Charm Helpers Documentation

Release 0.3.2

Charm Helpers Developers

Mar 02, 2017

Contents

The `charmhelpers` Python library is an extensive collection of functions and classes for simplifying the development of Juju Charms. It includes utilities for:

- Interacting with the host environment
- Managing hook events
- Reading and writing charm configuration
- Installing dependencies
- Much, much more!

CHAPTER 1

Getting Started

For a video introduction to `charmhelpers`, check out this Charm School session. To start using `charmhelpers`, proceed with the instructions on the remainder of this page.

Installing Charm Tools

First, follow these instructions to install the `charm-tools` package for your platform.

Creating a New Charm

```
$ cd ~
$ mkdirs -p charms/precise
$ cd charms/precise
$ charm create -t python mycharm
INFO: Generating template for mycharm in ./mycharm
INFO: No mycharm in apt cache; creating an empty charm instead.
Symlink all hooks to one python source file? [yN] y
INFO:root:Loading charm helper config from charm-helpers.yaml.
INFO:root:Checking out lp:charm-helpers to /tmp/tmpPAqUyN/charm-helpers.
Branched 160 revisions.
INFO:root:Syncing directory: /tmp/tmpPAqUyN/charm-helpers/charmhelpers/core -> lib/
↪charmhelpers/core.
INFO:root:Adding missing __init__.py: lib/charmhelpers/__init__.py
```

Let's see what our new charm looks like:

```
$ tree mycharm/
mycharm/
- charm-helpers.yaml
- config.yaml
- hooks
```

```
|   - config-changed -> hooks.py
|   - hooks.py
|   - install -> hooks.py
|   - start -> hooks.py
|   - stop -> hooks.py
|   - upgrade-charm -> hooks.py
- icon.svg
- lib
|   - charmhelpers
|       - core
|       |   - fstab.py
|       |   - hookenv.py
|       |   - host.py
|       |   - __init__.py
|       - __init__.py
- metadata.yaml
- README.ex
- revision
- scripts
|   - charm_helpers_sync.py
- tests
    - 00-setup
    - 10-deploy

6 directories, 20 files
```

The `charmhelpers` code is bundled in our charm in the `lib/` directory. All of our python code will go in `hooks/hook.py`. A look at that file reveals that `charmhelpers` has been added to the python path and imported for us:

```
$ head mycharm/hooks/hooks.py -n11
#!/usr/bin/python

import os
import sys

sys.path.insert(0, os.path.join(os.environ['CHARM_DIR'], 'lib'))

from charmhelpers.core import (
    hookenv,
    host,
)
```

Updating Charmhelpers Packages

By default, a new charm installs only the `charmhelpers.core` package, but other packages are available (for a complete list, see the *API Documentation*). The installed packages are controlled by the `charm-helpers.yaml` file in our charm:

```
$ cd mycharm
$ cat charm-helpers.yaml
destination: lib/charmhelpers
branch: lp:charm-helpers
include:
  - core
```

Let's update this file to include some more packages:

```
$ vim charm-helpers.yaml
$ cat charm-helpers.yaml
destination: lib/charmhelpers
branch: lp:charm-helpers
include:
  - core
  - contrib.storage
  - fetch
```

Now we need to download the new packages into our charm:

```
$ ./scripts/charm_helpers_sync.py -c charm-helpers.yaml
INFO:root:Loading charm helper config from charm-helpers.yaml.
INFO:root:Checking out lp:charm-helpers to /tmp/tmpT38Y87/charm-helpers.
Branched 160 revisions.
INFO:root:Syncing directory: /tmp/tmpT38Y87/charm-helpers/charmhelpers/core -> lib/
↪charmhelpers/core.
INFO:root:Syncing directory: /tmp/tmpT38Y87/charm-helpers/charmhelpers/contrib/
↪storage -> lib/charmhelpers/contrib/storage.
INFO:root:Adding missing __init__.py: lib/charmhelpers/contrib/__init__.py
INFO:root:Syncing directory: /tmp/tmpT38Y87/charm-helpers/charmhelpers/fetch -> lib/
↪charmhelpers/fetch.
```

A look at our charmhelpers directory reveals that the new packages have indeed been added. We are now free to import and use them in our charm:

```
$ tree lib/charmhelpers/
lib/charmhelpers/
- contrib
|   - __init__.py
|   - storage
|       - __init__.py
|       - linux
|           - ceph.py
|           - __init__.py
|           - loopback.py
|           - lvm.py
|           - utils.py
- core
|   - fstab.py
|   - hookenv.py
|   - host.py
|   - __init__.py
- fetch
|   - archiveurl.py
|   - bzrurl.py
|   - __init__.py
- __init__.py

5 directories, 15 files
```

Next Steps

Now that you have access to `charmhelpers` in your charm, check out the *Examples* or *API Documentation* to learn about all the great functionality that `charmhelpers` provides.

CHAPTER 2

Examples

If you'd like to contribute an example (please do!), please refer to the *Contributing* page for instructions on how to do so.

Interacting with Charm Configuration

The *charmhelpers.core.hookenv.config()*, when called with no arguments, returns a *charmhelpers.core.hookenv.Config* instance - a dictionary representation of a charm's `config.yaml` file. This object can be used to:

- get a charm's current config values
- check if a config value has changed since the last hook invocation
- view the previous value of a changed config item
- save arbitrary key/value data for use in a later hook

For the following examples we'll assume our charm has a config.yaml file that looks like this:

```
options:
  app-name:
    type: string
    default: "My App"
    description: "Name of your app."
```

Getting charm config values

```
# hooks/hooks.py

from charmhelpers.core import hookenv

hooks = hookenv.Hooks()
```

```
@hooks.hook('install')
def install():
    config = hookenv.config()

    assert config['app-name'] == 'My App'
```

Checking if a config value has changed

Let's say the user changes the `app-name` config value at runtime by executing the following juju command:

```
juju set mycharm app-name="My New App"
```

which triggers a `config-changed` hook:

```
# hooks/hooks.py

from charmhelpers.core import hookenv

hooks = hookenv.Hooks()

@hooks.hook('config-changed')
def config_changed():
    config = hookenv.config()

    assert config.changed('app-name')
    assert config['app-name'] == 'My New App'
    assert config.previous('app-name') == 'My App'
```

Saving arbitrary key/value data

The *Config* object maybe also be used to store arbitrary data that you want to persist across hook invocations:

```
# hooks/hooks.py

from charmhelpers.core import hookenv

hooks = hookenv.Hooks()

@hooks.hook('install')
def install():
    config = hookenv.config()

    config['mykey'] = 'myval'

@hooks.hook('config-changed')
def config_changed():
    config = hookenv.config()

    assert config['mykey'] == 'myval'
```

Managing Charms with the Services Framework [DEPRECATED]

Traditional charm authoring is focused on implementing hooks. That is, the charm author is thinking in terms of "What hook am I handling; what does this hook need to do?" However, in most cases, the real question should be "Do I have the information I need to configure and start this piece of software and, if so, what are the steps for doing so?" The services framework tries to bring the focus to the data and the setup tasks, in the most declarative way possible.

Hooks as Data Sources for Service Definitions

While the `install`, `start`, and `stop` hooks clearly represent state transitions, all of the other hooks are really notifications of changes in data from external sources, such as config data values in the case of `config-changed` or relation data for any of the `*-relation-*` hooks. Moreover, many charms that rely on external data from config options or relations find themselves needing some piece of external data before they can even configure and start anything, and so the `start` hook loses its semantic usefulness.

If data is required from multiple sources, it even becomes impossible to know which hook will be executing when all required data is available. (E.g., which relation will be the last to execute; will the required config option be set before or after all of the relations are available?) One common solution to this problem is to create "flag files" to track whether a given bit of data has been observed, but this can get cluttered quickly and is difficult to understand what conditions lead to which actions.

When using the services framework, all hooks other than `install` are handled by a single call to *manager.manage()*. This can be done with symlinks, or by having a `definitions.py` file containing the service definitions, and every hook can be reduced to:

```
#!/bin/env python
from charmhelpers.core.services import ServiceManager
from definitions import service_definitions
ServiceManager(service_definitions).manage()
```

So, what magic goes into `definitions.py`?

Service Definitions Overview

The format of service definitions are fully documented in *ServiceManager*, but most commonly will consist of one or more dictionaries containing four items: the name of a service being managed, the list of data contexts required before the service can be configured and started, the list of actions to take when the data requirements are satisfied, and list of ports to open. The service name generally maps to an Upstart job, the required data contexts are `dict` or `dict`-like structures that contain the data once available (usually subclasses of *RelationContext* or wrappers around *charmhelpers.core.hookenv.config()*), and the actions are just callbacks that are passed the service name for which they are executing (or a subclass of *ManagerCallback* for more complex cases).

An example service definition might be:

```
service_definitions = [
    {
        'service': 'wordpress',
        'ports': [80],
        'required_data': [config(), MySQLRelation()],
        'data_ready': [
            actions.install_frontend,
            services.render_template(source='wp-config.php.j2',
                                     target=os.path.join(WP_INSTALL_DIR, 'wp-config.
↪php'))
            services.render_template(source='wordpress.upstart.j2',
```

```
                                    target='/etc/init/wordpress'),
        ],
    },
]
```

Each time a hook is fired, the conditions will be checked (in this case, just that MySQL is available) and, if met, the appropriate actions taken (correct front-end installed, config files written / updated, and the Upstart job (re)started, implicitly).

Required Data Contexts

Required data contexts are, at the most basic level, are just dictionaries, and if they evaluate as True (e.g., if the contain data), their condition is considered to be met. A simple sentinal could just be a function that returns data if available or an empty `dict` otherwise.

For the common case of gathering data from relations, the *`RelationContext`* base class gathers data from a named relation and checks for a set of required keys to be present and set on the relation before considering that relation complete. For example, a basic MySQL context might be:

```
class MySQLRelation(RelationContext):
    name = 'db'
    interface = 'mysql'
    required_keys = ['host', 'user', 'password', 'database']
```

Because there could potentially be multiple units on a given relation, and to prevent conflicts when the data contexts are merged to be sent to templates (see below), the data for a `RelationContext` is nested in the following way:

```
relation[relation.name][unit_number][relation_key]
```

For example, to get the host of the first MySQL unit (`mysql/0`):

```
mysql = MySQLRelation()
unit_0_host = mysql[mysql.name][0]['host']
```

Note that only units that have set values for all of the required keys are included in the list, and if no units have set all of the required keys, instantiating the `RelationContext` will result in an empty list.

Data-Ready Actions

When a hook is triggered and all of the `required_data` contexts are complete, the list of "data ready" actions are executed. These callbacks are passed the service name from the `service` key of the service definition for which they are running, and are responsible for (re)configuring the service according to the required data.

The most common action should be to render a config file from a template. The *`render_template`* helper will merge all of the `required_data` contexts and render a Jinja2 template with the combined data. For example, to render a list of DSNs for units on the db relation, the template should include:

```
databases: [
  {% for unit in db %}
    "mysql://{{unit['user']}}:{{unit['password']}}@{{unit['host']}}/{{unit['database
↪']}}",
  {% endfor %}
]
```

Note that the actions need to be idempotent, since they will all be re-run if something about the charm changes (that is, if a hook is triggered). That is why rendering a template is preferred to editing a file via regular expression substitutions.

Also note that the actions are not responsible for starting the service; there are separate `start` and `stop` options that default to starting and stopping an Upstart service with the name given by the `service` value.

Conclusion

By using this framework, it is easy to see what the preconditions for the charm are, and there is never a concern about things being in a partially configured state. As a charm author, you can focus on what is important to you: what data is mandatory, what is optional, and what actions should be taken once the requirements are met.

CHAPTER 3

Contributing

All contributions, both code and documentation, are welcome!

Source

The source code is located at https://code.launchpad.net/charm-helpers. To submit contributions you'll need to create a Launchpad account if you do not already have one.

