OpenWrt` backend.

For example, in some cases you may need to define a "ppp" interface, which can use quite a few properties that are not defined in the schema:

```
from netjsonconfig import OpenWrt

o = OpenWrt({
    "interfaces": [
        {
            "name": "ppp0",
            "type": "other",
            "proto": "ppp",
            "device": "/dev/usb/modem1",
            "username": "user1",
            "password": "pwd0123",
            "keepalive": 3,
            "ipv6": True
        }
    ]
})
print(o.render())
```

UCI output:

```
package network

config interface 'ppp0'
        option device '/dev/usb/modem1'
        option ifname 'ppp0'
        option ipv6 '1'
        option keepalive '3'
        option password 'pwd0123'
        option proto 'ppp'
        option username 'user1'
```

Including custom lists

Under specific circumstances, OpenWRT allows adding configuration options in the form of lists. Many of these UCI options are not defined in the *JSON-Schema* of the `OpenWrt` backend, but the schema allows adding custom properties.

The `OpenWrt` backend recognizes list options for the following sections:

- interface settings
- ip address settings
- wireless settings
- radio settings

Interface list setting example

The following example shows how to set a list of `ip6class` options:

```
o = OpenWrt({
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "ip6class": ["wan6", "backbone"]
        }
    ]
})
print(o.render())
```

UCI Output:

```
package network

config interface 'eth0'
        option ifname 'eth0'
        list ip6class 'wan6'
        list ip6class 'backbone'
        option proto 'none'
```

Address list setting example

The following example shows how to set a list of dhcp `reqopts` settings:

```
o = OpenWrt({
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "addresses": [
                {
                    "proto": "dhcp",
                    "family": "ipv4",
                    "reqopts": ["43", "54"]
                }
```

```
            ]
        }
    ]
})
print(o.render())
```

UCI Output:

```
package network

config interface 'eth0'
        option ifname 'eth0'
        option proto 'dhcp'
        list reqopts '43'
        list reqopts '54'
```

Radio list setting example

The following example shows how to set a list of advanced capabilities supported by the radio using `ht_capab`:

```
o = OpenWrt({
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "driver": "mac80211",
            "protocol": "802.11n",
            "channel": 1,
            "channel_width": 20,
            "ht_capab": ["SMPS-STATIC", "SHORT-GI-20"]
        }
    ]
})
print(o.render())
```

UCI output:

```
package wireless

config wifi-device 'radio0'
        option channel '1'
        list ht_capab 'SMPS-STATIC'
        list ht_capab 'SHORT-GI-20'
        option htmode 'HT20'
        option hwmode '11g'
        option phy 'phy0'
        option type 'mac80211'
```

Wireless list setting example

The following example shows how to set the supported basic rates of a wireless interface using `basic_rate`:

```
o = OpenWrt({
    "interfaces": [
```

```
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "open",
                "basic_rate": ["6000", "9000"]
            }
        }
    ]
})
print(o.render())
```

UCI output:

```
package network

config interface 'wlan0'
        option ifname 'wlan0'
        option proto 'none'

package wireless

config wifi-iface 'wifi_wlan0'
        list basic_rate '6000'
        list basic_rate '9000'
        option device 'radio0'
        option ifname 'wlan0'
        option mode 'ap'
        option network 'wlan0'
        option ssid 'open'
```

Including additional files

The `OpenWrt` backend supports inclusion of arbitrary plain text files through the `files` key of the *configuration dictionary*. The value of the `files` key must be a list in which each item is a dictionary representing a file, each dictionary is structured as follows:

key name	type	required	function
`path`	string	yes	filesystem path, will be encoded in the tar.gz archive
`contents`	string	yes	plain text contents of the file, new lines must be encoded as `\n`
`mode`	string	yes	filesystem permissions, defaults to `0644`

The `files` key of the *configuration dictionary* is a custom NetJSON extension not present in the original NetJSON RFC.

Warning: The files are included in the output of the `render` method unless you pass `files=False`, eg: `openwrt.render(files=False)`

Plain file example

The following example code will generate an archive with one file in `/etc/crontabs/root`:

```
from netjsonconfig import OpenWrt

o = OpenWrt({
    "files": [
        {
            "path": "/etc/crontabs/root",
            "mode": "0644",
            # new lines must be escaped with ``\n``
            "contents": '* * * * * echo "test" > /etc/testfile\n'
                        '* * * * * echo "test2" > /etc/testfile2'
        }
    ]
})
o.generate()
```

Executable script file example

The following example will create an executable shell script:

```
o = OpenWrt({
    "files": [
        {
            "path": "/bin/hello_world",
            "mode": "0755",
            "contents": "#!/bin/sh\n"
                        "echo 'Hello world'"
        }
    ]
})
o.generate()
```

OpenVPN

This backend includes the schema of the `OpenVpn` backend, inheriting its features.

For details regarding the OpenVPN schema please see *OpenVPN backend schema*.

