Reconfigure

Release 1.0a1

Aug 02, 2017

Contents

CHAPTER 1

Links:

- Source at GitHub
- Questions? Email me
- PyPI

Browse API on SourceGraph

CHAPTER 2

Contents:

Quickstart

Adding lines to `fstab`:

```
>>> from reconfigure.configs import FSTabConfig
>>> from reconfigure.items.fstab import FilesystemData
>>>
>>> config = FSTabConfig(path='/etc/fstab')
>>> config.load()
>>> print config.tree
{
    "filesystems": [
        {
            "passno": "0",
            "device": "proc",
            "mountpoint": "/proc",
            "freq": "0",
            "type": "proc",
            "options": "nodev,noexec,nosuid"
        },
        {
            "passno": "1",
            "device": "UUID=dfccef1e-d46c-45b8-969d-51391898c55e",
            "mountpoint": "/",
            "freq": "0",
            "type": "ext4",
            "options": "errors=remount-ro"
        }
    ]
}
>>> tmpfs = FilesystemData()
>>> tmpfs.mountpoint = '/srv/cache'
>>> tmpfs.type = 'tmpfs'
>>> tmpfs.device = 'none'
```

```
>>> config.tree.filesystems.append(tmpfs)
>>> config.save()
>>> quit()
$ cat /etc/fstab
proc    /proc   proc    nodev,noexec,nosuid     0       0
UUID=dfccef1e-d46c-45b8-969d-51391898c55e / ext4 errors=remount-ro 0 1
none    /srv/cache      tmpfs   none    0       0
```

Changing Samba settings:

```
>>> from reconfigure.configs import SambaConfig
>>> config = SambaConfig(path='/etc/samba/smb.conf')
>>> config.load()
>>> print config.tree.shares
[
    {
        "comment": "All Printers",
        "browseable": false,
        "create_mask": "0700",
        "name": "printers",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": false,
        "path": "/var/spool/samba"
    },
    {
        "comment": "Printer Drivers",
        "browseable": true,
        "create_mask": "0744",
        "name": "print$",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": false,
        "path": "/var/lib/samba/printers"
    }
]
>>> config.tree.shares[0].guest_ok = True
>>> print config.tree.shares
[
    {
        "comment": "All Printers",
        "browseable": false,
        "create_mask": "0700",
        "name": "printers",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": true,
        "path": "/var/spool/samba"
    },
    {
        "comment": "Printer Drivers",
        "browseable": true,
        "create_mask": "0744",
        "name": "print$",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": false,
        "path": "/var/lib/samba/printers"
```

```
    }
]
>>> config.save()
```

Architecture

Trees

Reconfigure operates with three types of data:

- Raw config text
- Syntax tree
- Data tree

Config text

This is a raw content, as read from the config file. It is fed to *Parsers* to produce the *Syntax trees*.

Syntax trees

Syntax tree is an object tree built from *reconfigure.nodes.Node* objects, representing the syntax structure of the file. This is very similar to Abstract Syntax Trees.

Syntax trees are produced by *Parsers* classes.

Example:

```
>>> text = open('/etc/samba/smb.conf').read()
>>> text
'#\n# Sample configuration file for the Samba suite for Debian GNU/Linux.\
...
>>> from reconfigure.parsers import IniFileParser
>>> parser = IniFileParser()
>>> node_tree = parser.parse(text)
>>> print node_tree
(None)
        (global)
                workgroup = WORKGROUP
                server string = %h server (Samba, Ubuntu)
                dns proxy = no
                log file = /var/log/samba/log.%m
                max log size = 1000
                syslog = 0
                panic action = /usr/share/samba/panic-action %d
                encrypt passwords = true
                passdb backend = tdbsam
                obey pam restrictions = yes
                unix password sync = yes
                passwd program = /usr/bin/passwd %u
                passwd chat = *Enter\snew\s*\spassword:* %n\n␣
↪*Retype\snew\s*\spassword:* %n\n *password\supdated\ssuccessfully* .
                pam password change = yes
```

```
                map to guest = bad user
                usershare allow guests = yes
        (printers)
                comment = All Printers
                browseable = no
                path = /var/spool/samba
                printable = yes
                guest ok = no
                read only = yes
                create mask = 0700
>>> node_tree
<reconfigure.nodes.RootNode object at 0x219a150>
>>> node_tree.children[0]
<reconfigure.nodes.Node object at 0x219a950>
>>> node_tree.children[0].name
'global'
>>> node_tree.children[0].children[0]
<reconfigure.nodes.PropertyNode object at 0x219aa10>
>>> node_tree.children[0].children[0].name
'workgroup'
>>> node_tree.children[0].children[0].value
'WORKGROUP'
```

reconfigure.nodes.Node reference page contains more information on how to manipulate node trees.

