BundleWrap Documentation

Release 1.1.0

Torsten Rehn

August 12, 2014

Contents

Check out the *quickstart tutorial* to get started.

CHAPTER 1

Is BundleWrap the right tool for you?

We think you will enjoy BundleWrap a lot if you:

- know some Python
- like to write your configuration from scratch and control every bit of it
- have lots of unique nodes
- are trying to get a lot of existing systems under management
- are NOT trying to handle a massive amount of nodes (let's say more than 300)
- like to start small
- don't want yet more stuff to run on your nodes (or mess with appliances as little as possible)
- prefer a simple tool to a fancy one
- want as much as possible in git/hg/bzr
- have strongly segmented internal networks

You might be better served with a different config management system if you:

- are already using a config management system and don't have any major issues
- hate Python and/or JSON
- like to use community-maintained configuration templates
- need unattended bootstrapping of nodes
- need to manage non-Linux systems
- don't trust your coworkers

We have also prepared a *comparison* with other popular config management systems.

1.1 Quickstart

This is the 10 minute intro into BundleWrap. Fasten your seatbelt.

1.1.1 Installation

First, open a terminal and install BundleWrap:

```
pip install bundlewrap
```

1.1.2 Create a repository

Now you'll need to create your *repository*:

```
mkdir my_bundlewrap_repo
cd my_bundlewrap_repo
bw repo create
```

You will note that some files have been created. Let's check them out:

```
cat nodes.py
cat groups.py
```

The contents should be fairly self-explanatory, but you can always check the *docs* on these files if you want to go deeper.

Hint: It is highly recommended to use git or another SCM (Source Code Management) tool to keep track of your repository. You may want to start doing that right away.

At this point you will want to edit `nodes.py` and maybe change "localhost" to the hostname of a system you have passwordless (including sudo) SSH access to.

Note: BundleWrap will honor your `~/.ssh/config`, so if `ssh mynode.example.com sudo id` works without any password prompts in your terminal, you're good to go.

If you need a password for SSH and/or sudo, please add *-p* directly after **bw** when calling **bw run** or **bw apply**.

1.1.3 Run a command

The first thing you can do is run a command on your army of one *node*:

```
bw run node1 "uptime"
```

You should see something like this:

```
[node1] out:  17:23:19 up 97 days,  2:51,  2 users,  load average: 0.08, 0.03, 0.05
[node1] out:
[node1]  completed successfully after 1.18188s
```

Instead of a node name ("node1" in this case) you can also use a *group* name (such as "all") from your `groups.py`.

1.1.4 Create a bundle

BundleWrap stores node configuration in *bundles*. A bundle is a collection of *items* such as files, system packages or users. To create your first bundle, type:

```
bw repo bundle create mybundle
```

Now that you have created your bundle, it's important to tell BundleWrap which nodes will have this bundle. You can assign bundles to nodes using either `groups.py` or `nodes.py`, here we'll use the latter:

```
nodes = {
    'node1': {
        'bundles': (
            "mybundle",
        ),
        'hostname': "mynode1.local",
    },
}
```

1.1.5 Create a file template

To manage a file, you need two things:

1. a file item in your bundle
2. a template for the file contents

Add this to your `bundles/mybundle/bundle.py`:

```
files = {
    '/etc/motd': {
        'source': "etc/motd",
    },
}
```

Then write the file template:

```
mkdir bundles/mybundle/files/etc
vim bundles/mybundle/files/etc/motd
```

You can use this for example content:

```
Welcome to ${node.name}!
```

Note that the "source" attribute in `bundle.py` contains a path relative to the `files` directory of your bundle. It's up to you how to organize the contents of this directory.

1.1.6 Apply configuration

Now all that's left is to run **bw apply**:

```
bw apply -i node1
```

BundleWrap will ask to replace your previous MOTD (message of the day):

```
node1: run started at 2013-11-16 18:26:29

   file:/etc/motd

    content
    --- /etc/motd
    +++ <bundlewrap content>
    @@ -1 +1 @@
    -your old motd
    +Welcome to node1!

   Fix file:/etc/motd? [Y/n]
```

That completes the quickstart tutorial!

1.1.7 Further reading

Here are some suggestions on what to do next:

- take a moment to think about what groups and bundles you will create
- read up on how a *BundleWrap repository* is laid out
- ...especially what types of items you can add to your *bundles*
- familiarize yourself with the Mako template language
- explore the *command line interface*
- follow @bundlewrap on Twitter

Have fun! If you have any questions, feel free to drop by on IRC.

1.2 Installation

Hint: You may need to install **pip** first. This can be accomplished through your distribution's package manager, e.g.:

```
aptitude install python-pip
```

or the manual instructions.

1.2.1 Using *pip*

It's as simple as:

```
pip install bundlewrap
```

Note that you need Python 2.7 to run BundleWrap.

1.2.2 From git

Warning: This type of install will give you the very latest (and thus possibly broken) bleeding edge version of BundleWrap. You should only use this if you know what you're doing.

Note: The instructions below are for installing on Ubuntu Server 12.10 (Quantal), but should also work for other versions of Ubuntu/Debian. If you're on some other distro, you will obviously have to adjust the package install commands.

Note: The instructions assume you have root privileges.

Install basic requirements:

```
aptitude install build-essential git python-dev python-distribute
```

Clone the GitHub repository:

```
cd /opt
git clone https://github.com/bundlewrap/bundlewrap.git
```

Use `setup.py` to install in "development mode":

```
cd /opt/bundlewrap
python setup.py develop
```

You can now try running the `bw` command line utility:

```
bw --help
```

That's it.

To update your install, just pull the git repository and have `setup.py` check for new dependencies:

```
cd /opt/bundlewrap
git pull
python setup.py develop
```

1.3 Requirements for managed systems

While the following list might appear long, even very minimal systems should provide everything that's needed.

- **apt-get** (only used with *pkg_apt* items)
- **base64**
- **bash**

- **cat**
- **chmod**
- **chown**
- **dpkg** (only used with *pkg_apt* items)
- **echo**
- **export**
- **file**
- **grep**
- **groupadd**
- **groupmod**
- **id**
- **initctl** (only used with *svc_upstart* items)
- **mkdir**
- **mv**
- **pacman** (only used with *pkg_pacman* items)
- **rm**
- **sed**
- sftp-enabled SSH server (your home directory must be writable)
- **sudo**
- **sha1sum**
- **systemctl** (only used with *svc_systemd* items)
- **test**
- **useradd**
- **usermod**

1.4 Repository reference

A BundleWrap repository contains everything you need to contruct the configuration for your systems.

This page describes the various subdirectories and files than can exist inside a repo.

Name	Documentation	Purpose
`nodes.py`	*nodes.py*	This file tells BundleWrap what nodes (servers, VMs, ...) there are in your environment and lets you configure options such as hostnames.
`groups.py`	*groups.py*	This file allows you to organize your nodes into groups.
`bundles/`	*Bundles*	This required subdirectory contains the bulk of your configuration, organized into bundles of related items.
`data/`		This optional subdirectory contains data files that are not generic enough to be included in bundles (which are meant to be shareable).
`hooks/`	*Hooks*	This optional subdirectory contains hooks you can use to act on certain events when using BundleWrap.
`items/`	*Custom item types*	This optional subdirectory contains the code for your custom item types.
`libs/`	*Custom code*	This optional subdirectory contains reusable custom code for your bundles.