Schema additions

The `OpenWrt` backend adds a few properties to the OpenVPN schema, see below.

key name	type	default	allowed values
`disabled`	boolean	`False`	

OpenVPN example

The following *configuration dictionary*:

```
{
    "openvpn": [
        {
            "ca": "ca.pem",
```

```
            "cert": "cert.pem",
            "dev": "tap0",
            "dev_type": "tap",
            "dh": "dh.pem",
            "disabled": False,
            "key": "key.pem",
            "mode": "server",
            "name": "test-vpn-server",
            "proto": "udp",
            "tls_server": True
        }
    ]
}
```

Will be rendered as follows:

```
package openvpn

config openvpn 'test_vpn_server'
        option ca 'ca.pem'
        option cert 'cert.pem'
        option dev 'tap0'
        option dev_type 'tap'
        option dh 'dh.pem'
        option enabled '1'
        option key 'key.pem'
        option mode 'server'
        option proto 'udp'
        option tls_server '1'
```

All the other settings

Do you need to include some configuration directives that are not defined in the NetJSON spec nor in the schema of the `OpenWrt` backend? **Don't panic!**

Because netjsonconfig aims to be very flexible, it ships code that will try to render extra parts of the *configuration dictionary* into meaningful UCI output.

In order to accomplish this, you must add extra keys to the *configuration dictionary* which have to meet the following requirements:

- the name of the key must be the name of the package that needs to be configured
- the value of the key must be a `list`
- each element in the list must be a `dict`
- each `dict` MUST contain a key named `config_name`
- each `dict` MAY contain a key named `config_value`

This feature is best explained with a few examples.

Dropbear example

The following *configuration dictionary*:

```
{
    "dropbear": [
        {
            "config_name": "dropbear",
            "config_value": "dropbear_1",
            "PasswordAuth": "on",
            "RootPasswordAuth": "on",
            "Port": 22
        }
    ]
}
```

Will be rendered as follows:

```
package dropbear

config dropbear 'dropbear_1'
        option PasswordAuth 'on'
        option Port '22'
        option RootPasswordAuth 'on'
```

OLSRd2 example

The following *configuration dictionary*:

```
{
    "olsrd2": [
        {
            "config_name": "global",
            "config_value": "global",
            "pidfile": "/var/run/olsrd2.pid",
            "lockfile": "/var/lock/olsrd2"
        },
        {
            "config_name": "log",
            "config_value": "log",
            "syslog": "true",
            "stderr": "true",
            "file": "/var/log/olsrd2.log"
        },
        {
            "config_name": "interface",
            "config_value": "olsr2_common",
            "ifname": [
                "loopback",
                "wlan0",
                "wlan1"
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package olsrd2
```

```
config global 'global'
    option lockfile '/var/lock/olsrd2'
    option pidfile '/var/run/olsrd2.pid'

config log 'log'
    option file '/var/log/olsrd2.log'
    option stderr 'true'
    option syslog 'true'

config interface 'olsr2_common'
    list ifname 'loopback'
    list ifname 'wlan0'
    list ifname 'wlan1'
```

CHAPTER 5

OpenWISP 1.x Backend

The OpenWISP 1.x Backend is based on the OpenWRT backend, therefore it inherits all its features with some differences that are explained in this page.

Generate method

The `generate` method of the `OpenWisp` backend differs from the `OpenWrt` backend in a few ways.

1. the generated tar.gz archive is not designed to be installed with `sysupgrade -r`
2. the `generate` method will automatically add a few additional executable scripts:
   - `install.sh` to install the configuration
   - `uninstall.sh` to uninstall the configuration
   - `tc_script.sh` to start/stop traffic control settings
   - one "up" script for each tap VPN configured
   - one "down" script for each tap VPN configured
3. the openvpn certificates are expected to be located the following path: `/openvpn/x509/`
4. the crontabs are expected in to be located at the following path: `/crontabs/`

General settings

The `hostname` attribute in the `general` key is **required**.

Traffic Control

For backward compatibility with OpenWISP Manager the schema of the `OpenWisp` backend allows to define a `tc_options` section that will be used to generate `tc_script.sh`.

The `tc_options` key must be a list, each element of the list must be a dictionary which allows the following keys:

key name	type	function
`name`	string	**required**, name of the network interface that needs to be limited
`input_bandwidth`	integer	maximum input bandwidth in kbps
`output_bandwidth`	integer	maximum output bandwidth in kbps

Traffic control example

The following *configuration dictionary*:

```
{
    "tc_options": [
        {
            "name": "tap0",
            "input_bandwidth": 2048,
            "output_bandwidth": 1024
        }
    ]
}
```