Parsers work both ways - you can call `stringify()` and get the text representation back. Even more, you can feed the node tree to *another* parser and get the config in other format:

```
>>> from reconfigure.parsers import JsonParser
>>> json_parser = JsonParser()
>>> json_parser.stringify(node_tree)
>>> print json_parser.stringify(node_tree)
{
    "global": {
        "encrypt passwords": "true",
        "pam password change": "yes",
        "passdb backend": "tdbsam",
        "passwd program": "/usr/bin/passwd %u",
        ...
    },
    "print$": {
        "comment": "Printer Drivers",
        "path": "/var/lib/samba/printers",
        "read only": "yes",
        ...
```

Syntax trees might look useful to you, but they are not nearly as cool as *Data trees*

Data trees

Data tree represents the actual, meaningful ideas stored in the config. Straight to example:

```
>>> from reconfigure.builders import BoundBuilder
>>> from reconfigure.items.samba import SambaData
>>> builder = BoundBuilder(SambaData)
>>> data_tree = builder.build(node_tree)
>>> data_tree
```

```
{
    "global": {
        "server_string": "%h server (Samba, Ubuntu)",
        "workgroup": "WORKGROUP",
        "interfaces": "",
        "bind_interfaces_only": true,
        "security": "user",
        "log_file": "/var/log/samba/log.%m"
    },
    "shares": [
        {
            "comment": "All Printers",
            "browseable": false,
            "create_mask": "0700",
            "name": "printers",
            "directory_mask": "0755",
            "read_only": true,
            "guest_ok": false,
            "path": "/var/spool/samba"
        },
        {
            "comment": "Printer Drivers",
            "browseable": true,
            "create_mask": "0744",
            "name": "print$",
            "directory_mask": "0755",
            "read_only": true,
            "guest_ok": false,
            "path": "/var/lib/samba/printers"
        }
    ]
}

>>> data_tree.shares
<reconfigure.items.bound.BoundCollection object at 0x23d0610>
>>> [_.path for _ in data_tree.shares]
['/var/spool/samba', '/var/lib/samba/printers']
```

Data trees may consist of any Python objects, but the common approach is to use *Bound Data*

Data trees can be manipulated as you wish:

```
>>> from reconfigure.items.samba import ShareData
>>> share = ShareData()
>>> share.path = '/home/user'
>>> share.comment = 'New share'
>>> data_tree.shares.append(share)
>>> data_tree
{
    ....
    "shares": [
    {
        "comment": "All Printers",
        "browseable": false,
        "create_mask": "0700",
        "name": "printers",
        "directory_mask": "0755",
        "read_only": true,
```

```
        "guest_ok": false,
        "path": "/var/spool/samba"
    },
    {
        "comment": "Printer Drivers",
        "browseable": true,
        "create_mask": "0744",
        "name": "print$",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": false,
        "path": "/var/lib/samba/printers"
    },
    {
        "comment": "New share",
        "browseable": true,
        "create_mask": "0744",
        "name": "share",
        "directory_mask": "0755",
        "read_only": true,
        "guest_ok": false,
        "path": "/home/user"
    }
]
```

After you're done with the modifications, the data tree must be converted back to the node tree:

```
>>> node_tree = builder.unbuild(data_tree)
```

Bound Data

Bound data (*reconfigure.items.bound.BoundData*) is a special class that can be subclassed and stuffed with properties, which will act as proxies to an underlying *Node tree*. This can be confusing, so let's go with an example:

```
>>> from reconfigure.nodes import Node, PropertyNode
>>> from reconfigure.items.bound import BoundData
>>>
>>> node = Node('test')
>>> node.append(PropertyNode('name', 'Alice'))
>>> node.append(PropertyNode('age', '25'))
>>> node.append(PropertyNode('gender', 'f'))
>>> print node
(test)
        name = Alice
        age = 25
        gender = f
```