1.4.1 nodes.py

This file lets you specify or dynamically build a list of nodes in your environment.

Introduction

All you have to do here is define a Python dictionary called `nodes`. It should look something like this:

```
nodes = {
    'node1': {
        'hostname': "node1.example.com",
    },
}
```

Note: With BundleWrap, the DNS name and the internal identifier for a node are two separate things. This allows for clean and sortable hierarchies:

```
nodes = {
    'cluster1.node1': {
        'hostname': "node1.cluster1.example.com",
    },
}
```

All fields for a node (including `hostname`) are optional. If you don't give one, BundleWrap will attempt to use the internal identifier to connect to a node:

```
nodes = {
    'node1.example.com': {},
}
```

Dynamic node list

You are not confined to the static way of defining a node list as shown above. You can also assemble the `nodes` dictionary dynamically:

```
def get_my_nodes_from_ldap():
    [...]
    return ldap_nodes

nodes = get_my_nodes_from_ldap()
```

Node attribute reference

This section is a reference for all possible attributes you can define for a node:

```
nodes = {
    'node1': {
        # THIS PART IS EXPLAINED HERE
    },
}
```

bundles

A list of *bundles* to be assigned to this node.

hostname

A string used as a DNS name when connecting to this node. May also be an IP address.

Note: The username and SSH private key for connecting to the node cannot be configured in BundleWrap. If you need to customize those, BundleWrap will honor your `~/.ssh/config`.

---

metadata

This can be a dictionary of arbitrary data. You can access it from your templates as `node.metadata`. Use this to attach custom data (such as a list of IP addresses that should be configured on the target node) to the node. Note that you can also define metadata at the *group level*, but node metadata has higher priority.

password

SSH and sudo password to use for this node. Overrides passwords set at the group level and on the command line.

> **Warning:** Please do not write any passwords into your `nodes.py`. This attribute is intended to be used with an external source of passwords and filled dynamically.

use_shadow_passwords

> **Warning:** Changing this setting will affect the security of the target system. Only do this for legacy systems that don't support shadow passwords.

This setting will affect how the *user* item operates. If set to `False`, password hashes will be written directly to `/etc/passwd` and thus be accessible to any user on the system.

1.4.2 groups.py

This file lets you specify or dynamically build a list of *groups* in your environment.

Introduction

As with `nodes.py`, you define your groups as a dictionary:

```
groups = {
    'all': {
        'member_patterns': (
            r".*",
        ),
    },
    'group1': {
        'members': (
            'node1',
        ),
    },
}
```

All group attributes are optional.

Group attribute reference

This section is a reference for all possible attributes you can define for a group:

```
groups = {
    'group1': {
        # THIS PART IS EXPLAINED HERE
    },
}
```

bundles

A list of *bundles* to be assigned to each node in this group.

member_patterns

A list of regular expressions. Node names matching these expressions will be added to the group members.

Matches are determined using the search() method.

members

A tuple or list of node names that belong to this group.

metadata

A dictionary of arbitrary data that will be accessible from each node's `node.metadata`. For each node, BundleWrap will merge the metadata of all of the node's groups first, then merge in the metadata from the node itself.

Warning: Be careful when defining conflicting metadata (i.e. dictionaries that have some common keys) in multiple groups. BundleWrap will consider group hierarchy when merging metadata. For example, it is possible to define a default nameserver for the "eu" group and then override it for the "eu.frankfurt" subgroup. The catch is that this only works for groups that are connected through a subgroup hierarchy. Independent groups will have their metadata merged in an undefined order.

metadata_processors

Note: This is an advanced feature. You should already be very familiar with BundleWrap before using this.

A list of strings formatted as `module.function` where *module* is the name of a file in the `libs/` *subdirectory* of your repo (without the `.py` extension) and *function* is the name of a function in that file.

This function can be used to dynamically manipulate metadata after the static metadata has been generated. It looks like this:

```
def example1(node_name, groups, metadata, **kwargs):
    if "group1" in groups and "group2" in groups:
        metadata['foo'].append("bar")
    return metadata
```

As you can see, the metadata processor function is passed the node name, a list of group names and the metadata dictionary generated so far. You can then manipulate that dictionary based on these parameters and must return the modified metadata dictionary.

password

SSH and sudo password to use for nodes in this group. Overrides password set on the command line.

Warning: Please do not write any passwords into your `groups.py`. This attribute is intended to be used with an external source of passwords and filled dynamically.

Warning: Be careful when defining passwords in groups that share one or more nodes. BundleWrap will consider group hierarchy when determining the password to use. For example, it is possible to define a default password for the "eu" group and then override it for the "eu.frankfurt" subgroup. The catch is that this only works for groups that are connected through a subgroup hierarchy. Passwords in independent groups will be chosen in an undefined way.

subgroups

A tuple or list of group names whose members should be recursively included in this group.

1.4.3 Bundles

Actions

Actions will be run on every **bw apply**. They differ from regular items in that they cannot be "correct" in the first place. They can only succeed or fail.

```
actions = {
    'check_if_its_still_linux': {
        'command': "uname",
        'expected_return_code': 0,
        'expected_stdout': "Linux\n",
```

```
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

command The only required attribute. This is the command that will be run on the node with root privileges.

expected_return_code Defaults to `0`. If the return code of your command is anything else, the action is considered failed. You can also set this to `None` and any return code will be accepted.

expected_stdout If this is given, the stdout output of the command must match the given string or the action is considered failed.

expected_stderr Same as `expected_stdout`, but with stderr.

interactive If set to `True`, this action will be skipped in non-interactive mode. If set to `False`, this action will always be executed without asking (even in interactive mode). Defaults to `None`.

> **Warning:** Think hard before setting this to `False`. People might assume that interactive mode won't do anything without their consent.

Directory items

```
directories = {
    "/path/to/directory": {
        "mode": "0644",
        "owner": "root",
        "group": "root",
```

```
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

group Name of the group this directory belongs to. Defaults to `None` (don't care about group).

mode Directory mode as returned by **stat -c %a <directory>**. Defaults to `None` (don't care about mode).

owner Username of the directory's owner. Defaults to `None` (don't care about owner).

File items

Writing file templates

BundleWrap uses Mako for file templating by default (Jinja2 is also available). This enables you to dynamically contruct your config files. Templates reside in the `files` subdirectory of a bundle and are bound to a file item using the `source` *attribute*. This page explains how to get started with Mako.