Will generate the following `tc_script.sh`:

```
#!/bin/sh /etc/rc.common

KERNEL_VERSION=`uname -r`
KERNEL_MODULES="sch_htb sch_prio sch_sfq cls_fw sch_dsmark sch_ingress sch_tbf sch_
↪red sch_hfsc act_police cls_tcindex cls_flow cls_route cls_u32"
KERNEL_MPATH=/lib/modules/$KERNEL_VERSION/

TC_COMMAND=/usr/sbin/tc

check_prereq() {
    echo "Checking prerequisites..."

    echo "Checking kernel modules..."
    for kmod in $KERNEL_MODULES; do
    if [ ! -f $KERNEL_MPATH/$kmod.ko ]; then
        echo "Prerequisite error: can't find kernel module '$kmod' in '$KERNEL_MPATH'"
        exit 1
    fi
    done

    echo "Checking tc tool..."
    if [ ! -x $TC_COMMAND ]; then
        echo "Prerequisite error: can't find traffic control tool ($TC_COMMAND)"
        exit 1
    fi

    echo "Prerequisites satisfied."
}
```

```
load_modules() {
    for kmod in $KERNEL_MODULES; do
        insmod $KERNEL_MPATH/$kmod.ko  >/dev/null 2>&1
    done
}

unload_modules() {
    for kmod in $KERNEL_MODULES; do
        rmmod $kmod  >/dev/null 2>&1
    done
}


stop() {

    tc qdisc del dev tap0 root


    tc qdisc del dev tap0 ingress


    unload_modules
}

start() {
    check_prereq
    load_modules


    # shaping output traffic for tap0
    # creating parent qdisc for root
    tc qdisc add dev tap0 root handle 1: htb default 2

    # aggregated traffic shaping parent class

    tc class add dev tap0 parent 1 classid 1:1 htb rate 1024kbit burst 191k


    # default traffic shaping class
    tc class add dev tap0 parent 1:1 classid 1:2 htb rate 512kbit ceil 1024kbit


    # policing input traffic for tap0
    # creating parent qdisc for ingress
    tc qdisc add dev tap0 ingress


    # default policer with lowest preference (last checked)
    tc filter add dev tap0 parent ffff: preference 0 u32 match u32 0x0 0x0 police␣
↪rate 2048kbit burst 383k drop flowid :1

}

boot() {
    start
}

restart() {
```

```
    stop
    start
}
```

Full OpenWISP configuration example

The following example shows a full working *configuration dictionary* for the `OpenWisp` backend.

```
{
    "general": {
        "hostname": "OpenWISP"
    },
    "interfaces": [
        {
            "name": "tap0",
            "type": "virtual"
        },
        {
            "network": "service",
            "name": "br-service",
            "type": "bridge",
            "bridge_members": [
                "tap0"
            ]
        },
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "provinciawifi",
                "isolate": True,
                "network": ["service"]
            }
        }
    ],
    "radios": [
        {
            "name": "radio0",
            "phy": "phy0",
            "driver": "mac80211",
            "protocol": "802.11g",
            "channel": 11,
            "channel_width": 20,
            "tx_power": 10,
            "country": "IT"
        }
    ],
    "openvpn": [
        {
            "name": "2693",
            "cipher": "AES-128-CBC",
            "ca": "/tmp/owispmanager/openvpn/x509/ca_1_service.pem",
            "mute_replay_warnings": True,
            "script_security": 1,
```

```
            "proto": "tcp-client",
            "mute": 10,
            "up_delay": 1,
            "cert": "/tmp/owispmanager/openvpn/x509/l2vpn_client_2693.pem",
            "up": "/tmp/owispmanager/openvpn/vpn_2693_script_up.sh",
            "log": "/tmp/openvpn_2693.log",
            "verb": 1,
            "dev_type": "tap",
            "persist_tun": True,
            "keepalive": "5 40",
            "key": "/tmp/owispmanager/openvpn/x509/l2vpn_client_2693.pem",
            "ns_cert_type": "server",
            "mode": "p2p",
            "pull": True,
            "enabled": True,
            "comp_lzo": "yes",
            "down": "/tmp/owispmanager/openvpn/vpn_2693_script_down.sh",
            "dev": "tap0",
            "nobind": True,
            "remote": [
                {
                    "host": "vpn.openwisp.org",
                    "port": 12128
                }
            ],
            "tls_client": True,
            "resolv_retry": True,
            "up_restart": True
        }
    ],
    "tc_options": [
        {
            "name": "tap0",
            "input_bandwidth": 2048,
            "output_bandwidth": 1024
        }
    ],
    "files": [
        {
            "path": "/openvpn/x509/ca.pem",
            "mode": "0644",
            "contents": "-----BEGIN CERTIFICATE-----\nstripped_down\n-----END␣
↪CERTIFICATE-----\n"
        },
        {
            "path": "/openvpn/x509/l2vpn_client_1_2325_2693.pem",
            "mode": "0644",
            "contents": "-----BEGIN CERTIFICATE-----\nstripped_down\n-----END␣
↪CERTIFICATE-----\n-----BEGIN RSA PRIVATE KEY-----\nstripped_down\n-----END RSA␣
↪PRIVATE KEY-----\n"
        },
        {
            "path": "/crontabs/root",
            "mode": "0644",
            "contents": "* * * * * echo 'test' > /tmp/test-cron"
        }
    ]
}
```

CHAPTER 6

OpenVPN 2.3 Backend

The `OpenVpn` backend allows to generate OpenVPN 2.3.x compatible configurations.