To get the code:

```
$ bzr branch lp:charm-helpers
```

To build and run tests:

```
$ cd charm-helpers
$ make
```

Submitting a Merge Proposal

Run `make test` and ensure all tests pass. Then commit your changes and push them to a personal branch:

```
bzr ci -m "Description of your changes"
bzr push lp:~<launchpad-username>/charm-helpers/my-feature
```

Note that the branch name ('my-feature' in the above example) can be anything you choose - preferably something descriptive.

Once your branch is pushed, open it in a web browser, e.g.:

```
https://code.launchpad.net/~<launchpad-username>/charm-helpers/my-feature
```

Find and click on the 'Propose for merging' link, and on the following screen, click the 'Propose Merge' button.

Note: Do not set a value in the 'Reviewer' field - it will be set automatically.

Open Bugs

If you're looking for something to work on, the open bug/feature list can be found at https://bugs.launchpad.net/charm-helpers.

Documentation

If you'd like to contribute to the documentation, please refer to the `HACKING` document in the root of the source tree for instructions on building the documentation.

Contributions to the *Examples* section of the documentation are especially welcome, and are easy to add. Simply add a new `.rst` file under `charmhelpers/docs/examples`.

Getting Help

If you need help you can find it in `#juju` on the Freenode IRC network. Come talk to us - we're a friendly bunch!

CHAPTER 4

API Documentation

charmhelpers.core package

charmhelpers.core.reactive

This module serves as the basis for creating charms and relation implementations using the reactive pattern.

Overview

The pattern is "reactive" because you use *@when* and similar decorators to indicate that blocks of code react to certain conditions, such as a relation reaching a specific *state*, a file changing, certain config values being set, etc. More importantly, you can react to not just indvidual conditions, but meaningful combinations of conditions that can span multiple hook invocations, in a natural way.

For example, the following would update a config file when both a database and admin password were available, and, if and only if that file was changed, the appropriate service would be restarted:

```
@when('db.database.available', 'admin-pass')
def render_config(pgsql):
    render_template('app-config.j2', '/etc/app.conf', {
        'db_conn': pgsql.connection_string(),
        'admin_pass': hookenv.config('admin-pass'),
    })

@when_file_changed('/etc/app.conf')
def restart_service():
    hookenv.service_restart('myapp')

if __name__ == '__main__':
    reactive.main()
```

Structure of a Reactive Charm

The structure of a reactive charm is similar to existing charms, with the addition of `reactive` and `relations` directories under `hooks`:

```
.
- metadata.yaml
- hooks
    - pgsql-relation-changed
    - reactive
    |   - common.py
    - relations
        - pgsql
            - common
            |   - __init__.py
            - interface.yaml
            - peer.py
            - provides.py
            - requires.py
```

The hooks will need to call *reactive.main()*, and the decorated handler blocks can be placed in any file under the `reactive` directory. The `relations` directory can contain any relation stub implementations that your charm uses.

If using Charm Composition, as is recommended, the relation hooks and `relations` directories will be automatically managed for you based on your `metadata.yaml`, so you can focus on writing just the `install` and `config-changed` hooks, and the `reactive` handler files.

Relation Stubs

A big part of the reactive pattern is the use of relation stubs. These are classes, based on *RelationBase*, that are reponsible for managing the conversation with remote services or units and informing the charm when the conversation has reached key points, called states, upon which the charm can act and do useful work. They allow a single interface author to create code to handle both sides of the conversation, and to expose a well-defined API to charm authors.

Relation stubs allows charm authors to focus on implementing the behavior and resources that the relation provides, while the interface author focuses on the communication necessary to get that behavior and resources between the related services. In general, the author of the charm that provides a particular interface is likely to be the interface author that creates both the provides and requires sides of the relation. After that, charm authors that wish to make use of that interface can just re-use the existing relation stub.

Non-Python Reactive Handlers

Reactive handlers can be written in any language, provided they conform to the `ExternalHandler` protocol. In short, they must accept a `--test` and `--invoke` argument and do the appropriate thing when called with each.

There are helpers for writing handlers in bash. For example:

```
source `which reactive.sh`

@when 'db.database.available' 'admin-pass'
function render_config() {
    db_conn=$(state_relation_call 'db.database.available' connection_string)
    admin_pass=$(config-get 'admin-pass')
    chlp render_template 'app-config.j2' '/etc/app.conf' --db_conn="$db_conn" --admin_
↪pass="$admin_pass"
```

```
}

@when_not 'db.database.available'
function no_db() {
    status-set waiting 'Waiting on database'
}

@when_not 'admin-pass'
function no_db() {
    status-set blocked 'Missing admin password'
}

@when_file_changed '/etc/app.conf'
function restart_service() {
    service myapp restart
}

reactive_handler_main
```

Reactive API Documentation

charmhelpers.core.reactive.decorators

hook	Register the decorated function to run when the current hook matches any of the `hook_patterns`.
not_unless	Assert that the decorated function can only be called if the desired_states are active.
only_once	Ensure that the decorated function is only executed the first time it is called.
when	Register the decorated function to run when all `desired_states` are active.
when_file_changed	Register the decorated function to run when one or more files have changed.
when_not	Register the decorated function to run when **not** all desired_states are active.

charmhelpers.core.reactive.decorators.**hook**(**hook_patterns*)

Register the decorated function to run when the current hook matches any of the `hook_patterns`.

The hook patterns can use the `{interface:...}` and `{A,B,...}` syntax supported by *any_hook()*.

If the hook is a relation hook, an instance of that relation class will be passed in to the decorated function.

For example, to match any joined or changed hook for the relation providing the `mysql` interface:

```
class MySQLRelation(RelationBase):
    @hook('{provides:mysql}-relation-{joined,changed}')
    def joined_or_changed(self):
        pass
```

charmhelpers.core.reactive.decorators.**not_unless**(**desired_states*)

Assert that the decorated function can only be called if the desired_states are active.

Note that, unlike *when()*, this does **not** trigger the decorated function if the states match. It **only** raises an

exception if the function is called when the states do not match.

This is primarily for informational purposes and as a guard clause.

charmhelpers.core.reactive.decorators.**only_once**(*action*)
Ensure that the decorated function is only executed the first time it is called.

This can be used on reactive handlers to ensure that they are only triggered once, even if their conditions continue to match on subsequent calls, even across hook invocations.

charmhelpers.core.reactive.decorators.**when**(**desired_states*)
Register the decorated function to run when all `desired_states` are active.

This decorator will pass zero or more relation instances to the handler, if any of the states are associated with relations. If so, they will be passed in in the same order that the states are given to the decorator.

Note that handlers whose conditions match are triggered at least once per hook invocation.

charmhelpers.core.reactive.decorators.**when_file_changed**(**filenames*, ***kwargs*)
Register the decorated function to run when one or more files have changed.

Parameters

- **filenames** (*list*) – The names of one or more files to check for changes.
- **hash_type** (*str*) – The type of hash to use for determining if a file has changed. Defaults to 'md5'. Must be given as a kwarg.

charmhelpers.core.reactive.decorators.**when_not**(**desired_states*)
Register the decorated function to run when **not** all desired_states are active.

This decorator will never cause arguments to be passed to the handler.

Note that handlers whose conditions match are triggered at least once per hook invocation.

charmhelpers.core.reactive.helpers

any_file_changed	Check if any of the given files have changed since the last time this was called.
data_changed	Check if the given set of data has changed since the previous call.
mark_invoked	Mark the given ID as having been invoked, for use with `was_invoked()`.
was_invoked	Returns whether the given ID has been invoked before, as per `mark_invoked()`.

charmhelpers.core.reactive.helpers.**any_file_changed**(*filenames*, *hash_type='md5'*)
Check if any of the given files have changed since the last time this was called.

Parameters

- **filenames** (*list*) – Names of files to check.
- **hash_type** (*str*) – Algorithm to use to check the files.

charmhelpers.core.reactive.helpers.**data_changed**(*data_id*, *data*, *hash_type='md5'*)
Check if the given set of data has changed since the previous call.

This works by hashing the JSON-serialization of the data. Note that, while the data will be serialized using `sort_keys=True`, some types of data structures, such as sets, may lead to false positivies.

Parameters

- **data_id** (*str*) – Unique identifier for this set of data.
- **data** – JSON-serializable data.
- **hash_type** (*str*) – Any hash algorithm supported by `hashlib`.

charmhelpers.core.reactive.helpers.**mark_invoked**(*invocation_id*)
Mark the given ID as having been invoked, for use with *was_invoked()*.

charmhelpers.core.reactive.helpers.**was_invoked**(*invocation_id*)
Returns whether the given ID has been invoked before, as per *mark_invoked()*.

This is useful for ensuring that a given block only runs one time:

```
def foo():
    if was_invoked('foo'):
        return
    do_something()
    mark_invoked('foo')
```

This is also available as a decorator at *only_once()*.

charmhelpers.core.reactive.relations

AutoAccessors	Metaclass that converts fields referenced by `auto_accessors` into accessor methods with very basic doc strings.
Conversation	Converations are the persistent, evolving, two-way communication between this service and one or more remote services.
RelationBase	The base class for all relation implementations.
relation_call	Invoke a method on the class implementing a relation via the CLI
scopes	These are the recommended scope values for relation implementations.

class charmhelpers.core.reactive.relations.**AutoAccessors**
Bases: `type`

Metaclass that converts fields referenced by `auto_accessors` into accessor methods with very basic doc strings.

class charmhelpers.core.reactive.relations.**Conversation**(*relation_name=None*, *units=None*, *scope=None*)
Bases: `object`

Converations are the persistent, evolving, two-way communication between this service and one or more remote services.

Conversations are not limited to a single Juju hook context. They represent the entire set of interactions between the end-points from the time the relation is joined until it is departed.

Conversations evolve over time, moving from one semantic state to the next as the communication progresses.

Conversations may encompass multiple remote services or units. While a database client would connect to only a single database, that database will likely serve several other services. On the other hand, while the database

is only concerned about providing a database to each service as a whole, a load-balancing proxy must consider each unit of each service individually.

Conversations use the idea of `scope` to determine how units and services are grouped together.

depart()
Remove the current remote unit, for the active hook context, from this conversation. This should be called from a *-departed* hook.

TODO: Need to figure out a way to have this called implicitly, to ensure cleaning up of conversations that are no longer needed.

classmethod deserialize(*conversation*)
Deserialize a *serialized* conversation.

get_local(*key*, *default=None*)
Retrieve some data previously set via *set_local()* for this conversation.

get_remote(*key*, *default=None*)
Get a value from the remote end(s) of this conversation.

Note that if a conversation's scope encompasses multiple units, then those units are expected to agree on their data, whether that is through relying on a single leader to set the data or by all units eventually converging to identical data. Thus, this method returns the first value that it finds set by any of its units.

classmethod join(*scope*)
Get or create a conversation for the given scope and active hook context.

The current remote unit for the active hook context will be added to the conversation.

Note: This uses *charmhelpers.core.unitdata* and requires that *flush()* be called.

key
The key under which this conversation will be stored.

classmethod load(*keys*)
Load a set of conversations by their keys.

relation_ids
The set of IDs of the specific relation instances that this conversation is communicating with.

remove_state(*state*)
Remove this conversation from the given state, and potentially deactivate the state if no more conversations are in it.

The relation name will be interpolated in the state name, and it is recommended that it be included to avoid conflicts with states from other relations. For example:

```
conversation.remove_state('{relation_name}.state')
```

If called from a converation handling the relation "foo", this will remove the conversation from the "foo.state" state, and, if no more conversations are in this the state, will deactivate it.

classmethod serialize(*conversation*)
Serialize a conversation instance for storage.

set_local(*key=None*, *value=None*, *data=None*, ***kwdata*)
Locally store some data associated with this conversation.

Data can be passed in either as a single dict, or as key-word args.

For example, if you need to store the previous value of a remote field to determine if it has changed, you can use the following:

```
prev = conversation.get_local('field')
curr = conversation.get_remote('field')
if prev != curr:
    handle_change(prev, curr)
    conversation.set_local('field', curr)
```

Note: This uses *charmhelpers.core.unitdata* and requires that *flush()* be called.

