netjsonconfig documentation

Release 0.4

Federico Capoano

March 31, 2016

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 or OpenWisp Firmware.

Its main features are:

- OpenWRT support
- OpenWISP Firmware support
- Possibility to support more firmwares via custom backends
- Based on the NetJSON RFC
- **Validation** based on JSON-Schema
- **Templates**: store common configurations in template files
- **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

1.1 Install stable version from pypi

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

```
pip install netjsonconfig
```

1.2 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
```

1.3 Install git fork for contributing

If you want to contribute, we suggest to install your cloned fork:

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

CHAPTER 2

Basic concepts

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

- **configuration dictionary**: python dictionary representing the configuration of a router
- **backend**: python class used to process the configuration and generate the final router configuration
- **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
- **context**: variables that can be referenced from the *configuration dictionary*

2.1 Configuration format: NetJSON

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:

```
{
    "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.

2.2 Backends

A backend is a python class used to process the *configuration dictionary* and generate the final router configuration, 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)

Example initialization of `OpenWrt` backend:

```
from netjsonconfig import OpenWrt

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

```
                    "family": "ipv6"
                }
            ]
        }
    ]
})
```

2.3 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.

2.4 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.

2.5 Templates

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.

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])
```

The function used under the hood to merge dictionaries and lists is `netjsonconfig.utils.merge_config`:

`netjsonconfig.utils.`**`merge_config`**(*template*, *config*)

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 summed in order to create a list which contains elements of both
- values of type `dict` will be merged recursively

Parameters

- **`template`** – template `dict`
- **`config`** – config `dict`

Returns merged `dict`

2.6 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.

2.7 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'

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 }}`, including spaces around `var_name`;
- variable names can contain only alphanumeric characters, dashes and underscores;
- unrecognized variables will be ignored;

2.8 Project goals

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

2.9 Support

Send questions to the OpenWISP Mailing List.

2.10 License

CHAPTER 3

OpenWRT Backend

The `OpenWrt` backend is the base backend of the library.

3.1 Initialization

`OpenWrt.__init__`(*config*, *templates=[]*, *context={}*)

Parameters

- **`config`** – `dict` containing valid **NetJSON DeviceConfiguration**
- **`templates`** – `list` containing **NetJSON** dictionaries that will be used as a base for the main config, defaults to empty list
- **`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 OpenWrt

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

3.2 Render method

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

Converts the configuration dictionary into the native OpenWRT UCI 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": "192.168.1.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                },
                {
                    "address": "192.168.2.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                },
                {
                    "address": "fd87::1",
                    "mask": 128,
                    "proto": "static",
                    "family": "ipv6"
                }
            ]
        }
    ]
})
print(o.render())
```

Will return the following output:

```
package network

config interface 'eth0_1'
        option ifname 'eth0.1'
        option proto 'static'
        option ipaddr '192.168.1.1/24'

config interface 'eth0_1_2'
        option ifname 'eth0.1'
        option proto 'static'
        option ipaddr '192.168.2.1/24'

config interface 'eth0_1_3'
        option ifname 'eth0.1'
        option proto 'static'
        option ip6addr 'fd87::1/128'
```

3.3 Generate method

OpenWrt.**generate**()

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

The archive can be installed in OpenWRT with the following command:

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

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.

3.4 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`.

3.5 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"
    }
}
```

3.6 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	path of the file in the tar.gz archive
`contents`	string	yes	plain text contents of the file, new lines must be encoded as *n*
`mode`	string	no	permissions, if omitted will default 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)`

3.6.1 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()
```

3.6.2 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()
```

3.7 Including arbitrary options

It is very easy to add arbitrary UCI options in the resulting configuration **as long as the configuration dictionary does not violate the schema**.

Note: This feature is a deliberate design choice aimed at providing maximum flexibility. We want to avoid unnecessary limitations.

In the following example we will add two arbitrary options: `custom` and `fancy`.

```
from netjsonconfig import OpenWrt

o = OpenWrt({
    "interfaces": [
        {
            "name": "eth0",
            "type": "ethernet",
            "custom": "custom_value",
            "fancy": True
        }
    ]
})
print(o.render())
```

Will return the following output:

```
package network

config interface 'eth0'
        option ifname 'eth0'
        option custom 'custom_value'
        option fancy '1'
        option proto 'none'
```

Note: The hypotetical `custom` and `fancy` options would not be recognized by OpenWRT and they would be therefore ignored by the UCI parser.

