

---

# **netjsonconfig documentation**

***Release 0.4.2***

**Federico Capoano**

April 11, 2016



|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Setup</b>                                | <b>3</b>  |
| 1.1      | Install stable version from pypi . . . . .  | 3         |
| 1.2      | Install development version . . . . .       | 3         |
| 1.3      | Install git fork for contributing . . . . . | 3         |
| <b>2</b> | <b>Basic concepts</b>                       | <b>5</b>  |
| 2.1      | Configuration dictionary . . . . .          | 5         |
| 2.2      | Backend . . . . .                           | 6         |
| 2.3      | Schema . . . . .                            | 7         |
| 2.4      | Validation . . . . .                        | 7         |
| 2.5      | Template . . . . .                          | 7         |
| 2.6      | Multiple template inheritance . . . . .     | 9         |
| 2.7      | Context (configuration variables) . . . . . | 9         |
| 2.8      | Project goals . . . . .                     | 10        |
| 2.9      | Support . . . . .                           | 11        |
| 2.10     | License . . . . .                           | 11        |
| <b>3</b> | <b>OpenWRT Backend</b>                      | <b>13</b> |
| 3.1      | Initialization . . . . .                    | 13        |
| 3.2      | Render method . . . . .                     | 13        |
| 3.3      | Generate method . . . . .                   | 14        |
| 3.4      | Write method . . . . .                      | 15        |
| 3.5      | JSON method . . . . .                       | 16        |
| 3.6      | General settings . . . . .                  | 17        |
| 3.7      | Network interfaces . . . . .                | 17        |
| 3.8      | Bridge settings . . . . .                   | 20        |
| 3.9      | Wireless settings . . . . .                 | 22        |
| 3.10     | Radio settings . . . . .                    | 31        |
| 3.11     | Static Routes . . . . .                     | 34        |
| 3.12     | Policy routing . . . . .                    | 35        |
| 3.13     | Programmable switch settings . . . . .      | 37        |
| 3.14     | NTP settings . . . . .                      | 38        |
| 3.15     | LED settings . . . . .                      | 39        |
| 3.16     | Including custom options . . . . .          | 40        |
| 3.17     | Including custom lists . . . . .            | 41        |
| 3.18     | Including additional files . . . . .        | 43        |
| 3.19     | All the other settings . . . . .            | 44        |

|           |                                 |           |
|-----------|---------------------------------|-----------|
| <b>4</b>  | <b>OpenWISP 1.x Backend</b>     | <b>47</b> |
| 4.1       | Generate method . . . . .       | 47        |
| 4.2       | General settings . . . . .      | 47        |
| 4.3       | Traffic Control . . . . .       | 47        |
| <b>5</b>  | <b>Command line utility</b>     | <b>53</b> |
| 5.1       | Environment variables . . . . . | 54        |
| <b>6</b>  | <b>Running tests</b>            | <b>55</b> |
| 6.1       | Using runtests.py . . . . .     | 55        |
| 6.2       | Using nose . . . . .            | 55        |
| <b>7</b>  | <b>Contributing</b>             | <b>57</b> |
| <b>8</b>  | <b>Motivations and Goals</b>    | <b>59</b> |
| 8.1       | Motivations . . . . .           | 59        |
| 8.2       | Goals . . . . .                 | 59        |
| <b>9</b>  | <b>Change log</b>               | <b>61</b> |
| <b>10</b> | <b>Indices and tables</b>       | <b>63</b> |

Netjsonconfig is part of the [OpenWISP project](#). **netjsonconfig** is a python library that converts [NetJSON Device-Configuration](#) 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 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:



## 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 drive):

```
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
```





---

## 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
- *Multiple template inheritance*: possibility inherit common configuration options from more than one template
- *Context (configuration variables)*: variables that can be referenced from the *configuration dictionary*

### 2.1 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:

```
{
  "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 Backend

A backend is a python class used to process the *configuration dictionary* and generate the final router configuration, each supported firmware or operating 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({  
  "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 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.

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': 'xk7OzAlqN6h1Ggxy8UH5NI8kQnbuLxsE'
}

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 'xk7OzAlqN6h1Ggxy8UH5NI8kQnbuLxsE'
    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

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





---

## 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"
    }
})
```

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

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

### 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 configura-

tion.

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 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.