Parameters

- **key** (*str*) – The name of a field to set.
- **value** – A value to set.
- **data** (*dict*) – A mapping of keys to values.
- ****kwdata** – A mapping of keys to values, as keyword arguments.

set_remote(*key=None*, *value=None*, *data=None*, ***kwdata*)
Set data for the remote end(s) of this conversation.

Data can be passed in either as a single dict, or as key-word args.

Note that, in Juju, setting relation data is inherently service scoped. That is, if the conversation only includes a single unit, the data will still be set for that unit's entire service.

However, if this conversation's scope encompasses multiple services, the data will be set for all of those services.

Parameters

- **key** (*str*) – The name of a field to set.
- **value** – A value to set.
- **data** (*dict*) – A mapping of keys to values.
- ****kwdata** – A mapping of keys to values, as keyword arguments.

set_state(*state*)
Activate and put this conversation into the given state.

The relation name will be interpolated in the state name, and it is recommended that it be included to avoid conflicts with states from other relations. For example:

```
conversation.set_state('{relation_name}.state')
```

If called from a converation handling the relation "foo", this will activate the "foo.state" state, and will add this conversation to that state.

Note: This uses *charmhelpers.core.unitdata* and requires that *flush()* be called.

class charmhelpers.core.reactive.relations.**RelationBase**(*relation_name*, *conversations=None*)

Bases: object

The base class for all relation implementations.

auto_accessors = []
Remote field names to be automatically converted into accessors with basic documentation.

These accessors will just call *get_remote()* using the *default conversation*. Note that it is highly recommended that this be used only with *scopes.GLOBAL* scope.

conversation(*scope=None*)
Get a single conversation, by scope, that this relation is currently handling.

If the scope is not given, the correct scope is inferred by the current hook execution context. If there is no current hook execution context, it is assume that there is only a single global conversation scope for this relation. If this relation's scope is not global and there is no current hook execution context, then an error is raised.

conversations()
Return a list of the conversations that this relation is currently handling.

Note that "currently handling" means for the current state or hook context, and not all conversations that might be active for this relation for other states.

classmethod from_name(*relation_name*, *conversations=None*)
Find relation implementation in the current charm, based on the ID of the relation.

Returns A Relation instance, or None

classmethod from_state(*state*)
Find relation implementation in the current charm, based on the name of an active state.

get_local(*key*, *default=None*, *scope=None*)
Retrieve some data previously set via *set_local()*.

In Python, this is equivalent to:

```
relation.conversation(scope).get_local(key, default)
```

See *conversation()* and *Conversation.get_local()*.

get_remote(*key*, *default=None*, *scope=None*)
Get data from the remote end(s) of the *Conversation* with the given scope.

In Python, this is equivalent to:

```
relation.conversation(scope).get_remote(key, default)
```

See *conversation()* and *Conversation.get_remote()*.

relation_name
Name of the relation this instance is handling.

remove_state(*state*, *scope=None*)
Remove the state for the *Conversation* with the given scope.

In Python, this is equivalent to:

```
relation.conversation(scope).remove_state(state)
```

See *conversation()* and *Conversation.remove_state()*.

scope = 'unit'
Conversation scope for this relation.

The conversation scope controls how communication with connected units is aggregated into related *Conversations*, and can be any of the predefined *scopes*, or any arbitrary string. Connected units which share the same scope will be considered part of the same conversation. Data sent to a conversation is sent to all units that are a part of that conversation, and units that are part of a conversation are expected to agree on the data that they send, whether via eventual consistency or by having a single leader set the data.

The default scope is *scopes.UNIT*.

set_local(*key=None*, *value=None*, *data=None*, *scope=None*, ***kwdata*)
Locally store some data, namespaced by the current or given *Conversation* scope.

In Python, this is equivalent to:

```
relation.conversation(scope).set_local(data, scope, **kwdata)
```

See *conversation()* and *Conversation.set_local()*.

set_remote(*key=None*, *value=None*, *data=None*, *scope=None*, ***kwdata*)
Set data for the remote end(s) of the *Conversation* with the given scope.

In Python, this is equivalent to:

```
relation.conversation(scope).set_remote(data, scope, **kwdata)
```

See *conversation()* and *Conversation.set_remote()*.

set_state(*state*, *scope=None*)
Set the state for the *Conversation* with the given scope.

In Python, this is equivalent to:

```
relation.conversation(scope).set_state(state)
```

See *conversation()* and *Conversation.set_state()*.

charmhelpers.core.reactive.relations.**relation_call**(*method*, *relation_name=None*, *state=None*, **args*)
Invoke a method on the class implementing a relation via the CLI

class charmhelpers.core.reactive.relations.**scopes**
Bases: object

These are the recommended scope values for relation implementations.

To use, simply set the `scope` class variable to one of these:

```
class MyRelationClient(RelationBase):
    scope = scopes.SERVICE
```

GLOBAL = 'global'
All connected services and units for this relation will share a single conversation. The same data will be broadcast to every remote unit, and retrieved data will be aggregated across all remote units and is expected to either eventually agree or be set by a single leader.

SERVICE = 'service'
Each connected service for this relation will have its own conversation. The same data will be broadcast to every unit of each service's conversation, and data from all units of each service will be aggregated and is expected to either eventually agree or be set by a single leader.

UNIT = 'unit'
Each connected unit for this relation will have its own conversation. This is the default scope. Each unit's data will be retrieved individually, but note that due to how Juju works, the same data is still broadcast to all units of a single service.

charmhelpers.core.reactive.bus

ExternalHandler	A variant Handler for external executable actions (such as bash scripts).
Handler	Class representing a reactive state handler.
StateWatch	
all_states	Assert that all desired_states are active
any_hook	Assert that the currently executing hook matches one of the given patterns.
any_states	Assert that any of the desired_states are active
discover	Discover handlers based on convention.
dispatch	Dispatch registered handlers.
get_state	Return the value associated with an active state, or None
get_states	Return a mapping of all active states to their values
remove_state	Remove / deactivate a state
set_state	Set the given state as active, optionally associating with a relation

class charmhelpers.core.reactive.bus.**ExternalHandler**(*filepath*)
Bases: *charmhelpers.core.reactive.bus.Handler*

A variant Handler for external executable actions (such as bash scripts).

External handlers must adhere to the following protocol:

- The handler can be any executable
- When invoked with the `--test` command-line flag, it should exit with an exit code of zero to indicate that the handler should be invoked, and a non-zero exit code to indicate that it need not be invoked. It can also provide a line of output to be passed to the `--invoke` call, e.g., to indicate which sub-handlers should be invoked. The handler should **not** perform its action when given this flag.
- When invoked with the `--invoke` command-line flag (which will be followed by any output returned by the `--test` call), the handler should perform its action(s).

id()

invoke()
Call the external handler to be invoked.

classmethod register(*filepath*)

test()
Call the external handler to test whether it should be invoked.

class charmhelpers.core.reactive.bus.**Handler**(*action*)
Bases: `object`

Class representing a reactive state handler.

add_args(*args*)
Add arguments to be passed to the action when invoked.

Parameters **args** – Any sequence or iterable, which will be lazily evaluated to provide args. Subsequent calls to *add_args()* can be used to add additional arguments.

add_predicate(*predicate*)
Add a new predicate callback to this handler.

classmethod clear()
Clear all registered handlers.

classmethod get(*action*)
Get or register a handler for the given action.

Parameters

- **action** (*func*) – Callback that is called when invoking the Handler
- **args_source** (*func*) – Optional callback that generates args for the action

classmethod get_handlers()
Clear all registered handlers.

id()

invoke()
Invoke this handler.

test()
Check the predicate(s) and return True if this handler should be invoked.

class charmhelpers.core.reactive.bus.**StateWatch**
Bases: object

classmethod change(*state*)

classmethod commit()

classmethod iteration(*i*)

key = 'reactive.state_watch'

classmethod reset()

classmethod watch(*watcher*, *states*)

charmhelpers.core.reactive.bus.**all_states**(**desired_states*)
Assert that all desired_states are active

charmhelpers.core.reactive.bus.**any_hook**(**hook_patterns*)
Assert that the currently executing hook matches one of the given patterns.

Each pattern will match one or more hooks, and can use the following special syntax:

- `db-relation-{joined,changed}` can be used to match multiple hooks (in this case, `db-relation-joined` and `db-relation-changed`).
- `{provides:mysql}-relation-joined` can be used to match a relation hook by the role and interface instead of the relation name. The role must be one of `provides`, `requires`, or `peer`.
- The previous two can be combined, of course: `{provides:mysql}-relation-{joined,changed}`

charmhelpers.core.reactive.bus.**any_states**(**desired_states*)
Assert that any of the desired_states are active

charmhelpers.core.reactive.bus.**discover**()
Discover handlers based on convention.

Handlers will be loaded from the following directories and their subdirectories:

- `$CHARM_DIR/reactive/`
- `$CHARM_DIR/hooks/reactive/`
- `$CHARM_DIR/hooks/relations/`

They can be Python files, in which case they will be imported and decorated functions registered. Or they can be executables, in which case they must adhere to the *ExternalHandler* protocol.

charmhelpers.core.reactive.bus.**dispatch**()
Dispatch registered handlers.

Handlers are dispatched according to the following rules:

- Handlers are repeatedly tested and invoked in iterations, until the system settles into quiescence (that is, until no new handlers match to be invoked).
- In the first iteration, *@hook* and @action handlers will be invoked, if they match.
- In subsequent iterations, other handlers are invoked, if they match.
- Added states will not trigger new handlers until the next iteration, to ensure that chained states are invoked in a predictable order.
- Removed states will cause the current set of matched handlers to be re-tested, to ensure that no handler is invoked after its matching state has been removed.
- Other than the guarantees mentioned above, the order in which matching handlers are invoked is undefined.
- States are preserved between hook and action invocations, and all matching handlers are re-invoked for every hook and action. There are *decorators* and *helpers* to prevent unnecessary reinvocations, such as *only_once()*.

charmhelpers.core.reactive.bus.**get_state**(*state*, *default=None*)
Return the value associated with an active state, or None

charmhelpers.core.reactive.bus.**get_states**()
Return a mapping of all active states to their values

charmhelpers.core.reactive.bus.**remove_state**(*state*)
Remove / deactivate a state

charmhelpers.core.reactive.bus.**set_state**(*state*, *value=None*)
Set the given state as active, optionally associating with a relation

charmhelpers.core.reactive.**main**(*relation_name=None*)
This is the main entry point for the reactive framework. It calls *discover()* to find and load all reactive handlers (e.g., *@when* decorated blocks), and then *dispatch()* to trigger hook and state handlers until the state settles out. Finally, *unitdata.kv().flush* is called to persist the state.