We are using them here just to demonstrate how to add complex configuration options that are not defined in the NetJSON spec or in the schema of the `OpenWrt` backend.

3.8 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.

3.8.1 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)

3.8.2 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
        option hostname 'routerA'
        option timezone 'UTC'

package network

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

3.9 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).

3.9.1 Interface object extensions

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

- each interface item can specify a `network` option which allows to manually set the logical interface name
- the `proto` key of each item in the `addresses` list allows all the UCI proto options officially supported by OpenWRT, eg: dhcpv6, ppp, 3g, gre and others
- the `wireless` dictionary (valid only for wireless interfaces) can also specify a `network` key which allows to list on or more networks to which the wireless interface will be attached to (see the *relevant example*)

3.9.2 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 '127.0.0.1/8'
        option proto 'static'
```

3.9.3 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'
```

3.9.4 Bridge interface

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",
            "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 ipaddr '172.17.0.2/24'
        option proto 'static'
        option type 'bridge'
```

3.9.5 Wireless interface

The following *configuration dictionary*:

```
{
    "interfaces": [
        {
            "name": "wlan0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "access_point",
                "ssid": "wpa2-personal",
                "encryption": {
```

```
                    "enabled": True,
                    "protocol": "wpa2_personal",
                    "ciphers": [
                        "tkip",
                        "ccmp"
                    ],
                    "key": "passphrase012345"
                }
            }
        }
    ]
}
```

Will be rendered as follows:

```
package network

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

package wireless

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

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.

3.9.6 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 '192.168.0.2/24'
        option proto 'static'
        option type 'bridge'

package wireless

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

3.9.7 Wireless access point with macfilter ACL

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"
                ]
            }
        }
    ]
}
```

Will be rendered as:

```
package network

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

package wireless

config wifi-iface
        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'
```

3.9.8 Wireless mesh (802.11s) example

Setting up 802.11s interfaces is fairly simple, in the following example we bridge the `lan` interface with `mesh0`, the latter being an 802.11s wireless interface.

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

```
{
    "interfaces": [
        {
            "name": "mesh0",
            "type": "wireless",
            "wireless": {
                "radio": "radio0",
                "mode": "802.11s",
                "mesh_id": "ninux",
                "network": ["lan"]
            }
        },
        {
```

```
            "name": "lan",
            "type": "bridge",
            "bridge_members": ["mesh0"],
            "addresses": [
                {
                    "address": "192.168.0.1",
                    "mask": 24,
                    "proto": "static",
                    "family": "ipv4"
                }
            ]
        }
    ]
}
```

Will be rendered as follows:

```
package network

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

config interface 'lan'
        option ifname 'mesh0'
        option ipaddr '192.168.0.1/24'
        option proto 'static'
        option type 'bridge'

package wireless

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

3.9.9 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
        option bssid '02:b8:c0:00:00:00'
        option device 'radio0'
        option ifname 'wlan0'
        option mode 'adhoc'
        option network 'wlan0'
        option ssid 'freifunk'
```

3.9.10 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
        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
        option device 'radio1'
        option ifname 'wlan1'
        option mode 'ap'
        option network 'wds_bridge'
        option ssid 'FreeRomaWifi'
```

3.10 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).

3.10.1 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

3.10.2 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'
```

3.11 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).

3.11.1 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		MTU for route, only numbers are allowed
`table`	string	`False`	Routing table id, only numbers are allowed
`onlink`	boolean		When enabled, gateway is on link even if the gateway does not match any interface prefix

3.11.2 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
        option gateway 'fd88::1'
        option interface 'eth1'
        option metric '0'
        option target 'fd89::1/128'
```

3.12 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.

3.12.1 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
        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
        option dest '192.168.3.0/24'
        option goto '0'
        option src '192.168.1.0/24'

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

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

3.13 Switch settings

The 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.

3.13.1 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
        option enable_vlan '1'
        option name 'switch0'
        option reset '1'

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

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

3.14 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

3.14.1 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'
```

3.15 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.

3.15.1 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'
```

3.16 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!**

Netjsonconfig aims to be very flexible, that's why the `OpenWrt` backend ships a `DefaultRenderer`, which will try to parse any unrecognized key of the *configuration dictionary* and render meaningful UCI output.

To supply configuration options to the `DefaultRenderer` a few prerequisites must be met:

- the name of the key must be the name of the package that needs to be configured
- the value of the key must be of type `list`
- each element in the list must be of type `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.