Its schema is limited to a subset of the features available in OpenVPN and it doesn't recognize interfaces, radios, wireless settings and so on.

The main methods work just like the *OpenWRT backend*:

- `__init__`
- `render`
- `generate`
- `write`
- `json`

The main differences are in the resulting configuration and in its schema.

See an example of initialization and rendering below:

```
from netjsonconfig import OpenVpn

config = OpenVpn({
    "openvpn": [
        {
            "ca": "ca.pem",
            "cert": "cert.pem",
            "dev": "tap0",
            "dev_type": "tap",
            "dh": "dh.pem",
            "key": "key.pem",
            "mode": "server",
            "name": "example-vpn",
            "proto": "udp",
            "tls_server": True
        }
    ]
```

```
})
print(config.render())
```

Will return the following output:

```
# openvpn config: test-no-status

ca ca.pem
cert cert.pem
dev tap0
dev-type tap
dh dh.pem
key key.pem
mode server
proto udp
tls-server
```

OpenVPN backend schema

The `OpenVpn` backend schema is limited, it only recognizes an `openvpn` key with a list of dictionaries representing vpn instances. The structure of these dictionaries is described below.

Alternatively you may also want to take a look at the OpenVPN JSON-Schema source code.

According to the NetJSON spec, any unrecognized property will be ignored.

General settings (valid both for client and server)

Required properties:

- name
- mode
- proto
- dev

key name	type	default	allowed values
`name`	string		2 to 24 alphanumeric characters, dashes and underscores
`mode`	string		`p2p` or `server`
`proto`	string		`udp`, `tcp-client`, `tcp-server`
`port`	integer	`1194`	integers
`dev_type`	string		`tun`, `tap`
`dev`	string		any non-whitespace character (max length: 15)
`local`	string		any string
`comp_lzo`	string	`adaptive`	`yes`, `no` or `adaptive`
`auth`	string	`SHA1`	see auth property source code
`cipher`	string	`BF-CBC`	see cipher property source code
`engine`	string		`bsd`, `rsax`, `dynamic` or empty string
`ca`	string		any non whitespace character
`cert`	string		any non whitespace character
`key`	string		any non whitespace character

Continued on nex

Table 6.1 – continued from previous page

key name	type	default	allowed values
`ns_cert_type`	string		`client`, `server` or empty string
`mtu_disc`	string	`no`	`no`, `maybe` or `yes`
`mtu_test`	boolean	`False`	
`fragment`	integer	`0`	any positive integer
`mssfix`	integer	`1450`	any positive integer
`keepalive`	string		two numbers separated by one space
`persist_tun`	boolean	`False`	
`persist_key`	boolean	`False`	
`up`	string		any non whitespace character
`up_delay`	integer	`0`	any positive integer
`down`	string		any non whitespace character
`script_security`	integer	`1`	`0`, `1`, `2`, `3`
`user`	string		any string
`group`	string		any string
`mute`	integer	`0`	any positive integer
`status`	string		string and number separated by space, eg: `/var/log/openvpn.statu`
`status_version`	integer	`1`	`1`, `2`, `3`
`mute_replay_warnings`	boolean	`False`	
`secret`	string		any non whitespace character
`fast_io`	boolean	`False`	
`log`	string		filesystem path
`verb`	integer	`1`	from `0` (disabled) to `11` (very verbose)

Client specific settings

Required properties:

- remote

key name	type	default	allowed values
`remote`	list	`[]`	list of dictionaries containing `host` (str) and `port` (str). Must contain at least one element
`nobind`	boolean	`True`	
`resolv_retry`	boolean	`True`	
`tls_client`	boolean	`True`	
`pull`	boolean	`True`	
`auth_user_pass`	string		any non whitespace character

Server specific settings

key name	type	default	allowed values
`tls_server`	boolean	`True`	
`dh`	string		any non whitespace character
`crl_verify`	string		any non whitespace character
`duplicate_cn`	boolean	`False`	
`client_to_client`	boolean	`False`	
`client_cert_not_required`	boolean	`False`	
`username_as_common_name`	boolean	`False`	
`auth_user_pass_verify`	string		any non whitespace character

Working around schema limitations

The schema does not include all the possible OpenVPN settings, but it can render appropiately any property not included in the schema as long as its type is one the following:

- boolean
- integer
- strings
- lists

For a list of all the OpenVPN configuration settings, refer to the OpenVPN 2.3 manual.

Automatic generation of clients

classmethod OpenVpn.**auto_client**(*host*, *server*, *ca_path=None*, *ca_contents=None*, *cert_path=None*, *cert_contents=None*, *key_path=None*, *key_contents=None*)

Returns a configuration dictionary representing an OpenVPN client configuration that is compatible with the passed server configuration.