Parameters **relation_name** (*str*) – Optional name of the relation which is being handled.

charmhelpers.core.decorators

charmhelpers.core.decorators.**retry_on_exception**(*num_retries*, *base_delay=0*, *exc_type=<type 'exceptions.Exception'>*)
If the decorated function raises exception exc_type, allow num_retries retry attempts before raise the exception.

charmhelpers.core.fstab

class charmhelpers.core.fstab.**Fstab**(*path=None*)
Bases: _io.FileIO

This class extends file in order to implement a file reader/writer for file */etc/fstab*

DEFAULT_PATH = '/etc/fstab'

class `Entry` (*device*, *mountpoint*, *filesystem*, *options*, *d=0*, *p=0*)
Bases: `object`

Entry class represents a non-comment line on the */etc/fstab* file

classmethod `Fstab.add` (*device*, *mountpoint*, *filesystem*, *options=None*, *path=None*)

`Fstab.add_entry` (*entry*)

`Fstab.entries`

`Fstab.get_entry_by_attr` (*attr*, *value*)

classmethod `Fstab.remove_by_mountpoint` (*mountpoint*, *path=None*)

`Fstab.remove_entry` (*entry*)

charmhelpers.core.hookenv

Config	A dictionary representation of the charm's config.yaml, with some
Hooks	A convenient handler for hook functions.
Serializable	Wrapper, an object that can be serialized to yaml or json
UnregisteredHookError	Raised when an undefined hook is called
action_fail	Sets the action status to failed and sets the error message.
action_get	Gets the value of an action parameter, or all key/value param pairs
action_set	Sets the values to be returned after the action finishes
atexit	Schedule a callback to run on successful hook completion.
atstart	Schedule a callback to run before the main hook.
cached	Cache return values for multiple executions of func + args
charm_dir	Return the root directory of the current charm
charm_name	Get the name of the current charm as is specified on metadata.yaml
close_port	Close a service network port
config	Juju charm configuration
execution_environment	A convenient bundling of the current execution context
flush	Flushes any entries from function cache where the
has_juju_version	Return True if the Juju version is at least the provided version
hook_name	The name of the currently executing hook
in_relation_hook	Determine whether we're running in a relation hook
interface_to_relations	Given an interface, return a list of relation names for the current charm that use that interface.
is_leader	
is_relation_made	Determine whether a relation is established by checking for presence of key(s).
juju_version	Full version string (eg.
leader_get	
leader_set	
local_unit	Local unit ID
log	Write a message to the juju log
metadata	Get the current charm metadata.yaml contents as a python object

Continued on next page

Table 4.5 – continued from previous page

open_port	Open a service network port
related_units	A list of related units
relation_clear	Clears any relation data already set on relation r_id
relation_for_unit	Get the json represenation of a unit's relation
relation_get	Get relation information
relation_id	The relation ID for the current or a specified relation
relation_ids	A list of relation_ids
relation_set	Set relation information for the current unit
relation_to_interface	Given the name of a relation, return the interface that relation uses.
relation_to_role_and_interface	Given the name of a relation, return the role and the name of the interface that relation uses (where role is one of `provides`, `requires`, or `peer`).
relation_type	The scope for the current relation hook
relation_types	Get a list of relation types supported by this charm
relations	Get a nested dictionary of relation data for all related units
relations_for_id	Get relations of a specific relation ID
relations_of_type	Get relations of a specific type
remote_service_name	The remote service name for a given relation-id (or the current relation)
remote_unit	The remote unit for the current relation hook
role_and_interface_to_relations	Given a role and interface name, return a list of relation names for the current charm that use that interface under that role (where role is one of `provides`, `requires`, or `peer`).
service_name	The name service group this unit belongs to
status_get	Retrieve the previously set juju workload state
status_set	Set the workload state with a message
translate_exc	
unit_get	Get the unit ID for the remote unit
unit_private_ip	Get this unit's private IP address
unit_public_ip	Get this unit's public IP address

Interactions with the Juju environment

class `charmhelpers.core.hookenv.`**`Config`**(**args*, ***kw*)

Bases: `dict`

A dictionary representation of the charm's config.yaml, with some extra features:

- See which values in the dictionary have changed since the previous hook.
- For values that have changed, see what the previous value was.
- Store arbitrary data for use in a later hook.

NOTE: Do not instantiate this object directly - instead call `hookenv.config()`, which will return an instance of *Config*.

Example usage:

```
>>> # inside a hook
>>> from charmhelpers.core import hookenv
>>> config = hookenv.config()
>>> config['foo']
```

```
'bar'
>>> # store a new key/value for later use
>>> config['mykey'] = 'myval'


>>> # user runs `juju set mycharm foo=baz`
>>> # now we're inside subsequent config-changed hook
>>> config = hookenv.config()
>>> config['foo']
'baz'
>>> # test to see if this val has changed since last hook
>>> config.changed('foo')
True
>>> # what was the previous value?
>>> config.previous('foo')
'bar'
>>> # keys/values that we add are preserved across hooks
>>> config['mykey']
'myval'
```

CONFIG_FILE_NAME = '.juju-persistent-config'

changed(*key*)
Return True if the current value for this key is different from the previous value.

load_previous(*path=None*)
Load previous copy of config from disk.

In normal usage you don't need to call this method directly - it is called automatically at object initialization.

Parameters **path** – File path from which to load the previous config. If *None*, config is loaded from the default location. If *path* is specified, subsequent *save()* calls will write to the same path.

previous(*key*)
Return previous value for this key, or None if there is no previous value.

save()
Save this config to disk.

If the charm is using the `Services Framework` or :meth:'@hook <Hooks.hook>' decorator, this is called automatically at the end of successful hook execution. Otherwise, it should be called directly by user code.

To disable automatic saves, set `implicit_save=False` on this instance.

class charmhelpers.core.hookenv.**Hooks**(*config_save=None*)
Bases: `object`

A convenient handler for hook functions.

Example:

```
hooks = Hooks()

# register a hook, taking its name from the function name
@hooks.hook()
def install():
    pass  # your code here
```

```
# register a hook, providing a custom hook name
@hooks.hook("config-changed")
def config_changed():
    pass  # your code here

if __name__ == "__main__":
    # execute a hook based on the name the program is called by
    hooks.execute(sys.argv)
```

execute(*args*)
Execute a registered hook based on args[0]

hook(**hook_names*)
Decorator, registering them as hooks

register(*name*, *function*)
Register a hook

class charmhelpers.core.hookenv.**Serializable**(*obj*)
Bases: UserDict.UserDict

Wrapper, an object that can be serialized to yaml or json

json()
Serialize the object to json

yaml()
Serialize the object to yaml

exception charmhelpers.core.hookenv.**UnregisteredHookError**
Bases: exceptions.Exception

Raised when an undefined hook is called

charmhelpers.core.hookenv.**action_fail**(*message*)
Sets the action status to failed and sets the error message.

The results set by action_set are preserved.

charmhelpers.core.hookenv.**action_get**(**args*, ***kwargs*)
Gets the value of an action parameter, or all key/value param pairs

charmhelpers.core.hookenv.**action_set**(*values*)
Sets the values to be returned after the action finishes

charmhelpers.core.hookenv.**atexit**(*callback*, **args*, ***kwargs*)
Schedule a callback to run on successful hook completion.

Callbacks are run in the reverse order that they were added.

charmhelpers.core.hookenv.**atstart**(*callback*, **args*, ***kwargs*)
Schedule a callback to run before the main hook.

Callbacks are run in the order they were added.

This is useful for modules and classes to perform initialization and inject behavior. In particular:

- Run common code before all of your hooks, such as logging the hook name or interesting relation data.
- Defer object or module initialization that requires a hook context until we know there actually is a hook context, making testing easier.
- Rather than requiring charm authors to include boilerplate to invoke your helper's behavior, have it run automatically if your object is instantiated or module imported.

This is not at all useful after your hook framework as been launched.

charmhelpers.core.hookenv.**cached**(*func*)
Cache return values for multiple executions of func + args

For example:

```
@cached
def unit_get(attribute):
    pass

unit_get('test')
```

will cache the result of unit_get + 'test' for future calls.

charmhelpers.core.hookenv.**charm_dir**()
Return the root directory of the current charm

charmhelpers.core.hookenv.**charm_name**(**args*, ***kwargs*)
Get the name of the current charm as is specified on metadata.yaml

charmhelpers.core.hookenv.**close_port**(*port*, *protocol='TCP'*)
Close a service network port

charmhelpers.core.hookenv.**config**(**args*, ***kwargs*)
Juju charm configuration

charmhelpers.core.hookenv.**execution_environment**()
A convenient bundling of the current execution context

charmhelpers.core.hookenv.**flush**(*key*)
Flushes any entries from function cache where the key is found in the function+args

charmhelpers.core.hookenv.**has_juju_version**(**args*, ***kwargs*)
Return True if the Juju version is at least the provided version

charmhelpers.core.hookenv.**hook_name**()
The name of the currently executing hook

charmhelpers.core.hookenv.**in_relation_hook**()
Determine whether we're running in a relation hook

charmhelpers.core.hookenv.**interface_to_relations**(**args*, ***kwargs*)
Given an interface, return a list of relation names for the current charm that use that interface.

Returns A list of relation names.

charmhelpers.core.hookenv.**is_leader**(**args*, ***kwargs*)

charmhelpers.core.hookenv.**is_relation_made**(**args*, ***kwargs*)
Determine whether a relation is established by checking for presence of key(s). If a list of keys is provided, they must all be present for the relation to be identified as made

charmhelpers.core.hookenv.**juju_version**(**args*, ***kwargs*)
Full version string (eg. '1.23.3.1-trusty-amd64')

charmhelpers.core.hookenv.**leader_get**(**args*, ***kwargs*)

charmhelpers.core.hookenv.**leader_set**(**args*, ***kwargs*)

charmhelpers.core.hookenv.**local_unit**()
Local unit ID

charmhelpers.core.hookenv.**log**(*message*, *level=None*)
Write a message to the juju log

charmhelpers.core.hookenv.**metadata**(**args*, ***kwargs*)
Get the current charm metadata.yaml contents as a python object

charmhelpers.core.hookenv.**open_port**(*port*, *protocol='TCP'*)
Open a service network port

charmhelpers.core.hookenv.**related_units**(**args*, ***kwargs*)
A list of related units

charmhelpers.core.hookenv.**relation_clear**(*r_id=None*)
Clears any relation data already set on relation r_id

charmhelpers.core.hookenv.**relation_for_unit**(**args*, ***kwargs*)
Get the json represenation of a unit's relation

charmhelpers.core.hookenv.**relation_get**(**args*, ***kwargs*)
Get relation information

charmhelpers.core.hookenv.**relation_id**(**args*, ***kwargs*)
The relation ID for the current or a specified relation

charmhelpers.core.hookenv.**relation_ids**(**args*, ***kwargs*)
A list of relation_ids

charmhelpers.core.hookenv.**relation_set**(*relation_id=None*, *relation_settings=None*, ***kwargs*)
Set relation information for the current unit

charmhelpers.core.hookenv.**relation_to_interface**(**args*, ***kwargs*)
Given the name of a relation, return the interface that relation uses.

Returns The interface name, or `None`.

charmhelpers.core.hookenv.**relation_to_role_and_interface**(**args*, ***kwargs*)
Given the name of a relation, return the role and the name of the interface that relation uses (where role is one of `provides`, `requires`, or `peer`).

Returns A tuple containing `(role, interface)`, or `(None, None)`.

charmhelpers.core.hookenv.**relation_type**()
The scope for the current relation hook

charmhelpers.core.hookenv.**relation_types**(**args*, ***kwargs*)
Get a list of relation types supported by this charm

charmhelpers.core.hookenv.**relations**(**args*, ***kwargs*)
Get a nested dictionary of relation data for all related units

charmhelpers.core.hookenv.**relations_for_id**(**args*, ***kwargs*)
Get relations of a specific relation ID

charmhelpers.core.hookenv.**relations_of_type**(**args*, ***kwargs*)
Get relations of a specific type

charmhelpers.core.hookenv.**remote_service_name**(**args*, ***kwargs*)
The remote service name for a given relation-id (or the current relation)

charmhelpers.core.hookenv.**remote_unit**()
The remote unit for the current relation hook

charmhelpers.core.hookenv.**role_and_interface_to_relations**(**args*, ***kwargs*)
Given a role and interface name, return a list of relation names for the current charm that use that interface under that role (where role is one of `provides`, `requires`, or `peer`).