Here we have a very simple *Node tree*. Note that all values are `str` and the `gender` is coded in a single character (we have probably parsed this tree from some .ini file). Now let's define a BoundData class:

```
>>> class HumanData (BoundData):
...     pass
...
>>> HumanData.bind_property('name', 'name')
>>> HumanData.bind_property('age', 'age', getter=int, setter=str)
```

```
>>> HumanData.bind_property('gender', 'gender',
...     getter=lambda x: 'Male' if x == 'm' else 'Female',
...     setter=lambda x: 'm' if x == 'Male' else 'f')

>>> human = HumanData(node)
>>> human
<__main__.MyData object at 0x114ddd0>
>>> print human
{
    "gender": "Female",
    "age": 25,
    "name": "Alice"
}
```

First, we've defined our `BoundData` subclass. Then, we have defined three properties in it:

- `name` is the simplest property, it's directly bound to "name" child `PropertyNode`
- `age` also has a getter and setter. These are invoked when the property is read or written. In this case, we use `int()` to parse a number from the node tree and `str()` to stringify it when writing back.
- `gender` is similar to `age` but has more complex getter and setter that transform "m" and "f" to a human-readable description.

When the properties are mutated, the modifications are applied to Node tree immediately and vice versa:

```
>>> human.age
25
>>> human.age = 30
>>> node.get('age').value
'30'
>>> node.get('age').value = 27
>>> human.age
27
```

Using collections

Let's try a more complex node tree:

```
>>> nodes = Node('',
...     Node('Alice',
...             PropertyNode('Phone', '1234-56-78')
...     ),
...     Node('Bob',
...             PropertyNode('Phone', '8765-43-21')
...     )
... )
>>> print nodes
()
        (Alice)
                Phone = 1234-56-78
        (Bob)
                Phone = 8765-43-21
```

Bound data classes:

```
>>> class PersonData (BoundData):
...     def template(self, name, phone):
```

```
...             return Node(name,
...                     PropertyNode('Phone', phone)
...             )
...
>>> class PhonebookData (BoundData):
...     pass
...
>>> PersonData.bind_property('Phone', 'phone')
>>> PersonData.bind_name('name')
>>>
>>> PhonebookData.bind_collection('entries', item_class=PersonData)
>>>
>>> phonebook = PhonebookData(nodes)
>>> print phonebook
{
    "entries": [
        {
            "phone": "1234-56-78",
            "name": "Alice"
        },
        {
            "phone": "8765-43-21",
            "name": "Bob"
        }
    ]
}
```

Here, `bind_collection` method is used to create a collection property from child nodes. `item_class` class will be used to wrap these nodes.

Alternatively, you can employ *`reconfigure.items.bound.BoundDictionary`* class to create a dict-like property:

```
>>> PhonebookData.bind_collection('entries', collection_class=BoundDictionary, item_
↪class=PersonData, key=lambda x: x.name)
>>> print phonebook
{
    "entries": {
        "Bob": {
            "phone": "8765-43-21",
            "name": "Bob"
        },
        "Alice": {
            "phone": "1234-56-78",
            "name": "Alice"
        }
    }
}
```

Components

Parsers

Parsers are *`reconfigure.parsers.BaseParser`* subclasses which transform *raw config content* into *node trees* and vice versa

Making your own parser is as easy as subclassing *reconfigure.parsers.BaseParser* and overriding `parse` and `stringify` methods.

Includers

Includers are used to handle the "include" directives in config files. Includers assemble the config file by finding the included files and parsing them and attaching them to the *node tree* of the main config. Reconfigure keeps track of which node belongs to which file by setting `origin` attribute on the included nodes