Parameters

- **host** – remote VPN server
- **server** – dictionary representing a single OpenVPN server configuration
- **ca_path** – optional string representing path to CA, will consequently add a file in the resulting configuration dictionary
- **ca_contents** – optional string representing contents of CA file
- **cert_path** – optional string representing path to certificate, will consequently add a file in the resulting configuration dictionary
- **cert_contents** – optional string representing contents of cert file
- **key_path** – optional string representing path to key, will consequently add a file in the resulting configuration dictionary
- **key_contents** – optional string representing contents of key file

Returns dictionary representing a single OpenVPN client configuration

Example:

```
from netjsonconfig import OpenVpn

server_config = {
    "ca": "ca.pem",
    "cert": "cert.pem",
    "dev": "tap0",
    "dev_type": "tap",
    "dh": "dh.pem",
    "key": "key.pem",
    "mode": "server",
    "name": "example-vpn",
    "proto": "udp",
    "tls_server": True
}
```

```
dummy_contents = '------ EXAMPLE ------'
client_config = OpenVpn.auto_client('vpn1.test.com',
                                    server=server_config,
                                    ca_path='ca.pem',
                                    ca_contents=dummy_contents,
                                    cert_path='cert.pem',
                                    cert_contents=dummy_contents,
                                    key_path='key.pem',
                                    key_contents=dummy_contents)
client = OpenVpn(client_config)
print(client.render())
```

Will be rendered as:

```
# openvpn config: example-vpn

ca ca.pem
cert cert.pem
dev tap0
dev-type tap
key key.pem
mode p2p
nobind
proto udp
remote vpn1.test.com 1195
resolv-retry
tls-client

# ---------- files ---------- #

# path: ca.pem
# mode: 0644

------ EXAMPLE ------

# path: cert.pem
# mode: 0644

------ EXAMPLE ------

# path: key.pem
# mode: 0644

------ EXAMPLE ------
```

CHAPTER 7

Command line utility

netjsonconfig ships a command line utility that can be used from the interactive shell, bash scripts or other programming languages.

Check out the available options yourself with:

```
$ netjsonconfig --help
usage: netjsonconfig [-h] [--config CONFIG]
                 [--templates [TEMPLATES [TEMPLATES ...]]]
                 [--native NATIVE] --backend {openwrt,openwisp,openvpn}
                 --method {render,generate,write,validate,json}
                 [--args [ARGS [ARGS ...]]] [--verbose] [--version]

Converts a NetJSON DeviceConfiguration object to native router configurations.
Exhaustive documentation is available at: http://netjsonconfig.openwisp.org/

optional arguments:
  -h, --help            show this help message and exit

input:
  --config CONFIG, -c CONFIG
                        config file or string, must be valid NetJSON
                        DeviceConfiguration
  --templates [TEMPLATES [TEMPLATES ...]], -t [TEMPLATES [TEMPLATES ...]]
                        list of template config files or strings separated by
                        space
  --native NATIVE, -n NATIVE
                        path to native configuration file or archive

output:
  --backend {openwrt,openwisp,openvpn}, -b {openwrt,openwisp,openvpn}
                        Configuration backend
  --method {render,generate,write,validate,json}, -m {render,generate,write,validate,
↪json}
                        Backend method to use. "render" returns the
                        configuration in text format; "generate" returns a
```

```
                        tar.gz archive as output; "write" is like generate but
                        writes to disk; "validate" validates the combination
                        of config and templates passed in input;
                        "json" returns NetJSON output:
  --args [ARGS [ARGS ...]], -a [ARGS [ARGS ...]]
                        Optional arguments that can be passed to methods

debug:
  --verbose             verbose output
  --version, -v         show program's version number and exit
```

Here's the common use cases explained:

```
# generate tar.gz from a NetJSON DeviceConfiguration object and save its output to a␣
↪file
netjsonconfig --config config.json --backend openwrt --method generate > config.tar.gz

# convert an OpenWRT tar.gz to NetJSON and print to standard output (with 4 space␣
↪indentation)
netjsonconfig --native config.tar.gz --backend openwrt --method json -a indent="    "

# use write configuration archive to disk in /tmp/routerA.tar.gz
netjsonconfig --config config.json --backend openwrt --method write --args␣
↪name=routerA path=/tmp/

# see output of OpenWrt render method
netjsonconfig --config config.json --backend openwrt --method render

# same as previous but exclude additional files
netjsonconfig --config config.json --backend openwrt --method render --args files=0

# validate the config.json file against the openwrt backend
netjsonconfig --config config.json --backend openwrt --method validate

# abbreviated options
netjsonconfig -c config.json -b openwrt -m render -a files=0

# passing a JSON string instead of a file path
netjsonconfig -c '{"general": { "hostname": "example" }}' -b openwrt -m render
```

Using templates:

```
netjsonconfig -c config.json -t template1.json template2.json -b openwrt -m render

# validate the result of merging config.json, template1.json and template2.json
# against the openwrt backend schema
netjsonconfig -c config.json -t template1.json template2.json -b openwrt -m validate
```

Environment variables

Environment variables are automatically passed to the `context` argument (if you don't know what this argument does please read "*Context (configuration variables)*"), therefore you can reference environment variables inside *configurations* and *templates*:

```
export HOSTNAME=freedom
netjsonconfig -c '{"general": { "hostname": "{{ HOSTNAME }}" }}' -b openwrt -m render
```

You can also avoid using `export` and write everything in a one line command:

```
PORT=2009; netjsonconfig -c config.json -t template1.json -b openwrt -m render
```

CHAPTER 8

Running tests

Running the test suite is really straightforward!