Returns A list of relation names.

charmhelpers.core.hookenv.**service_name**()
The name service group this unit belongs to

charmhelpers.core.hookenv.**status_get**()
Retrieve the previously set juju workload state

If the status-set command is not found then assume this is juju < 1.23 and return 'unknown'

charmhelpers.core.hookenv.**status_set**(*workload_state*, *message*)
Set the workload state with a message

Use status-set to set the workload state with a message which is visible to the user via juju status. If the status-set command is not found then assume this is juju < 1.23 and juju-log the message unstead.

workload_state – valid juju workload state. message – status update message

charmhelpers.core.hookenv.**translate_exc**(*from_exc*, *to_exc*)

charmhelpers.core.hookenv.**unit_get**(**args*, ***kwargs*)
Get the unit ID for the remote unit

charmhelpers.core.hookenv.**unit_private_ip**()
Get this unit's private IP address

charmhelpers.core.hookenv.**unit_public_ip**()
Get this unit's public IP address

charmhelpers.core.host

ChecksumError	
add_group	Add a group to the system
add_user_to_group	Add a user to a group
adduser	Add a user to the system
chdir	
check_hash	Validate a file using a cryptographic checksum.
chownr	
cmp_pkgrevno	Compare supplied revno with the revno of the installed package
file_hash	Generate a hash checksum of the contents of 'path' or None if not found.
fstab_add	Adds the given device entry to the /etc/fstab file
fstab_remove	Remove the given mountpoint entry from /etc/fstab
get_nic_hwaddr	
get_nic_mtu	
lchownr	
list_nics	Return a list of nics of given type(s)
lsb_release	Return /etc/lsb-release in a dict
mkdir	Create a directory
mount	Mount a filesystem at a particular mountpoint
mounts	Get a list of all mounted volumes as [[mount-point,device],[...]]
path_hash	Generate a hash checksum of all files matching 'path'.
pwgen	Generate a random pasword.
restart_on_change	Restart services based on configuration files changing
rsync	Replicate the contents of a path

Continued on next page

Table 4.6 – continued from previous page

service	Control a system service
service_available	Determine whether a system service is available
service_reload	Reload a system service, optionally falling back to restart if
service_restart	Restart a system service
service_running	Determine whether a system service is running
service_start	Start a system service
service_stop	Stop a system service
set_nic_mtu	Set MTU on a network interface
symlink	Create a symbolic link
umount	Unmount a filesystem
write_file	Create or overwrite a file with the contents of a byte string.

Tools for working with the host system

exception charmhelpers.core.host.**ChecksumError**
Bases: exceptions.ValueError

charmhelpers.core.host.**add_group**(*group_name*, *system_group=False*)
Add a group to the system

charmhelpers.core.host.**add_user_to_group**(*username*, *group*)
Add a user to a group

charmhelpers.core.host.**adduser**(*username*, *password=None*, *shell='/bin/bash'*, *system_user=False*)
Add a user to the system

charmhelpers.core.host.**chdir**(**args*, ***kwds*)

charmhelpers.core.host.**check_hash**(*path*, *checksum*, *hash_type='md5'*)
Validate a file using a cryptographic checksum.

Parameters

- **checksum** (*str*) – Value of the checksum used to validate the file.
- **hash_type** (*str*) – Hash algorithm used to generate *checksum*. Can be any hash alrgorithm supported by hashlib, such as md5, sha1, sha256, sha512, etc.

Raises ***ChecksumError*** – If the file fails the checksum

charmhelpers.core.host.**chownr**(*path*, *owner*, *group*, *follow_links=True*)

charmhelpers.core.host.**cmp_pkgrevno**(*package*, *revno*, *pkgcache=None*)
Compare supplied revno with the revno of the installed package

•1 => Installed revno is greater than supplied arg

•0 => Installed revno is the same as supplied arg

•-1 => Installed revno is less than supplied arg

This function imports apt_cache function from charmhelpers.fetch if the pkgcache argument is None. Be sure to add charmhelpers.fetch if you call this function, or pass an apt_pkg.Cache() instance.

charmhelpers.core.host.**file_hash**(*path*, *hash_type='md5'*)
Generate a hash checksum of the contents of 'path' or None if not found.

Parameters **hash_type** (*str*) – Any hash alrgorithm supported by hashlib, such as md5, sha1, sha256, sha512, etc.

charmhelpers.core.host.**fstab_add**(*dev*, *mp*, *fs*, *options=None*)
Adds the given device entry to the /etc/fstab file

charmhelpers.core.host.**fstab_remove**(*mp*)
Remove the given mountpoint entry from /etc/fstab

charmhelpers.core.host.**get_nic_hwaddr**(*nic*)

charmhelpers.core.host.**get_nic_mtu**(*nic*)

charmhelpers.core.host.**lchownr**(*path*, *owner*, *group*)

charmhelpers.core.host.**list_nics**(*nic_type*)
Return a list of nics of given type(s)

charmhelpers.core.host.**lsb_release**()
Return /etc/lsb-release in a dict

charmhelpers.core.host.**mkdir**(*path*, *owner='root'*, *group='root'*, *perms=365*, *force=False*)
Create a directory

charmhelpers.core.host.**mount**(*device*, *mountpoint*, *options=None*, *persist=False*, *filesystem='ext3'*)
Mount a filesystem at a particular mountpoint

charmhelpers.core.host.**mounts**()
Get a list of all mounted volumes as [[mountpoint,device],[...]]

charmhelpers.core.host.**path_hash**(*path*)
Generate a hash checksum of all files matching 'path'. Standard wildcards like '*' and '?' are supported, see documentation for the 'glob' module for more information.

Returns dict: A { filename: hash } dictionary for all matched files. Empty if none found.

charmhelpers.core.host.**pwgen**(*length=None*)
Generate a random pasword.

charmhelpers.core.host.**restart_on_change**(*restart_map*, *stopstart=False*)
Restart services based on configuration files changing

This function is used a decorator, for example:

```
@restart_on_change({
    '/etc/ceph/ceph.conf': [ 'cinder-api', 'cinder-volume' ]
    '/etc/apache/sites-enabled/*': [ 'apache2' ]
    })
def config_changed():
    pass  # your code here
```

In this example, the cinder-api and cinder-volume services would be restarted if /etc/ceph/ceph.conf is changed by the ceph_client_changed function. The apache2 service would be restarted if any file matching the pattern got changed, created or removed. Standard wildcards are supported, see documentation for the 'glob' module for more information.

charmhelpers.core.host.**rsync**(*from_path*, *to_path*, *flags='-r'*, *options=None*)
Replicate the contents of a path

charmhelpers.core.host.**service**(*action*, *service_name*)
Control a system service

charmhelpers.core.host.**service_available**(*service_name*)
Determine whether a system service is available

charmhelpers.core.host.**service_reload**(*service_name*, *restart_on_failure=False*)
Reload a system service, optionally falling back to restart if reload fails

charmhelpers.core.host.**service_restart**(*service_name*)
Restart a system service

charmhelpers.core.host.**service_running**(*service*)
Determine whether a system service is running

charmhelpers.core.host.**service_start**(*service_name*)
Start a system service

charmhelpers.core.host.**service_stop**(*service_name*)
Stop a system service

charmhelpers.core.host.**set_nic_mtu**(*nic*, *mtu*)
Set MTU on a network interface

charmhelpers.core.host.**symlink**(*source*, *destination*)
Create a symbolic link

charmhelpers.core.host.**umount**(*mountpoint*, *persist=False*)
Unmount a filesystem

charmhelpers.core.host.**write_file**(*path*, *content*, *owner='root'*, *group='root'*, *perms=292*)
Create or overwrite a file with the contents of a byte string.

charmhelpers.core.strutils

charmhelpers.core.strutils.**bool_from_string**(*value*)
Interpret string value as boolean.

Returns True if value translates to True otherwise False.

charmhelpers.core.sysctl

charmhelpers.core.sysctl.**create**(*sysctl_dict*, *sysctl_file*)
Creates a sysctl.conf file from a YAML associative array

Parameters

- **sysctl_dict** (*str*) – a YAML-formatted string of sysctl options eg "{ 'kernel.max_pid': 1337 }"
- **sysctl_file** (*str or unicode*) – path to the sysctl file to be saved

Returns None

charmhelpers.core.templating

charmhelpers.core.templating.**render**(*source*, *target*, *context*, *owner='root'*, *group='root'*, *perms=292*, *templates_dir=None*, *encoding='UTF-8'*)
Render a template.

The *source* path, if not absolute, is relative to the *templates_dir*.

The *target* path should be absolute.

The context should be a dict containing the values to be replaced in the template.

The *owner*, *group*, and *perms* options will be passed to *write_file*.

If omitted, *templates_dir* defaults to the *templates* folder in the charm.

Note: Using this requires python-jinja2; if it is not installed, calling this will attempt to use charmhelpers.fetch.apt_install to install it.

charmhelpers.core.unitdata

Intro

A simple way to store state in units. This provides a key value storage with support for versioned, transactional operation, and can calculate deltas from previous values to simplify unit logic when processing changes.

Hook Integration

There are several extant frameworks for hook execution, including

- charmhelpers.core.hookenv.Hooks
- charmhelpers.core.services.ServiceManager

The storage classes are framework agnostic, one simple integration is via the HookData contextmanager. It will record the current hook execution environment (including relation data, config data, etc.), setup a transaction and allow easy access to the changes from previously seen values. One consequence of the integration is the reservation of particular keys ('rels', 'unit', 'env', 'config', 'charm_revisions') for their respective values.

Here's a fully worked integration example using hookenv.Hooks:

```
from charmhelper.core import hookenv, unitdata

hook_data = unitdata.HookData()
db = unitdata.kv()
hooks = hookenv.Hooks()

@hooks.hook
def config_changed():
    # Print all changes to configuration from previously seen
    # values.
    for changed, (prev, cur) in hook_data.conf.items():
        print('config changed', changed,
              'previous value', prev,
              'current value',  cur)

    # Get some unit specific bookeeping
    if not db.get('pkg_key'):
        key = urllib.urlopen('https://example.com/pkg_key').read()
        db.set('pkg_key', key)

    # Directly access all charm config as a mapping.
    conf = db.getrange('config', True)

    # Directly access all relation data as a mapping
    rels = db.getrange('rels', True)

if __name__ == '__main__':
    with hook_data():
        hook.execute()
```

A more basic integration is via the hook_scope context manager which simply manages transaction scope (and records hook name, and timestamp):

```
>>> from unitdata import kv
>>> db = kv()
>>> with db.hook_scope('install'):
...     # do work, in transactional scope.
...     db.set('x', 1)
>>> db.get('x')
1
```

Usage

Values are automatically json de/serialized to preserve basic typing and complex data struct capabilities (dicts, lists, ints, booleans, etc).

Individual values can be manipulated via get/set:

```
>>> kv.set('y', True)
>>> kv.get('y')
True

# We can set complex values (dicts, lists) as a single key.
>>> kv.set('config', {'a': 1, 'b': True'})

# Also supports returning dictionaries as a record which
# provides attribute access.
>>> config = kv.get('config', record=True)
>>> config.b
True
```

Groups of keys can be manipulated with update/getrange:

```
>>> kv.update({'z': 1, 'y': 2}, prefix="gui.")
>>> kv.getrange('gui.', strip=True)
{'z': 1, 'y': 2}
```

When updating values, its very helpful to understand which values have actually changed and how have they changed. The storage provides a delta method to provide for this:

```
>>> data = {'debug': True, 'option': 2}
>>> delta = kv.delta(data, 'config.')
>>> delta.debug.previous
None
>>> delta.debug.current
True
>>> delta
{'debug': (None, True), 'option': (None, 2)}
```

Note the delta method does not persist the actual change, it needs to be explicitly saved via 'update' method:

```
>>> kv.update(data, 'config.')
```

Values modified in the context of a hook scope retain historical values associated to the hookname.

```
>>> with db.hook_scope('config-changed'):
...       db.set('x', 42)
>>> db.gethistory('x')
[(1, u'x', 1, u'install', u'2015-01-21T16:49:30.038372'),
 (2, u'x', 42, u'config-changed', u'2015-01-21T16:49:30.038786')]
```

class charmhelpers.core.unitdata.**Delta**(*previous*, *current*)
Bases: tuple

current
Alias for field number 1

previous
Alias for field number 0

class charmhelpers.core.unitdata.**DeltaSet**
Bases: *charmhelpers.core.unitdata.Record*

class charmhelpers.core.unitdata.**HookData**
Bases: object

Simple integration for existing hook exec frameworks.

Records all unit information, and stores deltas for processing by the hook.