Example of includer in action:

```
>>> from reconfigure.parsers import *
>>> from reconfigure.includers import *
>>> parser = IniFileParser()
>>> includer = SupervisorIncluder(parser)
>>> nodes = parser.parse(open('/etc/supervisor/supervisord.conf').read())
>>> print nodes
(None)
        (unix_http_server)
                file = /var/run//supervisor.sock ((the path to the socket file))
                chmod = 0700 (sockef file mode (default 0700))
        (supervisord)
                logfile = /var/log/supervisor/supervisord.log ((main log file;default
↪$CWD/supervisord.log))
                pidfile = /var/run/supervisord.pid ((supervisord pidfile;default␣
↪supervisord.pid))
                childlogdir = /var/log/supervisor (('AUTO' child log dir, default
↪$TEMP))
        (rpcinterface:supervisor)
                supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_
↪rpcinterface
        (supervisorctl)
                serverurl = unix:///var/run//supervisor.sock (use a unix:// URL  for␣
↪a unix socket)
        (include)
                files = /etc/supervisor/conf.d/*.conf
```

Note the "include" node in the end. Now we'll run an includer over this tree:

```
>>> nodes = includer.compose('/etc/supervisor/supervisord.conf', nodes)
>>> print nodes
(None)
        (unix_http_server)
                file = /var/run//supervisor.sock ((the path to the socket file))
                chmod = 0700 (sockef file mode (default 0700))
        (supervisord)
                logfile = /var/log/supervisor/supervisord.log ((main log file;default
↪$CWD/supervisord.log))
                pidfile = /var/run/supervisord.pid ((supervisord pidfile;default␣
↪supervisord.pid))
                childlogdir = /var/log/supervisor (('AUTO' child log dir, default
↪$TEMP))
        (rpcinterface:supervisor)
                supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_
↪rpcinterface
        (supervisorctl)
                serverurl = unix:///var/run//supervisor.sock (use a unix:// URL  for␣
↪a unix socket)
```

```
        <include> /etc/supervisor/conf.d/*.conf
        (program:test)
                command = cat
```

Note how the include directive has turned into a junction point (*reconfigure.nodes.IncludeNode*) and content of included files was parsed and attached.

Calling `decompose` method will split the tree back into separate files:

```
>>> includer.decompose(nodes)
{
    '/etc/supervisor/conf.d/1.conf': <reconfigure.nodes.RootNode object at 0x2c5cf10>,
    '/etc/supervisor/supervisord.conf': <reconfigure.nodes.RootNode object at␣
↪0x2c5cb50>
}
```

Writing your own includer

If you're up to writing a custom includer, take a look at *reconfigure.includers.AutoIncluder*. It already implements the tree-walking and attachment logic, so you only need to implement two methods:

- `is_include(node)`: should check if the `node` is an include directive for this file format, and if it is, return a glob (wildcard) or path to the included files
- `remove_include(include_node)`: given an *reconfigure.nodes.IncludeNode*, should transform it back into file-format-specific include directive and return it (as a *node tree* chunk)

Builders

Builders transform *node trees* into *data trees*.

To write your own builder, subclass *reconfigure.builders.BaseBuilder* and override `build` and `unbuild` methods.

Reconfig objects

`reconfigure.config.Reconfig` objects are pre-set pipelines connecting *Parsers*, *Includers* and *Builders*

Reconfigure comes with many Reconfig objects out-of-the-box - see *reconfigure.configs*

Writing your Reconfig subclass

Use the following pattern:

```
class <name>Config (Reconfig):
    """
    <description>
    """

    def __init__(self, **kwargs):
        k = {
            'parser': <parser-class>(),
            'includer': <includer-class>(),
            'builder': BoundBuilder(<root-data-class>),
```

```
        }
        k.update(kwargs)
        Reconfig.__init__(self, **k)
```

Example:

```
class SupervisorConfig (Reconfig):
    """
    ``/etc/supervisor/supervisord.conf``
    """

    def __init__(self, **kwargs):
        k = {
            'parser': IniFileParser(),
            'includer': SupervisorIncluder(),
            'builder': BoundBuilder(SupervisorData),
        }
        k.update(kwargs)
        Reconfig.__init__(self, **k)
```

CHAPTER 3

API Reference:

reconfigure.configs

Configs are ready-to-use objects that link together Parsers, Includers and Builders to provide direct conversion between config files and Data tree.

class reconfigure.configs.**Reconfig**(*parser=None*, *includer=None*, *builder=None*, *path=None*, *content=None*)

Basic config class. Derivatives normally only need to override the constructor.