The most basic example would be:

```
Hello, this is ${node.name}!
```

After template rendering, it would look like this:

```
Hello, this is myexamplenodename!
```

As you can see, *${...}* can be used to insert the value of a context variable into the rendered file. By default, you have access to two variables in every template: *node* and *repo*. They are `bundlewrap.node.Node` and `bundlewrap.repo.Repository` objects, respectively. You can learn more about the attributes and methods of these objects *in the API docs*, but here are a few examples:

inserts the DNS hostname of the current node

```
${node.hostname}
```

a list of all nodes in your repo

```
% for node in repo.nodes:
${node.name}
% endfor
```

make exceptions for certain nodes

```
% if node.name == "node1":
option = foo
% elif node.name in ("node2", "node3"):
option = bar
% else:
option = baz
% endif
```

check for group membership

```
% if node.in_group("sparkle"):
enable_sparkles = 1
% endif
```

check for membership in any of several groups

```
% if node.in_any_group(("sparkle", "shiny")):
enable_fancy = 1
% endif
```

check for bundle

```
% if node.has_bundle("sparkle"):
enable_sparkles = 1
% endif
```

check for any of several bundles

```
% if node.has_any_bundle(("sparkle", "shiny")):
enable_fancy = 1
% endif
```

list all nodes in a group

```
% for gnode in repo.get_group("mygroup").nodes:
${gnode.name}
% endfor
```

Working with node metadata

Quite often you will attach custom metadata to your nodes in `nodes.py`, e.g.:

```
nodes = {
        "node1": {
                "metadata": {
                        "interfaces": {
                                "eth0": "10.1.1.47",
                                "eth1": "10.1.2.47",
                        },
                },
        },
}
```

You can easily access this information in templates:

```
% for interface, ip in node.metadata["interfaces"].items():
interface ${interface}
        ip = ${ip}
% endfor
```

This template will render to:

```
interface eth0
        ip = 10.1.1.47
interface eth1
        ip = 10.1.2.47
```

Manages regular files.

```
files = {
    "/path/to/file": {
        "mode": "0644",
        "owner": "root",
        "group": "root",
        "content_type": "mako",
        "encoding": "utf-8",
        "source": "my_template",
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

content May be used instead of `source` to provide file content without a template file. Must be a UTF-8 string. Defaults to `""`.

content_type How the file pointed to by `source` or the string given to `content` should be interpreted.

Value	Effect
`any`	only cares about file owner, group, and mode
`binary`	file is uploaded verbatim, no content processing occurs
`jinja2`	content is interpreted by the Jinja2 template engine
`mako` (default)	content is interpreted by the Mako template engine
`text`	like `binary`, but will be diffed in interactive mode

Note: In order to use Jinja2, you'll also need to install it manually, since BundleWrap doesn't explicitly depend on it:

```
pip install Jinja2
```

context Only used with Mako templates. The values of this dictionary will be available from within the template as variables named after the respective keys.

delete When set to `True`, the path of this file will be removed. It doesn't matter if there is not a file but a directory or something else at this path. When using `delete`, no other attributes are allowed.

encoding Encoding of the target file. Note that this applies to the remote file only, your template is still conveniently written in UTF-8 and will be converted by BundleWrap. Defaults to "utf-8". Other possible values (e.g. "latin-1") can be found here.

group Name of the group this file belongs to. Defaults to `None` (don't care about group).

mode File mode as returned by **stat -c %a <file>**. Defaults to `None` (don't care about mode).

owner Username of the file's owner. Defaults to `None` (don't care about owner).

source File name of the file template. If this says `my_template`, BundleWrap will look in `data/my_bundle/files/my_template` and then `bundles/my_bundle/files/my_template`. Most of the time, you will want to put config templates into the latter directory. The `data/` subdirectory is meant for files that are very specific to your infrastructure (e.g. DNS zone files). This separation allows you to write your bundles in a generic way so that they could be open-sourced and shared with other people.

See also:

Writing file templates

Group items

Manages system groups. Group members are managed through the *user* item.

```
groups = {
    "acme": {
        "gid": 2342,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

delete When set to `True`, this group will be removed from the system. When using `delete`, no other attributes are allowed.

gid Numerical ID of the group.

APT package items

Handles packages installed by `apt-get` on Debian-based systems.

```
pkg_apt = {
    "foopkg": {
        "installed": True,  # default
    },
    "bar": {
        "installed": False,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

Optional attributes

installed `True` when the package is expected to be present on the system; `False` if it should be purged.

Pacman package items

Handles packages installed by **pacman** (e.g., Arch Linux).

```
pkg_pacman = {
    "foopkg": {
        "installed": True,  # default
    },
    "bar": {
        "installed": False,
    },
    "somethingelse": {
        "tarball": "something-1.0.pkg.tar.gz",
    }
}
```

Warning: System updates on Arch Linux should *always* be performed manually and with great care. Thus, this item type installs packages with a simple `pacman -S $pkgname` instead of the commonly recommended `pacman -Syu $pkgname`. You should *manually* do a full system update before installing new packages via BundleWrap!

Attribute reference

See also:

The list of generic builtin item attributes

Optional attributes

`installed` `True` when the package is expected to be present on the system; `False` if this package and all dependencies that are no longer needed should be removed.

`tarball` Upload a local file to the node and install it using **pacman -U**. The value of `tarball` must point to a file relative to the `pkg_pacman` subdirectory of the current bundle.

Upstart service items

Handles services managed by Upstart.

```
svc_upstart = {
    "gunicorn": {
        "running": True,  # default
    },
    "celery": {
        "running": False,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

Optional attributes

`running` `True` if the service is expected to be running on the system; `False` if it should be stopped.

Canned actions

See also:

Explanation of how canned actions work

reload Reloads the service.

restart Restarts the service.

systemd service items

Handles services managed by systemd.

```
svc_systemd = {
    "fcron.service": {
        "running": True,  # default
    },
    "sgopherd.socket": {
        "running": False,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

Optional attributes

running `True` if the service is expected to be running on the system; `False` if it should be stopped.

Canned actions

See also:

Explanation of how canned actions work

reload Reloads the service.

restart Restarts the service.

System V service items

Handles services managed by traditional System V init scripts.

```
svc_systemv = {
    "apache2": {
        "running": True,  # default
    },
    "mysql": {
        "running": False,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

Optional attributes

`running` `True` if the service is expected to be running on the system; `False` if it should be stopped.

Canned actions

See also:

Explanation of how canned actions work

`reload` Reloads the service.

`restart` Restarts the service.

Symlink items

```
symlinks = {
    "/some/symlink": {
        "group": "root",
        "owner": "root",
        "target": "/target/file",
```

```
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

Required attributes

target File or directory this symlink points to. This attribute is required.

Optional attributes

group Name of the group this symlink belongs to. Defaults to `root`. Defaults to `None` (don't care about group).

owner Username of the symlink's owner. Defaults to `root`. Defaults to `None` (don't care about owner).

User items

Manages system user accounts.