### 3.6.1 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 <a href="#">allowed timezone values</a> (first element of each tuple) |

### 3.6.2 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'
```

## 3.7 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`

### 3.7.1 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.

### 3.7.2 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'
```

### 3.7.3 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 ipaddr '10.27.251.1/24'
    option proto 'static'

config interface 'eth0_2'
    option ifname 'eth0'
    option ip6addr 'fdb4:5f35:e8fd::1/48'
    option proto 'static'

```

### 3.7.4 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, unless an interface has DHCP enabled, in which case the UCI output won't contain the `dns` option, 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"
                }
            ]
        }
    ]
}

```

```
    ]
  }
]
}
```

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 '192.168.1.1/24'
    option proto 'static'

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

### 3.7.5 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 'dchp6'
```

## 3.8 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 |

### 3.8.1 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 igmp 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 '172.17.0.2/24'
```

```
option proto 'static'
option type 'bridge'
option stp '1'
```

### 3.9 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.

### 3.9.1 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
    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.

### 3.9.2 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'

```

### 3.9.3 Wireless access point with macfilter ACL

The OpenWrt renderer 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
    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'

```

### 3.9.4 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 '192.168.0.1/24'
    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'
```

### 3.9.5 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'
```

### 3.9.6 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'

```

### 3.9.7 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
    option device 'radio0'
    option encryption 'psk2+tkip+ccmp'
    option ifname 'wlan0'

```

```
option key 'passphrase012345'
option mode 'ap'
option network 'wlan0'
option ssid 'wpa2-personal'
```

### 3.9.8 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,
        }
      }
    }
  ]
}
```

UCI Output:

```
package network

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

package wireless

config wifi-iface
    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 network 'wlan0'
    option port '1812'
    option server '192.168.0.1'
    option ssid 'eduroam'
```

### 3.9.9 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
    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'
```

## 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'
```

### 3.10.3 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'
```

### 3.10.4 802.11ac 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'
```

## 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  |
|---------------------|---------|---------|--|
| <code>type</code>   | string  | unicast | unicast, local, broadcast, multicast, unreachable prohibit, blackhole, anycast           |
| <code>mtu</code>    | string  | None    | MTU for route, only numbers are allowed  |
| <code>table</code>  | string  | None    | Routing table id, only numbers are allowed   |
| <code>onlink</code> | boolean | False   | 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 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](#).

#### 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            |
|----------------------------|---------|---------------------|
| <code>enabled</code>       | boolean | ntp client enabled  |
| <code>enable_server</code> | boolean | ntp server enabled  |
| <code>server</code>        | 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 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'
```

## 3.17 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

### 3.17.1 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'
```

### 3.17.2 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'
```

### 3.17.3 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'
```

### 3.17.4 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
    list basic_rate '6000'
    list basic_rate '9000'
    option device 'radio0'
    option ifname 'wlan0'
    option mode 'ap'
    option network 'wlan0'
    option ssid 'open'
```

## 3.18 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 <code>\n</code> |
| mode     | string | yes      | filesystem permissions, defaults to <code>0644</code>                         |

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)`

### 3.18.1 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()
```

### 3.18.2 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()
```

## 3.19 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.

### 3.19.1 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'
```

### 3.19.2 OpenVPN example

The following *configuration dictionary*:

```
{
  "openvpn": [
    {
      "config_name": "openvpn",
      "config_value": "client_tun_0",
      "enabled": True,
      "client": True,
      "dev": "tun",
      "proto": "tcp",
      "resolve_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 'own.provinciawifi.it 1194'
    option resolv_retry 'infinite'
    option tls_auth '/etc/openvpn/ta.key 1'
    option verb '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.

### 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   |
|-------------------------------|---------|--|
| <code>name</code>             | string  | <b>required</b> , name of the network interface that needs to be limited |
| <code>input_bandwidth</code>  | integer | maximum input bandwidth in kbps  |
| <code>output_bandwidth</code> | 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_pie"
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",
        "resolve_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"
    }
]
}

```





---

## 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
```

---

## 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
```



---

# 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



---

## 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 independently 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](#)



---

## Change log

---

The complete [change log](#) is available on the github repo.



---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



## Symbols

`__init__()` (netjsonconfig.OpenWrt method), [13](#)

## G

`generate()` (netjsonconfig.OpenWrt method), [14](#)

## J

`json()` (netjsonconfig.OpenWrt method), [16](#)

## M

`merge_config()` (in module netjsonconfig.utils), [9](#)

## R

`render()` (netjsonconfig.OpenWrt method), [13](#)

## W

`write()` (netjsonconfig.OpenWrt method), [15](#)