3.16.1 Dropbear example

The following *configuration dictionary*:

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

Will be rendered as follows:

```
package dropbear

config dropbear
        option PasswordAuth 'on'
```

```
        option Port '22'
        option RootPasswordAuth 'on'
```

3.16.2 OpenVPN example

The following *configuration dictionary*:

```
{
    "openvpn": [
        {
            "config_name": "openvpn",
            "config_value": "client_tun_0",
            "enabled": True,
            "client": True,
            "dev": "tun",
            "proto": "tcp",
            "resolv_retry": "infinite",
            "nobind": True,
            "persist_tun": True,
            "persist_key": True,
            "ca": "/etc/openvpn/ca.crt",
            "cert": "/etc/openvpn/client.crt",
            "key": "/etc/openvpn/client.crt",
            "cipher": "BF-CBC",
            "comp_lzo": "yes",
            "remote": "vpn.myserver.com 1194",
            "enable": True,
            "tls_auth": "/etc/openvpn/ta.key 1",
            "verb": 5,
            "log": "/tmp/openvpn.log"
        }
    ]
}
```

Will be rendered as follows:

```
package openvpn

config openvpn 'client_tun_0'
        option ca '/etc/openvpn/ca.crt'
        option cert '/etc/openvpn/client.crt'
        option cipher 'BF-CBC'
        option client '1'
        option comp_lzo 'yes'
        option dev 'tun'
        option enable '1'
        option enabled '1'
        option key '/etc/openvpn/client.crt'
        option log '/tmp/openvpn.log'
        option nobind '1'
        option persist_key '1'
        option persist_tun '1'
        option proto 'tcp'
        option remote 'owm.provinciawifi.it 1194'
        option resolv_retry 'infinite'
        option tls_auth '/etc/openvpn/ta.key 1'
        option verb '5'
```

CHAPTER 4

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.

4.1 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/`

4.2 General settings

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

4.3 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

4.3.1 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_p
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

}

boot() {
    start
}

restart() {
    stop
    start
}
```

4.3.2 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": [
        {
            "config_name": "openvpn",
            "config_value": "2693",
            "enabled": "1",
            "client": "1",
            "dev": "tap0",
            "dev_type": "tap",
            "proto": "tcp-client",
            "remote": "vpn.wifiservice.com 12128",
            "nobind": "1",
            "keepalive": "5 40",
            "ns_cert_type": "server",
            "resolv_retry": "infinite",
            "comp_lzo": "yes",
            "tls_client": "1",
            "ca": "/tmp/owispmanager/openvpn/x509/ca.pem",
            "key": "/tmp/owispmanager/openvpn/x509/l2vpn_client_1_2325_2693.pem",
            "cert": "/tmp/owispmanager/openvpn/x509/l2vpn_client_1_2325_2693.pem",
            "up": "/tmp/owispmanager/openvpn/vpn_l2vpn_client_1_2325_2693_script_up.sh",
            "down": "/tmp/owispmanager/openvpn/vpn_l2vpn_client_1_2325_2693_script_down.sh",
            "cipher": "AES-128-CBC",
            "script_security": "3",
            "up_delay": "1",
```

```
            "up_restart": "1",
            "persist_tun": "1",
            "mute_replay_warnings": "1",
            "verb": "1",
            "mute": "10"
        }
    ],
    "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-----
        },
        {
            "path": "/crontabs/root",
            "mode": "0644",
            "contents": "* * * * * echo 'test' > /tmp/test-cron"
        }
    ]
}
```

CHAPTER 5

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 ...]]] --backend
                     {openwrt,openwisp} --method {render,generate,write}
                     [--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

output:
  --backend {openwrt,openwisp}, -b {openwrt,openwisp}
                        Configuration backend: openwrt or openwisp
  --method {render,generate,write}, -m {render,generate,write}
                        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;
  --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

# 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

# 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
```

5.1 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 6

Running tests

Running the test suite is really straightforward!

6.1 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
```

6.2 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 7

Contributing

We welcome contributions and feedback!

If you intend to contribute in any way please keep the following guidelines in mind:

1. Announce your intentions in the OpenWISP Mailing List
2. *Install git fork for contributing*
3. Follow PEP8, Style Guide for Python Code
4. Write code
5. Write tests for your code
6. Ensure all tests pass
7. Ensure test coverage does not decrease
8. Document your changes
9. Send pull request

CHAPTER 8

Motivations and Goals

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

8.1 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.

8.2 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 9

Change log

The complete change log is available on the github repo.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Symbols

G

J

M

R

W