```
users = {
    "jdoe": {
        "full_name": "Jane Doe",
        "gid": 2342,
        "groups": ["admins", "users", "wheel"],
        "home": "/home/jdoe",
        "password_hash": "$6$abcdef$ghijklmnopqrstuvwxyz",
        "shell": "/bin/zsh",
        "uid": 4747,
    },
}
```

Attribute reference

See also:

The list of generic builtin item attributes

All attributes are optional.

delete When set to `True`, this user will be removed from the system. Note that because of how **userdel** works, the primary group of the user will be removed if it contains no other users. When using `delete`, no other attributes are allowed.

full_name Full name of the user.

gid Primary group of the user as numerical ID or group name.

Note: Due to how `useradd` works, this attribute is required whenever you *don't* want the default behavior of `useradd` (usually that means automatically creating a group with the same name as the user). If you want to use an unmanaged group already on the node, you need this attribute. If you want to use a group managed by BundleWrap, you need this attribute. This is true even if the groups mentioned are in fact named like the user.

---

groups List of groups (names, not GIDs) the user should belong to. Must NOT include the group referenced by `gid`.

hash_method One of:

- `md5`
- `sha256`
- `sha512`

Defaults to `sha512`.

home Path to home directory. Defaults to `/home/USERNAME`.

password The user's password in plaintext.

> **Warning:** Please do not write any passwords into your bundles. This attribute is intended to be used with an external source of passwords and filled dynamically. If you don't have or want such an elaborate setup, specify passwords using the `password_hash` attribute instead.

Note: If you don't specify a `salt` along with the password, BundleWrap will use a static salt. Be aware that this is basically the same as using no salt at all.

password_hash Hashed password as it would be returned by `crypt()` and written to `/etc/shadow`.

salt Recommended for use with the `password` attribute. BundleWrap will use 5000 rounds of SHA-512 on this salt and the provided password.

shell Path to login shell executable.

uid Numerical user ID. It's your job to make sure it's unique.

Bundles are subdirectories of the `bundles/` directory of your BundleWrap repository. Within each bundle, there must be a file called `bundle.py`. They define any number of magic attributes that are automatically processed by BundleWrap. Each attribute is a dictionary mapping an item name (such as a file name) to a dictionary of attributes (e.g. file ownership information).

A typical bundle might look like this:

```
files = {
    '/etc/hosts': {
        'owner': "root",
        'group': "root",
        'mode': "0664",
        [...]
    },
}
```

```
users = {
    'janedoe': {
        'home': "/home/janedoe",
        'shell': "/bin/zsh",
        [...]
    },
    'johndoe': {
        'home': "/home/johndoe",
        'shell': "/bin/bash",
        [...]
    },
}
```

This bundle defines the attributes `files` and `users`. Within the `users` attribute, there are two `user` items. Each item maps its name to a dictionary that is understood by the specific kind of item. Below you will find a reference of all builtin item types and the attributes they understand. You can also *define your own item types*.

Item types

This table lists all item types included in BundleWrap along with the bundle attributes they understand.

Type name	Bundle attribute	Purpose
action	`actions`	Actions allow you to run commands on every `bw apply`
directory	`directories`	Manages permissions and ownership for directories
file	`files`	Manages contents, permissions, and ownership for files
group	`groups`	Manages groups by wrapping `groupadd`, `groupmod` and `groupdel`
pkg_apt	`pkg_apt`	Installs and removes packages with APT
pkg_pacman	`pkg_pacman`	Installs and removes packages with pacman
svc_upstart	`svc_upstart`	Starts and stops services with Upstart
svc_systemd	`svc_systemd`	Starts and stops services with systemd
svc_systemv	`svc_systemv`	Starts and stops services with traditional System V init scripts
symlink	`symlinks`	Manages symbolic links and their ownership
user	`users`	Manages users by wrapping `useradd`, `usermod` and `userdel`

Builtin attributes

There are also attributes that can be applied to any kind of item.

needs

One such attribute is `needs`. It allows for setting up dependencies between items. This is not something you will have to to very often, because there are already implicit dependencies between items types (e.g. all files depend on all users). Here are two examples:

```
my_items = {
    'item1': {
        [...]
```

```
        'needs': [
            'file:/etc/foo.conf',
        ],
    },
    'item2': {
        ...
        'needs': [
            'pkg_apt:',
            'bundle:foo',
        ],
    }
}
```

The first item (`item1`, specific attributes have been omitted) depends on a file called `/etc/foo.conf`, while `item2` depends on all APT packages being installed and every item in the foo bundle.

needed_by

This attribute is an alternative way of defining dependencies. It works just like `needs`, but in the other direction. There are only three scenarios where you should use `needed_by` over `needs`:

- if you need all items a certain type to depend on something or
- if you need all items in a bundle to depend on something or
- if you need an item in a bundle you can't edit (e.g. because it's provided by a community-maintained *plugin*) to depend on something in your bundles

triggers and triggered

In some scenarios, you may want to execute an *action* only when an item is fixed (e.g. restart a daemon after a config file has changed or run `postmap` after updating an alias file). To do this, BundleWrap has the builtin atttribute `triggers`. You can use it to point to any item that has its `triggered` attribute set to `True`. Such items will only be checked (or in the case of actions: run) if the triggering item is fixed (or a triggering action completes successfully).

```
files = {
    '/etc/daemon.conf': {
        [...]
        'triggers': [
            'action:restart_daemon',
        ],
    },
}

actions = {
    'restart_daemon': {
        'command': "service daemon restart",
        'triggered': True,
```

```
    },
}
```

The above example will run **service daemon restart** every time BundleWrap successfully applies a change to `/etc/daemon.conf`. If an action is triggered multiple times, it will only be run once.

unless

Another builtin item attribute is `unless`. For example, it can be used to construct a one-off file item where BundleWrap will only create the file once, but won't check or modify its contents once it exists.

```
files = {
    "/path/to/file": {
        [...]
        "unless": "test -x /path/to/file",
    },
}
```

This will run **test -x /path/to/file** before doing anything with the item. If the command returns 0, no action will be taken to "correct" the item.

Note: Another common use for `unless` is with actions that perform some sort of install operation. In this case, the `unless` condition makes sure the install operation is only performed when it is needed instead of every time you run **bw apply**. In scenarios like this you will probably want to set `cascade_skip` to `False` so that skipping the installation (because the thing is already installed) will not cause every item that depends on the installed thing to be skipped. Example:

```
actions = {
    'download_thing': {
        'command': "wget http://example.com/thing.bin -O /opt/thing.bin && chmod +x /opt/thing.bin",
        'unless': "test -x /opt/thing.bin",
        'cascade_skip': False,
    },
    'run_thing': {
        'command': "/opt/thing.bin",
        'needs': ["action:download_thing"],
    },
}
```

If `action:download_thing` would not set `cascade_skip` to `False`, `action:run_thing` would only be executed once: directly after the thing has been downloaded. On subsequent runs, `action:download_thing` will fail the `unless` condition and be skipped. This would also cause all items that depend on it to be skipped, including `action:run_thing`.

---

cascade_skip

There are some situations where you don't want to default behavior of skipping everything that depends on a skipped item. That's where `cascade_skip` comes in. Set it to `False` and skipping an item won't skip those that depend on it. Note that items can be skipped

- interactively or
- because they haven't been *triggered* or
- because one of their dependencies failed or
- they failed their *'unless' condition* or
- because an *action* had its `interactive` attribute set to `True` during a non-interactive run

The following example will offer to run an `apt-get update` before installing a package, but continue to install the package even if the update is declined interactively.