Sample:

```
from charmhelper.core import hookenv, unitdata

changes = unitdata.HookData()
db = unitdata.kv()
hooks = hookenv.Hooks()

@hooks.hook
def config_changed():
    # View all changes to configuration
    for changed, (prev, cur) in changes.conf.items():
        print('config changed', changed,
              'previous value', prev,
              'current value',  cur)

    # Get some unit specific bookeeping
    if not db.get('pkg_key'):
        key = urllib.urlopen('https://example.com/pkg_key').read()
        db.set('pkg_key', key)

if __name__ == '__main__':
    with changes():
        hook.execute()
```

class charmhelpers.core.unitdata.**Record**
Bases: dict

class charmhelpers.core.unitdata.**Storage**(*path=None*)
Bases: object

Simple key value database for local unit state within charms.

Modifications are not persisted unless *flush()* is called.

To support dicts, lists, integer, floats, and booleans values are automatically json encoded/decoded.

close()

debug(*fh=<open file '<stderr>', mode 'w'>*)

delta(*mapping*, *prefix*)
return a delta containing values that have changed.

flush(*save=True*)

get(*key*, *default=None*, *record=False*)

gethistory(*key*, *deserialize=False*)

getrange(*key_prefix*, *strip=False*)
Get a range of keys starting with a common prefix as a mapping of keys to values.

Parameters

- **key_prefix** (*str*) – Common prefix among all keys
- **strip** (*bool*) – Optionally strip the common prefix from the key names in the returned dict

Return dict A (possibly empty) dict of key-value mappings

hook_scope(**args*, ***kwds*)
Scope all future interactions to the current hook execution revision.

set(*key*, *value*)
Set a value in the database.

Parameters

- **key** (*str*) – Key to set the value for
- **value** – Any JSON-serializable value to be set

unset(*key*)
Remove a key from the database entirely.

unsetrange(*keys=None*, *prefix=''*)
Remove a range of keys starting with a common prefix, from the database entirely.

Parameters

- **keys** (*list*) – List of keys to remove.
- **prefix** (*str*) – Optional prefix to apply to all keys in `keys` before removing.

update(*mapping*, *prefix=''*)
Set the values of multiple keys at once.

Parameters

- **mapping** (*dict*) – Mapping of keys to values
- **prefix** (*str*) – Optional prefix to apply to all keys in *mapping* before setting

charmhelpers.core.unitdata.**kv**()

charmhelpers.core.services [DEPRECATED]

charmhelpers.core.services.base

ManagerCallback	Special case of a callback that takes the *ServiceManager* instance in addition to the service name.
PortManagerCallback	Callback class that will open or close ports, for use as either a start or stop action.
ServiceManager	
close_ports	Callback class that will open or close ports, for use as either a start or stop action.
manage_ports	Callback class that will open or close ports, for use as either a start or stop action.
open_ports	Callback class that will open or close ports, for use as either a start or stop action.
service_restart	Wrapper around host.service_restart to prevent spurious "unknown service" messages in the logs.
service_stop	Wrapper around host.service_stop to prevent spurious "unknown service" messages in the logs.

class charmhelpers.core.services.base.**ServiceManager**(*services=None*)

Bases: object

fire_event(*event_name*, *service_name*, *default=None*)

Fire a data_ready, data_lost, start, or stop event on a given service.

get_service(*service_name*)

Given the name of a registered service, return its service definition.

is_ready(*service_name*)

Determine if a registered service is ready, by checking its 'required_data'.

A 'required_data' item can be any mapping type, and is considered ready if *bool(item)* evaluates as True.

manage()

Handle the current hook by doing The Right Thing with the registered services.

provide_data()

Set the relation data for each provider in the provided_data list.

A provider must have a *name* attribute, which indicates which relation to set data on, and a *provide_data()* method, which returns a dict of data to set.

The *provide_data()* method can optionally accept two parameters:

- remote_service The name of the remote service that the data will be provided to. The *provide_data()* method will be called once for each connected service (not unit). This allows the method to tailor its data to the given service.
- service_ready Whether or not the service definition had all of its requirements met, and thus the data_ready callbacks run.

Note that the provided_data methods are now called **after** the data_ready callbacks are run. This gives the data_ready callbacks a chance to generate any data necessary for the providing to the remote services.

reconfigure_services(**service_names*)

Update all files for one or more registered services, and, if ready, optionally restart them.

If no service names are given, reconfigures all registered services.

save_lost(*service_name*)

Save an indicator that the given service is no longer data_ready.

save_ready(*service_name*)
Save an indicator that the given service is now data_ready.

stop_services(**service_names*)
Stop one or more registered services, by name.

If no service names are given, stops all registered services.

was_ready(*service_name*)
Determine if the given service was previously data_ready.

class charmhelpers.core.services.base.**ManagerCallback**
Bases: object

Special case of a callback that takes the *ServiceManager* instance in addition to the service name.

Subclasses should implement *__call__* which should accept three parameters:

- *manager* The *ServiceManager* instance
- *service_name* The name of the service it's being triggered for
- *event_name* The name of the event that this callback is handling

class charmhelpers.core.services.base.**PortManagerCallback**
Bases: *charmhelpers.core.services.base.ManagerCallback*

Callback class that will open or close ports, for use as either a start or stop action.

charmhelpers.core.services.base.**service_restart**(*service_name*)
Wrapper around host.service_restart to prevent spurious "unknown service" messages in the logs.

charmhelpers.core.services.base.**service_stop**(*service_name*)
Wrapper around host.service_stop to prevent spurious "unknown service" messages in the logs.

charmhelpers.core.services.helpers

HttpRelation	Relation context for the *http* interface.
MysqlRelation	Relation context for the *mysql* interface.
RelationContext	Base class for a context generator that gets relation data from juju.
RequiredConfig	Data context that loads config options with one or more mandatory options.
StoredContext	A data context that always returns the data that it was first created with.
TemplateCallback	Callback class that will render a Jinja2 template, for use as a ready action.
render_template	alias of TemplateCallback
template	alias of TemplateCallback

class charmhelpers.core.services.helpers.**RelationContext**(*name=None*, *additional_required_keys=None*)
Bases: dict

Base class for a context generator that gets relation data from juju.

Subclasses must provide the attributes *name*, which is the name of the interface of interest, *interface*, which is the type of the interface of interest, and *required_keys*, which is the set of keys required for the relation to be considered complete. The data for all interfaces matching the *name* attribute that are complete will used to

populate the dictionary values (see *get_data*, below).

The generated context will be namespaced under the relation *name*, to prevent potential naming conflicts.

Parameters

- **name** (*str*) – Override the relation *name*, since it can vary from charm to charm
- **additional_required_keys** (*list*) – Extend the list of required_keys

get_data()

Retrieve the relation data for each unit involved in a relation and, if complete, store it in a list under *self[self.name]*. This is automatically called when the RelationContext is instantiated.

The units are sorted lexographically first by the service ID, then by the unit ID. Thus, if an interface has two other services, 'db:1' and 'db:2', with 'db:1' having two units, 'wordpress/0' and 'wordpress/1', and 'db:2' having one unit, 'mediawiki/0', all of which have a complete set of data, the relation data for the units will be stored in the order: 'wordpress/0', 'wordpress/1', 'mediawiki/0'.

If you only care about a single unit on the relation, you can just access it as *{{ interface[0]['key'] }}*. However, if you can at all support multiple units on a relation, you should iterate over the list, like:

```
{% for unit in interface -%}
    {{ unit['key'] }}{% if not loop.last %},{% endif %}
{%- endfor %}
```

Note that since all sets of relation data from all related services and units are in a single list, if you need to know which service or unit a set of data came from, you'll need to extend this class to preserve that information.

interface = None

is_ready()

Returns True if all of the *required_keys* are available from any units.

name = None

provide_data()

Return data to be relation_set for this interface.

class charmhelpers.core.services.helpers.**TemplateCallback**(*source*, *target*, *owner='root'*, *group='root'*, *perms=292*)

Bases: *charmhelpers.core.services.base.ManagerCallback*

Callback class that will render a Jinja2 template, for use as a ready action.

Parameters

- **source** (*str*) – The template source file, relative to *$CHARM_DIR/templates*
- **target** (*str*) – The target to write the rendered template to
- **owner** (*str*) – The owner of the rendered file
- **group** (*str*) – The group of the rendered file
- **perms** (*int*) – The permissions of the rendered file

charmhelpers.core.services.helpers.**render_template**

alias of *TemplateCallback*

charmhelpers.core.services.helpers.**template**

alias of *TemplateCallback*

charmhelpers.contrib package

charmhelpers.contrib.ansible package

Charm Helpers ansible - declare the state of your machines.

This helper enables you to declare your machine state, rather than program it procedurally (and have to test each change to your procedures). Your install hook can be as simple as:

```
{{{
import charmhelpers.contrib.ansible


def install():
    charmhelpers.contrib.ansible.install_ansible_support()
    charmhelpers.contrib.ansible.apply_playbook('playbooks/install.yaml')
}}}
```

and won't need to change (nor will its tests) when you change the machine state.

All of your juju config and relation-data are available as template variables within your playbooks and templates. An install playbook looks something like:

```
{{{
---
- hosts: localhost
  user: root

  tasks:
    - name: Add private repositories.
      template:
        src: ../templates/private-repositories.list.jinja2
        dest: /etc/apt/sources.list.d/private.list

    - name: Update the cache.
      apt: update_cache=yes

    - name: Install dependencies.
      apt: pkg={{ item }}
      with_items:
        - python-mimeparse
        - python-webob
        - sunburnt

    - name: Setup groups.
      group: name={{ item.name }} gid={{ item.gid }}
      with_items:
        - { name: 'deploy_user', gid: 1800 }
        - { name: 'service_user', gid: 1500 }

  ...
}}}
```

Read more online about playbooks and standard ansible modules.

A further feature os the ansible hooks is to provide a light weight "action" scripting tool. This is a decorator that you apply to a function, and that function can now receive cli args, and can pass extra args to the playbook.

e.g.

@hooks.action() def some_action(amount, force="False"):

"Usage: some-action AMOUNT [force=True]" # <– shown on error # process the arguments # do some calls # return extra-vars to be passed to ansible-playbook return {

'amount': int(amount), 'type': force,

}

You can now create a symlink to hooks.py that can be invoked like a hook, but with cli params:

link actions/some-action to hooks/hooks.py

actions/some-action amount=10 force=true

class charmhelpers.contrib.ansible.**AnsibleHooks**(*playbook_path*, *default_hooks=None*)

Bases: *charmhelpers.core.hookenv.Hooks*

Run a playbook with the hook-name as the tag.

This helper builds on the standard hookenv.Hooks helper, but additionally runs the playbook with the hook-name specified using –tags (ie. running all the tasks tagged with the hook-name).

Example:

```
hooks = AnsibleHooks(playbook_path='playbooks/my_machine_state.yaml')

# All the tasks within my_machine_state.yaml tagged with 'install'
# will be run automatically after do_custom_work()
@hooks.hook()
def install():
    do_custom_work()

# For most of your hooks, you won't need to do anything other
# than run the tagged tasks for the hook:
@hooks.hook('config-changed', 'start', 'stop')
def just_use_playbook():
    pass

# As a convenience, you can avoid the above noop function by specifying
# the hooks which are handled by ansible-only and they'll be registered
# for you:
# hooks = AnsibleHooks(
#     'playbooks/my_machine_state.yaml',
#     default_hooks=['config-changed', 'start', 'stop'])

if __name__ == "__main__":
    # execute a hook based on the name the program is called by
    hooks.execute(sys.argv)
```

action(**action_names*)

Decorator, registering them as actions

execute(*args*)

Execute the hook followed by the playbook using the hook as tag.

register_action(*name*, *function*)

Register a hook

charmhelpers.contrib.ansible.**apply_playbook**(*playbook*, *tags=None*, *extra_vars=None*)

charmhelpers.contrib.ansible.**install_ansible_support**(*from_ppa=True*, *ppa_location='ppa:rquillo/ansible'*)

Installs the ansible package.

By default it is installed from the PPA linked from the ansible website or from a ppa specified by a charm config..