Using runtests.py

Install your forked repo:

```
git clone git://github.com/<your_fork>/netjsonconfig
cd netjsonconfig/
python setup.py develop
```

Install test requirements:

```
pip install -r requirements-test.txt
```

Run tests with:

```
./runtests.py
```

Using nose

Alternatively, you can use the `nose` tool (which has a ton of available options):

```
nosetests
```

See test coverage with:

```
coverage run --source=netjsonconfig runtests.py && coverage report
```

CHAPTER 9

Contributing

Thank you for taking the time to contribute to netjsonconfig.

Follow these guidelines to speed up the process.

Table of Contents:

Reach out before you start

Before opening a new issue, try the following steps:

- look if somebody else has already started working on the same issue by looking in the github issues and pull requests
- look also in the OpenWISP mailing list
- announce your intentions by opening a new issue
- present yourself on the mailing list

Create a virtual environment

Please use a python virtual environment while developing your feature, it keeps everybody on the same page and it helps reproducing bugs and resolving problems.

We suggest you to use virtualenvwrapper for this task (consult install instructions in the virtualenvwrapper docs).

```
mkvirtualenv netjsonconfig  # create virtualenv
```

Fork repo and install your fork

Once you have forked this repository to your own github account or organization, install your own fork in your development environment:

```
git clone git@github.com:<your_fork>/netjsonconfig.git
cd netjsonconfig
workon netjsonconfig  # activate virtualenv
python setup.py develop
```

Ensure test coverage does not decrease

First of all, install the test requirements:

```
workon netjsonconfig  # activate virtualenv
pip install -r requirements-test.txt
```

When you introduce changes, ensure test coverage is not decreased with:

```
nosetests --with-coverage --cover-package=netjsonconfig
```

Follow style conventions (PEP8, isort)

First of all, install the test requirements:

```
workon netjsonconfig  # activate virtualenv
pip install -r requirements-test.txt
```

Before committing your work check that your changes are not breaking the style conventions with:

```
flake8
isort --check-only --recursive .
```

For more information, please see:

- PEP8: Style Guide for Python Code
- isort: a python utility / library to sort imports

Update the documentation

If you introduce new features or change existing documented behavior, please remember to update the documentation!

The documentation is located in the `/docs` directory of the repository.

To do work on the docs, proceed with the following steps:

```
workon netjsonconfig  # activate virtualenv
pip install sphinx
cd docs
make html
```

Send pull request

Now is time to push your changes to github and open a pull request!

CHAPTER 10

Motivations and Goals

In this page we explain the goals of this project and the motivations that led us on this path.

Motivations

Federico Capoano (@nemesisdesign) has written in detail the motivations that brought us here in a blog post: netjson-config: convert NetJSON to OpenWRT UCI.

Goals

The main goal of this library is to replace the configuration generation feature that is shipped in OpenWISP Manager.

We have learned a lot from *OpenWISP Manager*, one of the most important lessons we learned is that the configuration generation feature must be a library decoupled from web framework specific code (eg Rails, Django), this brings many advantages:

- the project can evolve indipendently from the rest of the OpenWISP modules
- easier to use and integrate in other projects
- more people can use it and contribute
- easier maintainance
- easier to document

Another important goal is to build a tool which is **flexible** and **powerful**. We do not want to limit our system to OpenWISP Firmware only, we want to be able to control vanilla OpenWRT devices or other OpenWRT based devices too.

We did this by starting out with the *OpenWrt backend* first, only afterwards we built the *OpenWisp backend* on top of it.

To summarize, our goals are:

- build a reusable library to generate router configurations from NetJSON objects
- support the widely used router specific unix/linux distributions
- provide good and extensive documentation
- keep it simple stupid
- avoid complexity unless extremely necessary
- provide ways to add custom configuration options easily
- provide ways to extend the library
- *encourage contributions*

CHAPTER 11

Change log

Version 0.6.2 [2017-08-29]

- #78 [base] Added support for multiple renderers
- #94 [schema] Made `bssid` not required for wireless stations
- #97 [python2] Fixed `py2-ipaddress` related unicode bug

Version 0.6.1 [2017-07-05]

- 5ddc201: [general] Avoid default mutable arguments
- dde3c9b: [openvpn] Added explicit `list_identifiers` attribute
- 8c26cd6: [docs] Updated outdated OpenWRT rendering examples
- 5f8483e: [openwrt] Fixed repeated bridge gateway case
- #84 [exceptions] Improved validation errors (thanks to @EdoPut)
- #85 [openwrt] Added "vid" option in "switch"
- #86 [openwrt] Added support for "ip6gw" option
- #70 [feature] Backward conversion
- #87 [openwrt] Removed automatic timezone