```
actions = {
    'apt_update': {
        'cascade_skip': False,
        'command': "apt-get update",
    },
}

pkg_apt = {
    'somepkg': {
        'needs': ["action:apt_update"],
    },
}
```

Canned actions

Some item types have what we call "canned actions". Those are pre-defined actions attached directly to an item. Take a look at this example:

```
svc_upstart = {'mysql': {'running': True}}

files = {
    "/etc/mysql/my.cnf": {
        'source': "my.cnf",
        'triggers': [
            "svc_upstart:mysql:reload",  # this triggers the canned action
        ],
    },
}
```

Canned actions always have to be triggered in order to run. In the example above, a change in the file `/etc/mysql/my.cnf` will trigger the `reload` action defined by the *svc_upstart item type* for the mysql service.

1.4.4 Hooks

Hooks enable you to execute custom code at certain points during a BundleWrap run. This is useful for integrating with other systems e.g. for team notifications, logging or statistics.

To use hooks, you need to create a subdirectory in your repo called `hooks`. In that directory you can place an arbitrary number of Python source files. If those source files define certain functions, these functions will be called at the appropriate time.

Example

`hooks/my_awesome_notification.py`:

```
from my_awesome_notification_system import post_message

def node_apply_start(repo, node, interactive=False, **kwargs):
    post_message("Starting apply on {}, everything is gonna be OK!".format(node.name))
```

Note: Always define your hooks with `**kwargs` so we can pass in more information in future updates without breaking your hook.

Functions

This is a list of all functions a hook file may implement.

action_run_start (*repo*, *node*, *action*, ***kwargs*)

Called each time a **bw apply** command reaches a new action.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **item** (*Item*) – The current action.

action_run_end(*repo*, *node*, *action*, *duration=None*, *status=None*, ***kwargs*)
Called each time a **bw apply** command completes processing an action.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **item** (*Item*) – The current action.
- **duration** (*timedelta*) – How long the action was running.
- **status** (*ItemStatus*) – An object with these attributes: `correct`, `skipped`.

apply_start(*repo*, *target*, *nodes*, *interactive=False*, ***kwargs*)
Called when you start a **bw apply** command.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **target** (*str*) – The group or node name you gave on the command line.
- **nodes** (*list*) – A list of node objects affected (list of `bundlewrap.node.Node` instances).
- **interactive** (*bool*) – Indicates whether the apply is interactive or not.

apply_end(*repo*, *target*, *nodes*, *duration=None*, ***kwargs*)
Called when a **bw apply** command completes.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **target** (*str*) – The group or node name you gave on the command line.
- **nodes** (*list*) – A list of node objects affected (list of `bundlewrap.node.Node` instances).
- **duration** (*timedelta*) – How long the apply took.

item_apply_start(*repo*, *node*, *item*, ***kwargs*)
Called each time a **bw apply** command reaches a new item.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **item** (*Item*) – The current item.

item_apply_end(*repo*, *node*, *item*, *duration=None*, *status_before=None*, *status_after=None*, ***kwargs*)
Called each time a **bw apply** command completes processing an item.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **item** (*Item*) – The current item.
- **duration** (*timedelta*) – How long the apply took.
- **status_before** (*ItemStatus*) – An object with these attributes: `correct`, `info`, `skipped`.
- **status_after** (*ItemStatus*) – See `status_before`.

node_apply_start(*repo*, *node*, *interactive=False*, ***kwargs*)
Called each time a **bw apply** command reaches a new node.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **interactive** (*bool*) – `True` if this is an interactive apply run.

node_apply_end(*repo*, *node*, *duration=None*, *interactive=False*, *result=None*, ***kwargs*)
Called each time a **bw apply** command finishes processing a node.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **duration** (*timedelta*) – How long the apply took.
- **interactive** (*bool*) – `True` if this was an interactive apply run.
- **result** (*ApplyResult*) – An object with these attributes: `correct`, `failed`, `fixed`, `skipped`.

node_run_start(*repo*, *node*, *command*, ***kwargs*)

Called each time a **bw run** command reaches a new node.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **command** (*str*) – The command that will be run on the node.

node_run_start(*repo*, *node*, *command*, *duration=None*, *return_code=None*, *stdout=""*, *stderr=""*, ***kwargs*)

Called each time a **bw run** command finishes on a node.

Parameters

- **repo** (*Repository*) – The current repository (instance of `bundlewrap.repo.Repository`).
- **node** (*Node*) – The current node (instance of `bundlewrap.node.Node`).
- **command** (*str*) – The command that was run on the node.
- **duration** (*timedelta*) – How long it took to run the command.
- **return_code** (*int*) – Return code of the remote command.
- **stdout** (*str*) – The captured stdout stream of the remote command.
- **stderr** (*str*) – The captured stderr stream of the remote command.

run_start(*repo*, *target*, *nodes*, *command*, ***kwargs*)

Called each time a **bw run** command starts.

Parameters

- **repo** (*Repository*) – The current repository (instance of bundlewrap.repo.Repository).
- **target** (*str*) – The group or node name you gave on the command line.
- **nodes** (*list*) – A list of node objects affected (list of bundlewrap.node.Node instances).
- **command** (*str*) – The command that will be run on the node.

run_start (*repo*, *target*, *nodes*, *command*, *duration=None*, ***kwargs*)

Called each time a **bw run** command finishes.

Parameters

- **repo** (*Repository*) – The current repository (instance of bundlewrap.repo.Repository).
- **target** (*str*) – The group or node name you gave on the command line.
- **nodes** (*list*) – A list of node objects affected (list of bundlewrap.node.Node instances).
- **command** (*str*) – The command that was run.
- **duration** (*timedelta*) – How long it took to run the command on all nodes.

1.4.5 Custom item types

Step 1: Create an item module

Create a new file called /your/bundlewrap/repo/items/foo.py. You can use this as a template:

```
from bundlewrap.items import Item, ItemStatus


class Foo(Item):
    """
    A foo.
    """
    BUNDLE_ATTRIBUTE_NAME = "foo"
    DEPENDS_STATIC = []
    ITEM_ATTRIBUTES = {
        'attribute': "default value",
    }
    ITEM_TYPE_NAME = "foo"
    PARALLEL_APPLY = True
    REQUIRED_ATTRIBUTES = ['attribute']

    def __repr__(self):
        return "<Foo attribute:{}>".format(self.attributes['attribute'])
```

```
    def ask(self, status):
        """
        Returns a string asking the user if this item should be
        implemented.
        """
        return ""

    def fix(self, status):
        """
        Do whatever is necessary to correct this item.
        """
        raise NotImplementedError

    def get_status(self):
        """
        Returns an ItemStatus instance describing the current status of
        the item on the actual node. Must not be cached.
        """
        return ItemStatus(
            correct=True,
            description="No description available.",
            info={},
        )
```

Step 2: Define attributes

`BUNDLE_ATTRIBUTE_NAME` is the name of the variable defined in a bundle module that holds the items of this type. If your bundle looks like this:

```
foo = { [...] }
```

...then you should put `BUNDLE_ATTRIBUTE_NAME = "foo"` here.

`DEPENDS_STATIC` is a list of hard-wired dependencies for all intances of your item. For example, all services inherently depend on all packages (because you can't start the service without installing its package first). Most of the time, this will be a wildcard dependency on a whole type of items, not a specific one:

```
DEPENDS_STATIC = ["file:/etc/hosts", "user:"]  # depends on /etc/hosts and all users
```

`ITEM_ATTRIBUTES` is a dictionary of the attributes users will be able to configure for your item. For files, that would be stuff like owner, group, and permissions. Every attribute (even if it's mandatory) needs a default value, `None` is totally acceptable:

```
ITEM_ATTRIBUTES = {'attr1': "default1"}
```

`ITEM_TYPE_NAME` sets the first part of an items ID. For the file items, this is "file". Therefore, file ID look this this: `file:/path`. The second part is the name a user assigns to your item in a bundle. Example:

```
ITEM_TYPE_NAME = "foo"
```

`PARALLEL_APPLY` indicates whether multiple instances of this item can be applied in parallel. For most items this is OK (e.g. creating multiple files at the same time), but some types of items have to be applied sequentially (e.g. package managers usually employ locks to ensure only one package is installed at a time):

```
PARALLEL_APPLY = False
```

`REQUIRED_ATTRIBUTES` is a list of attribute names that must be set on each item of this type. If BundleWrap encounters an item without all these attributes during bundle inspection, an exception will be raised. Example:

```
REQUIRED_ATTRIBUTES = ['attr1', 'attr2']
```

Step 3: Implement methods

1.4.6 Custom code

The `libs/` subdirectory of your repository provides a convenient place to put reusable code used throughout your bundles and hooks.

A Python module called `example.py` placed in this directory will be available as `repo.libs.example` wherever you have access to a `bundlewrap.repo.Repository` object. In `nodes.py` and `groups.py`, you can do the same thing with just `libs.example`.

> **Warning:** Only single files, no subdirectories or packages, are supported at the moment.

1.4.7 Plugins

The plugin system in BundleWrap is an easy way of integrating third-party code into your repository.

> **Warning:** While plugins are subject to some superficial code review by BundleWrap developers before being accepted, we cannot make any guarantees as to the quality and trustworthiness of plugins. Always do your due diligence before running third-party code.

Finding plugins

It's as easy as **bw repo plugin search <term>**. Or you can browse plugins.bundlewrap.org.

Installing plugins

You probably guessed it: **bw repo plugin install <plugin>**

Installing the first plugin in your repo will create a file called `plugins.json`. You should commit this file (and any files installed by the plugin of course) to version control.

> **Hint:** Avoid editing files provided by plugins at all costs. Local modifications will prevent future updates to the

plugin.

Updating plugins

You can update all installed plugins with this command: **bw repo plugin update**

Removing a plugin

bw repo plugin remove <plugin>

Writing your own

See also:

The guide on publishing your own plugins

1.5 Command Line Interface

The **bw** utility is BundleWrap's command line interface.

Note: This page is not meant as a complete reference. It provides a starting point to explore the various subcommands. If you're looking for details, `--help` is your friend.

1.5.1 `bw apply`

```
$ bw apply -i mynode
```

The most important and most used part of BundleWrap, **bw apply** will apply your configuration to a set of *nodes*. By default, it operates in a non-interactive mode. When you're trying something new or are otherwise unsure of some changes, use the `-i` switch to have BundleWrap interactively ask before each change is made.

1.5.2 bw run

```
$ bw run mygroup "uname -a"
```

Unsurprisingly, the `run` subcommand is used to run commands on nodes.

As with most commands that accept node names, you can also give a *group* name or any combination of node and group names, separated by commas (without spaces, e.g. `node1,group2,node3`). A third option is to use a bundle selector like `bundle:my_bundle`. It will select all nodes with the named *bundle*. You can freely mix and match node names, group names, and bundle selectors.

Negation is also possible for bundles and groups. `!bundle:foo` will add all nodes without the foo bundle, while `!group:foo` will add all nodes that aren't in the foo group.

1.5.3 bw nodes and bw groups

```
$ bw nodes --hostnames | xargs -n 1 ping -c 1
```

With these commands you can quickly get a list of all nodes and groups in your *repository*. The example above uses *`--hostnames`* to get a list of all DNS names for your nodes and send a ping to each one.

1.5.4 bw debug

```
$ bw debug
bundlewrap X.Y.Z interactive repository inspector
> You can access the current repository as 'repo'.
>>> len(repo.nodes)
121
```

This command will drop you into a Python shell with direct access to BundleWrap's *API*. Once you're familiar with it, it can be a very powerful tool.

1.5.5 bw plot

Hint: You'll need Graphviz installed on your machine for this to be useful.

```
$ bw plot node mynode | dot -Tsvg -omynode.svg
```

You won't be using this every day, but it's pretty cool. The above command will create an SVG file (you can open these in your browser) that shows the item dependency graph for the given node. You will see bundles as dashed rectangles, static dependencies (defined in BundleWrap itself) in green, auto-generated dependencies (calculated dynamically each time you run **bw apply**) in blue and dependencies you defined yourself in red.

It offers an interesting view into the internal complexities BundleWrap has to deal with when figuring out the order in which your items can be applied to your node.

1.5.6 bw test

```
$ bw test
 node1:pkg_apt:samba
 node1:file:/etc/samba/smb.conf

[...]

+----- traceback from worker ------
|
|  Traceback (most recent call last):
|    File "/Users/trehn/Projects/software/bundlewrap/src/bundlewrap/concurrency.py", line 78, in _wo
|      return_value = target(*msg['args'], **msg['kwargs'])
|    File "<string>", line 378, in test
|  BundleError: file:/etc/samba/smb.conf from bundle 'samba' refers to missing file '/path/to/bundle
|
+---------------------------------
```

This command is meant to be run automatically like a test suite after every commit. It will try to catch any errors in your bundles and file templates by initializing every item for every node (but without touching the network).

1.6 API

While most users will interact with BundleWrap through the **bw** command line utility, you can also use it from your own code to extract data or further automate config management tasks.

Even within BundleWrap itself (e.g. templates, libs, and hooks) you are often given repo and/or node objects to work with. Their methods and attributes are documented below.

Some general notes on using BundleWrap's API:

- There can be an arbitrary amount of `bundlewrap.repo.Repository` objects per process.
- Repositories are read as needed and not re-read when something changes. Modifying files in a repo during the lifetime of the matching Repository object may result in undefined behavior.

1.6.1 Example

Here's a short example of how to use BundleWrap to get the uptime for a node.