Config data is loaded either from `path` or from `content`

Parameters

- **parser** – overrides the Parser instance
- **includer** – overrides the Includer instance
- **builder** – overrides the Builder instance
- **path** – config file path. Not compatible with `content`
- **content** – config file content. Not compatible with `path`

load()

Loads the config data, parses and builds it. Sets `tree` attribute to point to Data tree.

save()

Unbuilds, stringifies and saves the config. If the config was loaded from string, returns `{ origin: data }` dict

class reconfigure.configs.**AjentiConfig**(***kwargs*)

class reconfigure.configs.**BIND9Config**(***kwargs*)

`named.conf`

class reconfigure.configs.**CrontabConfig**(***kwargs*)

class reconfigure.configs.**CTDBConfig**(***kwargs*)
CTDB main config

class reconfigure.configs.**CTDBNodesConfig**(***kwargs*)
CTDB node list file

class reconfigure.configs.**CTDBPublicAddressesConfig**(***kwargs*)
CTDB public address list file

class reconfigure.configs.**DHCPDConfig**(***kwargs*)
DHCPD

class reconfigure.configs.**ExportsConfig**(***kwargs*)
/etc/fstab

class reconfigure.configs.**FSTabConfig**(***kwargs*)
/etc/fstab

class reconfigure.configs.**GroupConfig**(***kwargs*)
/etc/group

class reconfigure.configs.**HostsConfig**(***kwargs*)
/etc/hosts

class reconfigure.configs.**IPTablesConfig**(***kwargs*)
iptables-save and iptables-restore

class reconfigure.configs.**NetatalkConfig**(***kwargs*)
Netatalk afp.conf

class reconfigure.configs.**NSDConfig**(***kwargs*)
NSD DNS server nsd.conf

class reconfigure.configs.**PasswdConfig**(***kwargs*)
/etc/passwd

class reconfigure.configs.**ResolvConfig**(***kwargs*)
/etc/resolv.conf

class reconfigure.configs.**SambaConfig**(***kwargs*)

class reconfigure.configs.**SquidConfig**(***kwargs*)

class reconfigure.configs.**SupervisorConfig**(***kwargs*)
/etc/supervisor/supervisord.conf

reconfigure.parsers

class reconfigure.parsers.**BaseParser**
A base parser class

parse(*content*)

Parameters **content** – string config content

Returns a *reconfigure.nodes.Node* tree

stringify(*tree*)

Parameters **tree** – a *reconfigure.nodes.Node* tree

Returns string config content

class reconfigure.parsers.**BIND9Parser**
A parser for named.conf

token_section_end = '};'

tokens = [('(acl|key|masters|server|trusted-keys|managed-keys|controls|logging|lwres|options|view|zone|channel|category

class reconfigure.parsers.**CrontabParser**(*remove_comments=False*)

parse(*content*)

stringify(*tree*)

stringify_env_setting(*node*)

stringify_normal_task(*node*)

stringify_special_task(*node*)

class reconfigure.parsers.**ExportsParser**(**args*, ***kwargs*)
A parser for NFS' /etc/exports

parse(*content*)

stringify(*tree*)

class reconfigure.parsers.**IniFileParser**(*sectionless=False*, *nullsection='__default__'*)
A parser for standard `.ini` config files.

Parameters **sectionless** – if `True`, allows a section-less attributes appear in the beginning of file

parse(*content*)

stringify(*tree*)

class reconfigure.parsers.**IPTablesParser**
A parser for `iptables` configuration as produced by `iptables-save`

parse(*content*)

stringify(*tree*)

class reconfigure.parsers.**JsonParser**
A parser for JSON files (using `json` module)

load_node_rec(*node*, *json*)

parse(*content*)

save_node_rec(*node*)

stringify(*tree*)

class reconfigure.parsers.**NginxParser**
A parser for nginx configs

parse(*content*)

stringify(*tree*)

stringify_rec(*node*)

token_comment = '#'

token_section_end = '}'

tokens = [('[\\w_]+\\s*?[^\\n]*?{', <function <lambda>>), ('[\\w_]+?.+?;', <function <lambda>>), ('\\s', <function <lamb

class reconfigure.parsers.**NSDParser**
A parser for NSD DNS server nsd.conf file

parse(*content*)

stringify(*tree*)

stringify_comment(*line*, *comment*)

class reconfigure.parsers.**ShellParser**(**args*, ***kwargs*)
A parser for shell scripts with variables

parse(*content*)

stringify(*tree*)

class reconfigure.parsers.**SSVParser**(*separator=None*, *maxsplit=-1*, *comment='#'*, *continuation=None*, **args*, ***kwargs*)
A parser for files containing space-separated value (notably, `/etc/fstab` and friends)