Version 0.6.0 [2017-06-01]

- #70 [general] Preliminary work for backward conversion, more info in the OpenWISP Mailing List
- #58: [openwrt] Dropped obsolete code in `OpenVpn` converter

- #59: [openwrt] Improved multiple ip address output

Version 0.5.6 [2017-05-24]

- #69: [docs] Improved contributing guidelines (thanks to @EdoPut)
- #71: [bin] Added `validate` to available methods of command line tool (thanks to @EdoPut)
- 845ed83: [version] Improved get_version to follow PEP440
- #73: [netjson] Fixed compatibility with NetJSON specification

Version 0.5.5.post1 [2017-04-18]

- d481781: [docs] Added OpenWRT PPPoE example
- beb435b: [docs] Fixed Basic Concepts summary

Version 0.5.5 [2017-03-15]

- #65: [openwrt] Added missing zonename attribute

Version 0.5.4.post1 [2017-03-07]

- 4aaecae: [docs] Added documentation regarding template overrides

Version 0.5.4 [2017-02-14]

- 6f712d1: [utils] Implemented identifiers as parameters in `utils.merge_list`
- fcae96c: [openwrt] Added `config_value` identifier in `utils.merge_list`
- eaa04de: [docs] Improved "All the other settings" section in `OpenWrt` backend
- #60 [openvpn] Fixed `resolv_retry` bug; **minor backward incompatible change**: handled in django-netjsonconfig with a migration
- f25e77e: [openvpn] Added `topology` attribute to schema
- c4aa07a: [openvpn] Allow to omit seconds in status attribute

Version 0.5.3 [2017-01-17]

- #56: [general] Implemented smarter merge mechanism
- #57: [openwrt] Fixed interface `enabled` bug
- 7a152a3: [openwrt] Renamed `enabled` to `disabled` in OpenVPN section (for consistency)

Version 0.5.2 [2016-12-29]

- #55: [vars] Fixed broken evaluation of multiple variables

Version 0.5.1 [2016-09-22]

- b486c4d: [openvpn] corrected wrong `client` mode, renamed to `p2p`
- c7e51c6: [openvpn] added `pull` option for clients
- dde3128: [openvpn] differentiate server between manual, routed and bridged

Version 0.5.0 [2016-09-19]

- added `OpenVpn` backend
- afbc3a3: [openwisp] fixed openvpn integration (partially backward incompatible)
- 1234c34: [context] improved flexibility of configuration variables
- #54: [openwrt] fixed netmask issue on ipv4

Version 0.4.5 [2016-09-05]

- #53: [docs] avoid ambiguity on dashes in context
- #52: [schema] added countries list as `enum` for radios (thanks to @zachantre)

Version 0.4.4 [2016-06-27]

- #50: [openwrt] add logical name to all generated configuration items

Version 0.4.3 [2016-04-23]

- c588e5d: [openwrt] avoid adding `dns` and `dns_search` if `proto` is `none`

Version 0.4.2 [2016-04-11]

- 92f9a43: [schema] added human readable values for mode `access_point` and `802.11s`
- #47: [openwrt] improved encryption support
- 1a4c493: [openwrt] `igmp_snooping` now correctlt defaults to `True`
- #49: [schema] added descriptions and titles

Version 0.4.1 [2016-04-04]

- b903c6f: [schema] corrected wrong ipv4 minLength and maxLength
- de98ae6: [schema] fixed interface minLength attribute
- 4679282: [schema] added regexp pattern for interface mac address (can be empty)
- 067b471: [schema] switched order between MTU and MAC address properties
- 26b62dd: [schema] added pattern for wireless BSSID attribute
- 11da509: [openwrt] added regexp pattern to `maclist` elements
- b061ee4: [openwrt] fixed empty output bug if addresses is empty list
- 7f74209: [openwrt] removed support for `chanbw` for types `ath5k` and `ath9k` (**backward incompatible change**)
- #46: [schema] introduced different profiles for radio settings
- 6ab9d5b [openwrt] added support for "Automatic Channel Selection"
- #48: [openwrt] improved support for config lists
- 9f93776: [openwrt] simplified definition of custom interface "proto" options
- a5f63f0: [openwrt] allow to override general dns and dns_search settings
- 1b58f97: [schema] added `stp` (spanning tree protocol) property on bridge interfaces
- bfbf23d: [openwrt] added `igmp_snooping` property on bridge interfaces
- 269c7bf: [openwrt] added `isolate` property on wireless access points
- 2cbc242: [openwrt] fixed `autostart` when `False`
- 85bd7dc: [openwrt] fixed mac address override on interfaces
- 45159e8: [openwrt] allow overriding `htmode` option
- b218f7d: [schema] added `enum_titles` in `encryption` protocols
- ef8c296: [schema] validate general hostname format
- 2f23cfd: [schema] validate interface ipv4 address format
- 612959e: [openwrt] validate ntp server hostname format
- f1116f0: [schema] validate `dns_search` hostname format #42
- 372d634 [openwrt] do not set dns to dhcp interfaces