If from_ppa is empty, you must ensure that the package is available from a configured repository.

charmhelpers.contrib.charmhelpers package

charmhelpers.contrib.charmhelpers.**unit_info**(*service_name*, *item_name*, *data=None*, *unit=None*)

charmhelpers.contrib.charmhelpers.**wait_for_machine**(*num_machines=1*, *timeout=300*)

Wait *timeout* seconds for *num_machines* machines to come up.

This wait_for... function can be called by other wait_for functions whose timeouts might be too short in situations where only a bare Juju setup has been bootstrapped.

Returns A tuple of (num_machines, time_taken). This is used for testing.

charmhelpers.contrib.charmhelpers.**wait_for_page_contents**(*url*, *contents*, *timeout=120*, *validate=None*)

charmhelpers.contrib.charmhelpers.**wait_for_relation**(*service_name*, *relation_name*, *timeout=120*)

Wait *timeout* seconds for a given relation to come up.

charmhelpers.contrib.charmhelpers.**wait_for_unit**(*service_name*, *timeout=480*)

Wait *timeout* seconds for a given service name to come up.

charmhelpers.contrib.charmsupport package

charmhelpers.contrib.charmsupport.nrpe module

Compatibility with the nrpe-external-master charm

class charmhelpers.contrib.charmsupport.nrpe.**Check**(*shortname*, *description*, *check_cmd*)

Bases: object

run()

service_template = '\n#--\n# This file is Juju managed\n#--

shortname_re = '[A-Za-z0-9-_]+$'

write(*nagios_context*, *hostname*, *nagios_servicegroups*)

write_service_config(*nagios_context*, *hostname*, *nagios_servicegroups*)

exception charmhelpers.contrib.charmsupport.nrpe.**CheckException**

Bases: exceptions.Exception

class charmhelpers.contrib.charmsupport.nrpe.**NRPE**(*hostname=None*)

Bases: object

add_check(**args*, ***kwargs*)

nagios_exportdir = '/var/lib/nagios/export'

nagios_logdir = '/var/log/nagios'

nrpe_confdir = '/etc/nagios/nrpe.d'

write()

charmhelpers.contrib.charmsupport.nrpe.**add_haproxy_checks**(*nrpe*, *unit_name*)
Add checks for each service in list

Parameters

- **nrpe** (NRPE) – NRPE object to add check to
- **unit_name** (*str*) – Unit name to use in check description

charmhelpers.contrib.charmsupport.nrpe.**add_init_service_checks**(*nrpe*, *services*, *unit_name*)
Add checks for each service in list

Parameters

- **nrpe** (NRPE) – NRPE object to add check to
- **services** (*list*) – List of services to check
- **unit_name** (*str*) – Unit name to use in check description

charmhelpers.contrib.charmsupport.nrpe.**copy_nrpe_checks**()
Copy the nrpe checks into place

charmhelpers.contrib.charmsupport.nrpe.**get_nagios_hostcontext**(*relation_name='nrpe-external-master'*)
Query relation with nrpe subordinate, return the nagios_host_context

Parameters **relation_name** (*str*) – Name of relation nrpe sub joined to

charmhelpers.contrib.charmsupport.nrpe.**get_nagios_hostname**(*relation_name='nrpe-external-master'*)
Query relation with nrpe subordinate, return the nagios_hostname

Parameters **relation_name** (*str*) – Name of relation nrpe sub joined to

charmhelpers.contrib.charmsupport.nrpe.**get_nagios_unit_name**(*relation_name='nrpe-external-master'*)
Return the nagios unit name prepended with host_context if needed

Parameters **relation_name** (*str*) – Name of relation nrpe sub joined to

charmhelpers.contrib.charmsupport.volumes module

Functions for managing volumes in juju units. One volume is supported per unit. Subordinates may have their own storage, provided it is on its own partition.

Configuration stanzas:

```
volume-ephemeral:
  type: boolean
  default: true
  description: >
    If false, a volume is mounted as sepecified in "volume-map"
    If true, ephemeral storage will be used, meaning that log data
       will only exist as long as the machine. YOU HAVE BEEN WARNED.
volume-map:
  type: string
  default: {}
  description: >
```

```
    YAML map of units to device names, e.g:
      "{ rsyslog/0: /dev/vdb, rsyslog/1: /dev/vdb }"
    Service units will raise a configure-error if volume-ephemeral
    is 'true' and no volume-map value is set. Use 'juju set' to set a
    value and 'juju resolved' to complete configuration.
```

Usage:

```
from charmsupport.volumes import configure_volume, VolumeConfigurationError
from charmsupport.hookenv import log, ERROR
def post_mount_hook():
    stop_service('myservice')
def post_mount_hook():
    start_service('myservice')

if __name__ == '__main__':
    try:
        configure_volume(before_change=pre_mount_hook,
                         after_change=post_mount_hook)
    except VolumeConfigurationError:
        log('Storage could not be configured', ERROR)
```

exception charmhelpers.contrib.charmsupport.volumes.**VolumeConfigurationError**
Bases: exceptions.Exception

Volume configuration data is missing or invalid

charmhelpers.contrib.charmsupport.volumes.**configure_volume**(*before_change=<function <lambda>>*, *after_change=<function <lambda>>*)
Set up storage (or don't) according to the charm's volume configuration. Returns the mount point or "ephemeral". before_change and after_change are optional functions to be called if the volume configuration changes.

charmhelpers.contrib.charmsupport.volumes.**get_config**()
Gather and sanity-check volume configuration data

charmhelpers.contrib.charmsupport.volumes.**managed_mounts**()
List of all mounted managed volumes

charmhelpers.contrib.charmsupport.volumes.**mount_volume**(*config*)

charmhelpers.contrib.charmsupport.volumes.**unmount_volume**(*config*)

charmhelpers.contrib.hahelpers package

charmhelpers.contrib.hahelpers.apache module

charmhelpers.contrib.hahelpers.apache.**get_ca_cert**()

charmhelpers.contrib.hahelpers.apache.**get_cert**(*cn=None*)

charmhelpers.contrib.hahelpers.apache.**install_ca_cert**(*ca_cert*)

charmhelpers.contrib.hahelpers.cluster module

Helpers for clustering and determining "cluster leadership" and other clustering-related helpers.

exception charmhelpers.contrib.hahelpers.cluster.**CRMDCNotFound**
Bases: exceptions.Exception

exception charmhelpers.contrib.hahelpers.cluster.**CRMResourceNotFound**
Bases: exceptions.Exception

exception charmhelpers.contrib.hahelpers.cluster.**HAIncompleteConfig**
Bases: exceptions.Exception

charmhelpers.contrib.hahelpers.cluster.**canonical_url**(*configs*, *vip_setting='vip'*)
Returns the correct HTTP URL to this host given the state of HTTPS configuration and hacluster.

:configs [OSTemplateRenderer: A config tempating object to inspect for] a complete https context.

Vip_setting str: Setting in charm config that specifies VIP address.

charmhelpers.contrib.hahelpers.cluster.**determine_apache_port**(*public_port*, *singlenode_mode=False*)
Description: Determine correct apache listening port based on public IP + state of the cluster.

public_port: int: standard public port for given service

singlenode_mode: boolean: Shuffle ports when only a single unit is present

returns: int: the correct listening port for the HAProxy service

charmhelpers.contrib.hahelpers.cluster.**determine_api_port**(*public_port*, *singlenode_mode=False*)
Determine correct API server listening port based on existence of HTTPS reverse proxy and/or haproxy.

public_port: int: standard public port for given service

singlenode_mode: boolean: Shuffle ports when only a single unit is present

returns: int: the correct listening port for the API service

charmhelpers.contrib.hahelpers.cluster.**eligible_leader**(*resource*)

charmhelpers.contrib.hahelpers.cluster.**get_hacluster_config**(*exclude_keys=None*)
Obtains all relevant configuration from charm configuration required for initiating a relation to hacluster:

ha-bindiface, ha-mcastport, vip

param: exclude_keys: list of setting key(s) to be excluded. returns: dict: A dict containing settings keyed by setting name. raises: HAIncompleteConfig if settings are missing.

charmhelpers.contrib.hahelpers.cluster.**https**()
Determines whether enough data has been provided in configuration or relation data to configure HTTPS . returns: boolean

charmhelpers.contrib.hahelpers.cluster.**is_clustered**()

charmhelpers.contrib.hahelpers.cluster.**is_crm_dc**()
Determine leadership by querying the pacemaker Designated Controller

charmhelpers.contrib.hahelpers.cluster.**is_elected_leader**(*resource*)
Returns True if the charm executing this is the elected cluster leader.

It relies on two mechanisms to determine leadership: 1. If juju is sufficiently new and leadership election is supported, the is_leader command will be used. 2. If the charm is part of a corosync cluster, call corosync to determine leadership. 3. If the charm is not part of a corosync cluster, the leader is determined as being "the alive unit with the lowest unit numer". In other words, the oldest surviving unit.

charmhelpers.contrib.hahelpers.cluster.**is_leader**(*resource*)

charmhelpers.contrib.hahelpers.cluster.**oldest_peer**(*peers*)
Determines who the oldest peer is by comparing unit numbers.

charmhelpers.contrib.hahelpers.cluster.**peer_ips**(*peer_relation='cluster'*, *addr_key='private-address'*)
Return a dict of peers and their private-address

charmhelpers.contrib.hahelpers.cluster.**peer_units**(*peer_relation='cluster'*)

charmhelpers.contrib.network package

charmhelpers.contrib.network.ovs package

Helpers for interacting with OpenvSwitch

charmhelpers.contrib.network.ovs.**add_bridge**(*name*)
Add the named bridge to openvswitch

charmhelpers.contrib.network.ovs.**add_bridge_port**(*name*, *port*, *promisc=False*)
Add a port to the named openvswitch bridge

charmhelpers.contrib.network.ovs.**del_bridge**(*name*)
Delete the named bridge from openvswitch

charmhelpers.contrib.network.ovs.**del_bridge_port**(*name*, *port*)
Delete a port from the named openvswitch bridge

charmhelpers.contrib.network.ovs.**full_restart**()
Full restart and reload of openvswitch

charmhelpers.contrib.network.ovs.**get_certificate**()
Read openvswitch certificate from disk

charmhelpers.contrib.network.ovs.**set_manager**(*manager*)
Set the controller for the local openvswitch

charmhelpers.contrib.network.ip module

charmhelpers.contrib.network.ip.**format_ipv6_addr**(*address*)
If address is IPv6, wrap it in '[]' otherwise return None.

This is required by most configuration files when specifying IPv6 addresses.

charmhelpers.contrib.network.ip.**get_address_in_network**(*network*, *fallback=None*, *fatal=False*)
Get an IPv4 or IPv6 address within the network from the host.

Parameters

- **(str)** (*fallback*) – CIDR presentation format. For example, '192.168.1.0/24'.
- **(str)** – If no address is found, return fallback.
- **(boolean)** (*fatal*) – If no address is found, fallback is not set and fatal is True then exit(1).

charmhelpers.contrib.network.ip.**get_bridge_nics**(*bridge*, *vnic_dir='/sys/devices/virtual/net'*)
Return a list of nics comprising a given bridge on the system.

charmhelpers.contrib.network.ip.**get_bridges**(*vnic_dir='/sys/devices/virtual/net'*)
Return a list of bridges on the system.

charmhelpers.contrib.network.ip.**get_host_ip**(*hostname*, *fallback=None*)
Resolves the IP for a given hostname, or returns the input if it is already an IP.

charmhelpers.contrib.network.ip.**get_hostname**(*address*, *fqdn=True*)
Resolves hostname for given IP, or returns the input if it is already a hostname.

charmhelpers.contrib.network.ip.**get_iface_addr**(*iface='eth0'*, *inet_type='AF_INET'*, *inc_aliases=False*, *fatal=True*, *exc_list=None*)
Return the assigned IP address for a given interface, if any.

charmhelpers.contrib.network.ip.**get_iface_from_addr**(*addr*)
Work out on which interface the provided address is configured.

charmhelpers.contrib.network.ip.**get_ipv6_addr**(**args*, ***kwargs*)

charmhelpers.contrib.network.ip.**is_address_in_network**(*network*, *address*)
Determine whether the provided address is within a network range.