Parameters

- **separator** – separator character, defaults to whitespace
- **maxsplit** – max number of tokens per line, defaults to infinity
- **comment** – character denoting comments
- **continuation** – line continuation character, None to disable

parse(*content*)

stringify(*tree*)

class reconfigure.parsers.**SquidParser**
A parser for Squid configs

parse(*content*)

stringify(*tree*)

reconfigure.nodes

class reconfigure.nodes.**IncludeNode**(*files*)
A node that indicates a junction point between two config files

class reconfigure.nodes.**Node**(*name=None*, **args*, ***kwargs*)
A base node class for the Node Tree. This class represents a named container node.

append(*node*)

get(*name*, *default=None*)

Returns a child node by its name or `default`

get_all(*name*)

Returns list of child nodes with supplied `name`

indexof(*node*)

Returns index of the node in the children array or `None` if it's not a child

remove(*node*)

replace(*name*, *node=None*)
Replaces the child nodes by `name`

Parameters **node** – replacement node or list of nodes

```
n.append(Node('a'))
n.append(Node('a'))
n.replace('a', None)
assert(len(n.get_all('a')) == 0)
```

set_property(*name*, *value*)
Creates or replaces a child *PropertyNode* by name.

class `reconfigure.nodes.`**PropertyNode**(*name*, *value*, *comment=None*)
A node that serves as a property of its parent node.

class `reconfigure.nodes.`**RootNode**(*name=None*, **args*, ***kwargs*)
A special node class that indicates tree root

reconfigure.includers

class `reconfigure.includers.`**BaseIncluder**(*parser=None*, *content_map={}*)
A base includer class

Parameters

- **parser** – Parser instance that was used to parse the root config file
- **content_map** – a dict that overrides config content for specific paths

compose(*origin*, *tree*)
Should locate the include nodes in the Node tree, replace them with *reconfigure.nodes.IncludeNode*, parse the specified include files and append them to tree, with correct node `origin` attributes

decompose(*origin*, *tree*)
Should detach the included subtrees from the Node tree and return a `{ origin: content-node-tree }` dict.

class `reconfigure.includers.`**AutoIncluder**(*parser=None*, *content_map={}*)
This base includer automatically walks the node tree and loads the include files from `IncludeNode.files` properties. `files` is supposed to contain absolute path, relative path or a shell wildcard.

compose(*origin*, *tree*)

compose_rec(*root*, *origin*, *node*)

decompose(*tree*)

decompose_rec(*node*, *result*)

is_include(*node*)
Should return whether the node is an include node and return file pattern glob if it is

remove_include(*node*)
Shoud transform *reconfigure.nodes.IncludeNode* into a normal Node to be stringified into the file

class `reconfigure.includers.`**BIND9Includer**(*parser=None*, *content_map={}*)

is_include(*node*)

remove_include(*node*)

class reconfigure.includers.**NginxIncluder**(*parser=None*, *content_map={}*)

is_include(*node*)

remove_include(*node*)

class reconfigure.includers.**SupervisorIncluder**(*parser=None*, *content_map={}*)

is_include(*node*)

remove_include(*node*)

reconfigure.builders

Builders are used to convert Node Tree to Data Tree

class reconfigure.builders.**BaseBuilder**
A base class for builders

build(*tree*)

Parameters **tree** – *reconfigure.nodes.Node* tree

Returns Data tree

unbuild(*tree*)

Parameters **tree** – Data tree

Returns *reconfigure.nodes.Node* tree

class reconfigure.builders.**BoundBuilder**(*root_class*)
A builder that uses *reconfigure.items.bound.BoundData* to build stuff

Parameters **root_class** – a BoundData class that used as processing root

build(*nodetree*)

unbuild(*tree*)

reconfigure.items.bound

class reconfigure.items.bound.**BoundCollection**(*node*, *item_class*, *selector=<function <lambda>>*)
Binds a list-like object to a set of nodes

Parameters

- **node** – target node (its children will be bound)
- **item_class** – *BoundData* class for items
- **selector** – lambda x: bool, used to filter out a subset of nodes

append(*item*)

insert(*index*, *item*)

pop(*index*)

rebuild()
Discards cached collection and rebuilds it from the nodes

remove(*item*)

to_dict()

to_json()

class reconfigure.items.bound.**BoundData**(*node=None*, ***kwargs*)
Binds itself to a node.