Version 0.4.0 [2016-03-22]

- #40: [openwrt] added support for ULA prefix
- #44: [schema] added `none` to encryption choices
- #45: [schema] improved address definition
- #43: [openwrt] improved static routes
- #41: [schema] added `wds` property & removed `wds` mode
- #36: [schema] added specific settings for 802.11s (mesh) mode

- 3f6d2c6: [schema] removed NetJSON `type` from schema
- 04c6058: [openwrt] made file `mode` property required (**backward incompatible change**)
- 00e784e: [openwrt] added default switch settings
- dd708cb: [openwrt] added NTP default settings
- f4148e4: [schema] removed `txqueuelen` from interface definition
- 574a48d: [schema] added `title` and `type` to `bridge_members`
- c6276f2: [schema] MTU title and minimum value
- d8ab0e0: [schema] added `minLength` to interface name
- 67a0916: [schema] added `minLength` to radio name
- 258892e: [schema] added possible `ciphers`
- 2751fe3: [schema] improved definition of wireless interface fields
- 478ef16: [openwrt] added `wmm` property for wireless access points
- b9a14f3: [schema] added `minLength` and `maxLength` to interface `mac` property
- 526c2d1: [schema] added `minLength` and maxLength to wireless `bssid` property
- c8c95d6: [schema] improved ordering and titles of wireless properties
- a226e90: [openwrt] ignore advanced wifi options if zero
- e008ef6: [openwrt] added `macfilter` to wireless access points
- c70ab76: [openwrt] fixed empty dns and dns-search bug
- 778615a: [openwrt] increased network `maxLength`

Version 0.3.7 [2016-02-19]

- 007da6e: renamed "Coordinated Universal Time" to "UTC"
- 2c1e72e: fixed 'tx_power' `KeyError`, introduced in 71b083e
- aa8b485: added `utils.evaluate_vars` function
- 7323491: simplified implementation of *configuration variables*

Version 0.3.6 [2016-02-17]

- fixed `flake8` and `isort` warnings
- added `flake8` and `isort` checks to travis build
- 6ec5ce8: minor regexp optimization for generate method
- #39: added configuration variables feature
- a3486d2: the shell utility can now use environment variables in `config` and `templates`, read relevant docs

Version 0.3.5 [2016-02-10]

- 18ecf28: removed `hardware` and `operating_system` sections
- 75c259d: reordered schema sections
- 010ca98: file contents can now be only strings (**backward incompatible change**)
- e2bb3b2: added non-standard `propertyOrder` attributes to schemas to facilitate UI ordering
- #37: [schema] radio `tx_power` not required anymore
- #38: [openwrt schema] hardened file mode contraints
- c2cc3fc: [schema] added minlength and maxlength to hostname

Version 0.3.4 [2016-01-14]

- #35 wifi inherits `disabled` from interface

Version 0.3.3 [2015-12-18]

- 219f638 [cli] fixed binary standard output for `generate` method
- a0b1373 removed timestamp from generated configuration archive to ensure reliable checksums

Version 0.3.2 [2015-12-11]

- #31 added files in `render` output
- #32 `generate` now returns an in-memory file object
- badf292 updated command line utility script and examples
- #33 added `write` method
- 5ff7360 [cli] positional `config` param is now `--config` or `-c`
- 28de4a5 [cli] marked required arguments: `--config`, `--backend` and `--method`
- f55cc4a [cli] added `--arg` option to pass arguments to methods

Version 0.3.1 [2015-12-02]

- 69197ed added "details" attribute to `ValidationError`
- 0005186 avoid modifying original `config` argument

Version 0.3 [2015-11-30]

- #18 added `OpenWisp` backend
- 66ee96 added file permission feature
- #19 added sphinx documentation (published at netjsonconfig.openwisp.org)
- 30348e hardened ntp server option schema for `OpenWrt` backend
- c31375 added madwifi to the allowed drivers in schema `OpenWrt` backend
- #30 updated schema according to latest NetJSON spec

Version 0.2 [2015-11-23]

- #20 added support for array of lines in files
- #21 date is now correctly set in tar.gz files
- 82cc5e configuration archive is now compatible with `sysupgrade -r`
- #22 improved and simplified bridging
- #23 do not ignore interfaces with no addresses
- #24 restricted schema for interface names
- #25 added support for logical interface names
- #26 `merge_dict` now returns a copy of all the elements
- d22d59 restricted SSID to 32 characters
- #27 improved wireless definition
- #28 removed "enabled" in favour of "disabled"

Version 0.1 [2015-10-20]

- Added `OpenWrt` Backend
- Added command line utility `netjsonconfig`
- Added multiple templating feature
- Added file inclusion feature

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

G

J

M

P

R

W