```
from bundlewrap.repo import Repository

repo = Repository("/path/to/my/repo")
node = repo.get_node("mynode")
uptime = node.run("uptime")
print(uptime.stdout)
```

1.6.2 Reference

class bundlewrap.repo.**Repository**(*path*)

The starting point of any interaction with BundleWrap. An object of this class represents the repository at the given path.

groups

A list of all groups in the repo (instances of bundlewrap.group.Group)

group_names

A list of all group names in this repo.

nodes

A list of all nodes in the repo (instances of bundlewrap.node.Node)

node_names

A list of all node names in this repo

revision

The current git, hg or bzr revision of this repo. None if no SCM was detected.

get_group(*group_name*)

Get the group object with the given name.

Parameters **group_name** (*str*) – Name of the desired group

Returns The group object for the given name

Return type bundlewrap.group.Group

get_node(*node_name*)

Get the node object with the given name.

Parameters **node_name** (*str*) – Name of the desired node

Returns The node object for the given name

Return type bundlewrap.node.Node

nodes_in_all_groups(*group_names*)
Returns a list of nodes where every node is a member of every group given.

Parameters **group_names** (*list*) – Names of groups to search for common nodes.

Returns All nodes common to all given groups.

Return type list

nodes_in_any_group(*group_names*)
Returns all nodes that are a member of at least one of the given groups.

Parameters **group_names** (*str*) – Names of groups to search for nodes.

Returns All nodes present in at least one group.

Return type generator

class bundlewrap.node.**Node**
A system managed by BundleWrap.

bundles
A list of all bundles associated with this node (instances of bundlewrap.bundle.Bundle)

groups
A list of bundlewrap.group.Group objects this node belongs to

hostname
The DNS name BundleWrap uses to connect to this node

items
A list of items on this node (instances of subclasses of bundlewrap.items.Item)

metadata
A dictionary of custom metadata, merged from information in *nodes.py* and *groups.py*

name
The internal identifier for this node

download(*remote_path*, *local_path*)
Downloads a file from the node.

Parameters

- **remote_path** (*str*) – Which file to get from the node
- **local_path** (*str*) – Where to put the file

get_item(*item_id*)
Get the item object with the given ID (e.g. "file:/etc/motd").

Parameters **item_id** (*str*) – ID of the desired item

Returns The item object for the given ID

Return type instances of subclasses of `bundlewrap.items.Item`

has_bundle(*bundle_name*)
`True` if the node has a bundle with the given name.

Parameters **bundle_name** (*str*) – Name of the bundle to look for

Return type bool

has_any_bundle(*bundle_names*)
`True` if the node has a bundle with any of the given names.

Parameters **bundle_names** (*list*) – List of bundle names to look for

Return type bool

in_group(*group_name*)
`True` if the node is in a group with the given name.

Parameters **group_name** (*str*) – Name of the group to look for

Return type bool

in_any_group(*group_names*)
`True` if the node is in a group with any of the given names.

Parameters **group_names** (*list*) – List of group names to look for

Return type bool

run(*command*, *may_fail=False*)
Runs a command on the node.

Parameters

- **command** (*str*) – What should be executed on the node
- **may_fail** (*bool*) – If `False`, `bundlewrap.exceptions.RemoteException` will be raised if the command does not return 0.

Returns An object that holds the return code as well as captured stdout and stderr

Return type `bundlewrap.operations.RunResult`

upload(*local_path*, *remote_path*, *mode=None*, *owner=""*, *group=""*)
Uploads a file to the node.

Parameters

- **local_path** (*str*) – Which file to upload
- **remote_path** (*str*) – Where to put the file on the target node
- **mode** (*str*) – File mode, e.g. "0644"
- **owner** (*str*) – Username of the file owner
- **group** (*str*) – Group name of the file group

class `bundlewrap.group.`**Group**
A user-defined group of nodes.

name
The name of this group

nodes
A list of all nodes in this group (instances of `bundlewrap.node.Node`, includes subgroup members)

1.7 Contributing

We welcome all input and contributions to BundleWrap. If you've never done this sort of thing before, maybe check out contribution-guide.org. But don't be afraid to make mistakes, nobody expects your first contribution to be perfect. We'll gladly help you out.

1.7.1 Submitting bug reports

Please use the GitHub issue tracker and take a few minutes to look for existing reports of the same problem (open or closed!).

Hint: If you've found a security issue or are not at all sure, just contact trehn@bundlewrap.org.

1.7.2 Contributing code

Note: Before working on new features, try reaching out to one of the core authors first. We are very concerned with keeping BundleWrap lean and not introducing bloat. If your idea is not a good fit for all or most BundleWrap users, it can still be included *as a plugin*.

Here are the steps:

1. Write your code. Awesome!
2. If you haven't already done so, please consider writing tests. Otherwise, someone else will have to do it for you.
3. Same goes for documentation.
4. Review and sign the CAA (Copyright Assignment Agreement) by adding your name and email to the `AUTHORS` file. (This step can be skipped if your contribution is too small to be considered intellectual property, e.g. spelling fixes)
5. Open a pull request on GitHub.
6. Feel great. Thank you.

1.7.3 Help

If at any point you need help or are not sure what to do, just drop by in #bundlewrap on Freenode or poke @bundlewrap on Twitter.

Writing your own plugins

Plugins can provide almost any file in a BundleWrap repository: bundles, custom items, hooks, libs, etc.

Notable exceptions are `nodes.py` and `groups.py`. If your plugin wants to extend those, use a *lib* instead and ask users to add the result of a function call in your lib to their nodes or groups dicts.

Warning: If your plugin depends on other libraries, make sure that it catches ImportErrors in a way that makes it obvious for the user what's missing. Keep in mind that people will often just **git pull** their repo and not install your plugin themselves.

Starting a new plugin

Step 1: Clone the plugins repo

Create a clone of the official plugins repo on GitHub.

Step 2: Create a branch

You should work on a branch specific to your plugin.

Step 3: Copy your plugin files

Now take the files that make up your plugin and move them into a subfolder of the plugins repo. The subfolder must be named like your plugin.

Step 4: Create required files

In your plugin subfolder, create a file called `manifest.json` from this template:

```
{
        "desc": "Concise description (keep it somewhere around 80 characters)",
        "help": "Optional verbose help text to be displayed after installing. May\ninclude\nnewlines.
        "provides": [
                "bundles/example/bundle.py",
                "hooks/example.py"
        ],
        "version": 1
}
```

The `provides` section must contain a list of all files provided by your plugin.

You also have to create an `AUTHORS` file containing your name and email address.

Last but not least we require a `LICENSE` file with an OSI-approved Free Software license.

Step 5: Update the plugin index

Run the `update_index.py` script at the root of the plugins repo.

Step 6: Run tests

Run the `test.py` script at the root of the plugins repo. It will tell you if there is anything wrong with your plugin.

Step 7: Commit

Commit all changes to your branch

Step 8: Create pull request

Create a pull request on GitHub to request inclusion of your new plugin in the official repo. Only then will your plugin become available to be installed by **bw repo plugin install yourplugin**.

Updating an existing plugin

To release a new version of your plugin:

- Increase the version number in `manifest.json`
- Update the list of provided files in `manifest.json`
- If you're updating someone elses plugin, you should get their consent and add your name to `AUTHORS`

Then just follow the instructions above from step 5 onward.

1.8 FAQ

1.8.1 Technical

BundleWrap says an item failed to apply, what do I do now?

Try running **bw apply -i nodename** to see which attribute of the item could not be fixed. If that doesn't tell you enough, try `bw --debug apply -i nodename` and look for the command BundleWrap is using to fix the *item* in question. Then try running that command yourself and check for any errors.

What happens when two people start applying configuration to the same node?

BundleWrap uses a locking mechanism to prevent collisions like this. When BundleWrap finds a lock on a *node* in interactive mode, it will display information about who acquired the lock (and when) and will ask whether to ignore the lock or abort the process. In noninteractive mode, the operation is always cancelled for the node in question unless `--force` is used.

How can I have BundleWrap reload my services after config changes?

See *canned actions* and *triggers*.

Will BundleWrap keep track of package updates?

No. BundleWrap will only care about whether a package is installed or not. Updates will have to be installed through a separate mechanism (I like to create an *action* with the `interactive` attribute set to `True`). Selecting specific versions should be done through your package manager.

Is there a probing mechanism like Ohai?

No. BundleWrap is meant to be very push-focused. The node should not have any say in what configuration it will receive. If you disagree with this ideology and really need data from the node beforehand, you can use a *hook* to gather the data and populate `node.metadata`.

Is there a way to remove any unmanaged files/directories in a directory?

Not at the moment. We're tracking this topic in issue #56.

Is there any integration with my favorite Cloud?

Not right now. A separate project (called "cloudwart") is in planning, but no code has been written and it's not a priority at the moment.

Is BundleWrap secure?

BundleWrap is more concerned with safety than security. Due to its design, it is possible for your coworkers to introduce malicious code into a BundleWrap repository that could compromise your machine. You should only use trusted repositories and plugins. We also recommend following commit logs to your repos.

1.8.2 The BundleWrap Project

Why do contributors have to sign a Copyright Assignment Agreement?

While it sounds scary, Copyright assignment is used to improve the enforceability of the GPL. Even the FSF does it, read their explanation why. The agreement used by BundleWrap is from harmonyagreements.org.

If you're still concerned, please do not hesitate to contact @trehn.

Isn't this all very similar to Ansible?

Some parts are, but there are significant differences as well. Check out the *Alternatives page* for a writeup of the details.

1.9 Glossary

action Actions are a special kind of *item* used for running shell commands during each **bw apply**. They allow you to do things that aren't persistent in nature

apply An "apply" is what we call the process of what's otherwise known as "converging" the state described by your repository and the actual status quo on the *node*.

bundle A collection of *items*. Most of the time, you will create one bundle per application. For example, an Apache bundle will include the httpd service, the virtual host definitions and the apache2 package.

group Used for organizing your *nodes*.

hook *Hooks* can be used to run your own code automatically during various stages of BundleWrap operations.

item A single piece of configuration on a node, e.g. a file or an installed package.

You might be interested in *this overview of item types*.

lib *Libs* are a way to store Python modules in your repository and make them accessible to your bundles and templates.

node A managed system, no matter if physical or virtual.

repo A repository is a directory with *some stuff* in it that tells BundleWrap everything it needs to know about your infrastructure.

1.10 About

Development on BundleWrap started in July 2012, borrowing some ideas from Bcfg2. Some key features that are meant to set BundleWrap apart from other config management systems are:

- decentralized architecture
- pythonic and easily extendable
- easy to get started with
- true item-level parallelism (in addition to working on multiple nodes simultaneously, BundleWrap will continue to fix config files while installing a package on the same node)
- very customizable item dependencies
- collaboration features like node locking (to prevent simultaneous applies to the same node) and hooks for chat notifications

- built-in testing facility (**bw test**)
- can be used as a library

BundleWrap is a "pure" free software project licensed under the terms of the GPLv3, with no *Enterprise Edition* or commercial support.

1.11 Alternatives

This page is an effort to compare BundleWrap to other config management systems. It very hard to keep this information complete and up to date, so please feel free to raise issues or create pull requests if something is amiss.

BundleWrap has the following properties that are unique to it or at least not common among other solutions:

- server- and agent-less architecture
- item-level parallelism to speed up convergence of complex nodes
- interactive mode to review configuration as it it being applied
- *Mako file templates*
- verifies that each action taken actually fixed the item in question
- useful and actionable error messages
- built-in *visualization* of node configuration
- nice *Python API*
- designed to be mastered quickly and easily remembered
- for better or worse: no commercial agenda/support
- no support for non-Linux target nodes (BundleWrap itself can be run from Mac OS as well)

1.11.1 Ansible

Ansible is very similar to BundleWrap in how it communicates with nodes. Both systems do not use server or agent processes, but SSH. Ansible can optionally use OpenSSH instead of a Python SSH implementation to speed up performance. On the other hand, BundleWrap will always use the Python implementation, but with multiple connections to each node. This should give BundleWrap a performance advantage on very complex systems with many items, since each connection can work on a different item simultaneously.

To apply configuration, Ansible uploads pieces of code called modules to each node and runs them there. Many Ansible modules depend on the node having a Python 2.x interpreter installed. BundleWrap runs commands on the target node just as you would in an interactive SSH session. Most of the *commands needed* by BundleWrap are provided by coreutils and should be present on all standard Linux systems.

Ansible ships with loads of modules while BundleWrap will only give you the most needed primitives to work with. For example, we will not add an item type for remote downloads because you can easily build that yourself using an *action* with **wget**.

Ansible's playbooks roughly correspond to BundleWrap's bundles, but are written in YAML using a special playbook language. BundleWrap uses Python for this purpose, so if you know some basic Python you only need to learn the schema of the dictionaries you're building.

File templates in Ansible are Jinja2, while BundleWrap uses Mako by default and offers Jinja2 as an option.

Ansible, Inc. offers paid support for Ansible and an optional web-based addon called Ansible Tower.

1.11.2 BCFG2

BCFG2's bundles obviously were an inspiration for BundleWrap. One important difference is that BundleWrap's bundles are usually completely isolated and self-contained within their directory while BCFG2 bundles may need resources (e.g. file templates) from elsewhere in the repository.

On a practical level BundleWrap prefers pure Python and Mako over the XML- and text-variants of Genshi used for bundle and file templating in BCFG2.

And of course BCFG2 has a very traditional client/server model while BundleWrap runs only on the operators computer.

1.11.3 Chef

Chef has basically two modes of operation: The most widely used one involves a server component and the **chef-client** agent. The second option is **chef-solo**, which will apply configuration from a local repository to the node the repository is located on. BundleWrap supports neither of these modes and always applies configuration over SSH.

Overall, Chef is harder to get into, but will scale to thousands of nodes.

The community around Chef is quite large and probably the largest of all config management systems. This means lots of community-maintained cookbooks to choose from. BundleWrap does have a *plugin system* to provide almost anything in a repository, but there aren't many plugins to choose from yet.

Chef is written in Ruby and uses the popular ERB template language. BundleWrap is heavily invested in Python and offers support for Mako and Jinja2 templates.

OpsCode offers paid support for Chef and SaaS hosting for the server component. AWS OpsWorks also integrates Chef cookbooks.

b