`bind_*` classmethods should be called on module-level, after subclass declaration.

Parameters

- **node** – all bindings will be relative to this node
- **kwargs** – if `node` is `None`, `template(**kwargs)` will be used to create node tree fragment

classmethod bind(*data_property*, *getter*, *setter*)
Creates an arbitrary named property in the class with given getter and setter. Not usually used directly.

Parameters

- **data_property** – property name
- **getter** – `lambda: object`, property getter
- **setter** – `lambda value: None`, property setter

classmethod bind_attribute(*node_attribute*, *data_property*, *default=None*, *path=<function <lambda>>*, *getter=<function <lambda>>*, *setter=<function <lambda>>*)
Binds the value of node object's attribute to a property

Parameters

- **node_attribute** – `Node`'s attribute name
- **data_property** – property name to be created
- **default** – default value of the property (is `PropertyNode` doesn't exist)
- **path** – `lambda self.node: PropertyNode`, can be used to point binding to another Node instead of `self.node`.
- **getter** – `lambda object: object`, used to transform value when getting
- **setter** – `lambda object: object`, used to transform value when setting

classmethod bind_child(*data_property*, *path=<function <lambda>>*, *item_class=None*)
Directly binds a child Node to a BoundData property

Parameters

- **data_property** – property name to be created
- **path** – `lambda self.node: PropertyNode`, can be used to point binding to another Node instead of `self.node`.
- **item_class** – a *BoundData* subclass to be used for the property value

classmethod bind_collection(*data_property*, *path=<function <lambda>>*, *selector=<function <lambda>>*, *item_class=None*, *collection_class=<class 'reconfigure.items.bound.BoundCollection'>*, ***kwargs*)
Binds the subset of node's children to a collection property

Parameters

- **data_property** – property name to be created
- **path** – `lambda self.node: PropertyNode`, can be used to point binding to another Node instead of `self.node`.
- **selector** – `lambda Node: bool`, can be used to filter out a subset of child nodes
- **item_class** – a *BoundData* subclass to be used for collection items
- **collection_class** – a *BoundCollection* subclass to be used for collection property itself

classmethod bind_name(*data_property*, *getter=<function <lambda>>*, *setter=<function <lambda>>*)
Binds the value of node's `name` attribute to a property

Parameters

- **data_property** – property name to be created
- **getter** – `lambda object: object`, used to transform value when getting
- **setter** – `lambda object: object`, used to transform value when setting

classmethod bind_property(*node_property*, *data_property*, *default=None*, *default_remove=[]*, *path=<function <lambda>>*, *getter=<function <lambda>>*, *setter=<function <lambda>>*)
Binds the value of a child `reconfigure.node.PropertyNode` to a property

Parameters

- **node_property** – `PropertyNode`'s `name`
- **data_property** – property name to be created
- **default** – default value of the property (is `PropertyNode` doesn't exist)
- **default_remove** – if setting a value contained in default_remove, the target property is removed
- **path** – `lambda self.node: PropertyNode`, can be used to point binding to another Node instead of `self.node`.
- **getter** – `lambda object: object`, used to transform value when getting
- **setter** – `lambda object: object`, used to transform value when setting

template(***kwargs*)
Override to create empty objects.

Returns a *reconfigure.nodes.Node* tree that will be used as a template for new BoundData instance

to_dict()

to_json()

class `reconfigure.items.bound.`**BoundDictionary**(*key=None*, ***kwargs*)
Binds a dict-like object to a set of nodes. Accepts same params as *BoundCollection* plus `key`

Parameters **key** – `lambda value: object`, is used to get key for value in the collection

items()

iteritems()

pop(*key*)

rebuild()

rebuild_dict()

setdefault(*k*, *v*)

to_dict()

update(*other*)

values()

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

J

L

N

P

R

S

T

U

V