Parameters

- **(str)** (network) – CIDR presentation format. For example, '192.168.1.0/24'.
- **address** – An individual IPv4 or IPv6 address without a net mask or subnet prefix. For example, '192.168.1.1'.

Returns boolean Flag indicating whether address is in network.

charmhelpers.contrib.network.ip.**is_bridge_member**(*nic*)
Check if a given nic is a member of a bridge.

charmhelpers.contrib.network.ip.**is_ip**(*address*)
Returns True if address is a valid IP address.

charmhelpers.contrib.network.ip.**is_ipv6**(*address*)
Determine whether provided address is IPv6 or not.

charmhelpers.contrib.network.ip.**no_ip_found_error_out**(*network*)

charmhelpers.contrib.network.ip.**ns_query**(*address*)

charmhelpers.contrib.network.ip.**sniff_iface**(*f*)
Ensure decorated function is called with a value for iface.

If no iface provided, inject net iface inferred from unit private address.

charmhelpers.contrib.openstack package

charmhelpers.contrib.openstack.templates package

charmhelpers.contrib.openstack.alternatives module

Helper for managing alternatives for file conflict resolution

charmhelpers.contrib.openstack.alternatives.**install_alternative**(*name*, *target*, *source*, *priority=50*)
Install alternative configuration

charmhelpers.contrib.openstack.context module

class charmhelpers.contrib.openstack.context.**AMQPContext**(*ssl_dir=None*, *rel_name='amqp'*, *relation_prefix=None*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**ApacheSSLContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Generates a context for an apache vhost configuration that configures HTTPS reverse proxying for one or many endpoints. Generated context looks something like:

```
{
    'namespace': 'cinder',
    'private_address': 'iscsi.mycinderhost.com',
    'endpoints': [(8776, 8766), (8777, 8767)]
}
```

The endpoints list consists of a tuples mapping external ports to internal ports.

canonical_names()
Figure out which canonical names clients will access this service.

configure_ca()

configure_cert(*cn=None*)

enable_modules()

external_ports = []

get_network_addresses()

For each network configured, return corresponding address and vip (if available).

Returns a list of tuples of the form:

[(address_in_net_a, vip_in_net_a), (address_in_net_b, vip_in_net_b), ...]

or, if no vip(s) available:

[(address_in_net_a, address_in_net_a), (address_in_net_b, address_in_net_b), ...]

interfaces = ['https']

service_namespace = None

class charmhelpers.contrib.openstack.context.**BindHostContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**CephContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Generates context for /etc/ceph/ceph.conf templates.

interfaces = ['ceph']

class charmhelpers.contrib.openstack.context.**DataPortContext**
Bases: *charmhelpers.contrib.openstack.context.NeutronPortContext*

class charmhelpers.contrib.openstack.context.**ExternalPortContext**
Bases: *charmhelpers.contrib.openstack.context.NeutronPortContext*

class charmhelpers.contrib.openstack.context.**HAProxyContext**(*singlenode_mode=False*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Provides half a context for the haproxy template, which describes all peers to be included in the cluster. Each charm needs to include its own context generator that describes the port mapping.

interfaces = ['cluster']

class charmhelpers.contrib.openstack.context.**IdentityServiceContext**(*service=None*, *service_user=None*, *rel_name='identity-service'*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**ImageServiceContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

interfaces = ['image-service']

class charmhelpers.contrib.openstack.context.**LogLevelContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**NetworkServiceContext**(*rel_name='quantum-network-service'*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**NeutronAPIContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Inspects current neutron-plugin-api relation for neutron settings. Return defaults if it is not present.

get_neutron_options(*rdata*)

interfaces = ['neutron-plugin-api']

class charmhelpers.contrib.openstack.context.**NeutronContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

calico_ctxt()

interfaces = []

n1kv_ctxt()

network_manager

neutron_ctxt()

neutron_security_groups

nuage_ctxt()

nvp_ctxt()

ovs_ctxt()

packages

plugin

class charmhelpers.contrib.openstack.context.**NeutronPortContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

NIC_PREFIXES = ['eth', 'bond']

resolve_ports(*ports*)
Resolve NICs not yet bound to bridge(s)

If hwaddress provided then returns resolved hwaddress otherwise NIC.

class charmhelpers.contrib.openstack.context.**NotificationDriverContext**(*zmq_relation='zeromq-configuration'*, *amqp_relation='amqp'*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**OSConfigFlagContext**(*charm_flag='config-flags'*, *template_flag='user_config_flags'*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Provides support for user-defined config flags.

Users can define a comma-seperated list of key=value pairs in the charm configuration and apply them at any point in any file by using a template flag.

Sometimes users might want config flags inserted within a specific section so this class allows users to specify the template flag name, allowing for multiple template flags (sections) within the same context.

NOTE: the value of config-flags may be a comma-separated list of key=value pairs and some Openstack config files support comma-separated lists as values.

exception charmhelpers.contrib.openstack.context.**OSContextError**
Bases: exceptions.Exception

class charmhelpers.contrib.openstack.context.**OSContextGenerator**
Bases: object

Base class for all context generators.

interfaces = []

class charmhelpers.contrib.openstack.context.**PhyNICMTUContext**
Bases: *charmhelpers.contrib.openstack.context.DataPortContext*

class charmhelpers.contrib.openstack.context.**PostgresqlDBContext**(*database=None*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

interfaces = ['pgsql-db']

class charmhelpers.contrib.openstack.context.**SharedDBContext**(*database=None*, *user=None*, *relation_prefix=None*, *ssl_dir=None*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

interfaces = ['shared-db']

class charmhelpers.contrib.openstack.context.**SubordinateConfigContext**(*service*, *config_file*, *interface*)
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

Responsible for inspecting relations to subordinates that may be exporting required config via a json blob.

The subordinate interface allows subordinates to export their configuration requirements to the principle for multiple config files and multiple serivces. Ie, a subordinate that has interfaces to both glance and nova may export to following yaml blob as json:

```
glance:
    /etc/glance/glance-api.conf:
        sections:
            DEFAULT:
                - [key1, value1]
    /etc/glance/glance-registry.conf:
            MYSECTION:
                - [key2, value2]
nova:
    /etc/nova/nova.conf:
        sections:
            DEFAULT:
                - [key3, value3]
```

It is then up to the principle charms to subscribe this context to the service+config file it is interestd in. Configuration data will be available in the template context, in glance's case, as:

```
ctxt = {
    ... other context ...
    'subordinate_config': {
        'DEFAULT': {
            'key1': 'value1',
        },
        'MYSECTION': {
            'key2': 'value2',
        },
    }
}
```

class charmhelpers.contrib.openstack.context.**SysctlContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

This context check if the 'sysctl' option exists on configuration then creates a file with the loaded contents

class charmhelpers.contrib.openstack.context.**SyslogContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

class charmhelpers.contrib.openstack.context.**WorkerConfigContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

num_cpus

class charmhelpers.contrib.openstack.context.**ZeroMQContext**
Bases: *charmhelpers.contrib.openstack.context.OSContextGenerator*

interfaces = ['zeromq-configuration']

charmhelpers.contrib.openstack.context.**config_flags_parser**(*config_flags*)
Parses config flags string into dict.

This parsing method supports a few different formats for the config flag values to be parsed:

1. A string in the simple format of key=value pairs, with the possibility of specifying multiple key value pairs within the same string. For example, a string in the format of 'key1=value1, key2=value2' will return a dict of:

   {'key1': 'value1', 'key2': 'value2'}.

2. A string in the above format, but supporting a comma-delimited list of values for the same key. For example, a string in the format of 'key1=value1, key2=value3,value4,value5' will return a dict of:

   {'key1', 'value1', 'key2', 'value2,value3,value4'}

3. A string containing a colon character (:) prior to an equal character (=) will be treated as yaml and parsed as such. This can be used to specify more complex key value pairs. For example, a string in the format of 'key1: subkey1=value1, subkey2=value2' will return a dict of:

   {'key1', 'subkey1=value1, subkey2=value2'}

The provided config_flags string may be a list of comma-separated values which themselves may be comma-separated list of values.

charmhelpers.contrib.openstack.context.**context_complete**(*ctxt*)

charmhelpers.contrib.openstack.context.**db_ssl**(*rdata*, *ctxt*, *ssl_dir*)

charmhelpers.contrib.openstack.context.**ensure_packages**(*packages*)
Install but do not upgrade required plugin packages.

charmhelpers.contrib.openstack.neutron module

charmhelpers.contrib.openstack.neutron.**determine_dkms_package**()
Determine which DKMS package should be used based on kernel version

charmhelpers.contrib.openstack.neutron.**headers_package**()
Ensures correct linux-headers for running kernel are installed, for building DKMS package

charmhelpers.contrib.openstack.neutron.**kernel_version**()
Retrieve the current major kernel version as a tuple e.g. (3, 13)

charmhelpers.contrib.openstack.neutron.**network_manager**()
Deals with the renaming of Quantum to Neutron in H and any situations that require compatability (eg, deploying H with network-manager=quantum, upgrading from G).

charmhelpers.contrib.openstack.neutron.**neutron_plugin_attribute**(*plugin*, *attr*, *net_manager=None*)

charmhelpers.contrib.openstack.neutron.**neutron_plugins**()

charmhelpers.contrib.openstack.neutron.**parse_bridge_mappings**(*mappings*)
Parse bridge mappings.

Mappings must be a space-delimited list of provider:bridge mappings.

Returns dict of the form {provider:bridge}.

charmhelpers.contrib.openstack.neutron.**parse_data_port_mappings**(*mappings*, *default_bridge='br-data'*)
Parse data port mappings.

Mappings must be a space-delimited list of bridge:port mappings.

Returns dict of the form {bridge:port}.

charmhelpers.contrib.openstack.neutron.**parse_mappings**(*mappings*)

charmhelpers.contrib.openstack.neutron.**parse_vlan_range_mappings**(*mappings*)
Parse vlan range mappings.

Mappings must be a space-delimited list of provider:start:end mappings.

The start:end range is optional and may be omitted.

Returns dict of the form {provider: (start, end)}.

charmhelpers.contrib.openstack.neutron.**quantum_plugins**()

charmhelpers.contrib.openstack.templating module

exception charmhelpers.contrib.openstack.templating.**OSConfigException**
Bases: exceptions.Exception

class charmhelpers.contrib.openstack.templating.**OSConfigRenderer**(*templates_dir*, *openstack_release*)
Bases: object

This class provides a common templating system to be used by OpenStack charms. It is intended to help charms share common code and templates, and ease the burden of managing config templates across multiple OpenStack releases.

Basic usage:

```
# import some common context generates from charmhelpers
from charmhelpers.contrib.openstack import context

# Create a renderer object for a specific OS release.
configs = OSConfigRenderer(templates_dir='/tmp/templates',
                           openstack_release='folsom')
# register some config files with context generators.
configs.register(config_file='/etc/nova/nova.conf',
                 contexts=[context.SharedDBContext(),
                           context.AMQPContext()])
configs.register(config_file='/etc/nova/api-paste.ini',
                 contexts=[context.IdentityServiceContext()])
configs.register(config_file='/etc/haproxy/haproxy.conf',
                 contexts=[context.HAProxyContext()])
# write out a single config
configs.write('/etc/nova/nova.conf')
# write out all registered configs
configs.write_all()
```

OpenStack Releases and template loading

When the object is instantiated, it is associated with a specific OS release. This dictates how the template loader will be constructed.

The constructed loader attempts to load the template from several places in the following order: - from the most recent OS release-specific template dir (if one exists) - the base templates_dir - a template directory shipped in the charm with this helper file.

For the example above, '/tmp/templates' contains the following structure:

```
/tmp/templates/nova.conf
/tmp/templates/api-paste.ini
/tmp/templates/grizzly/api-paste.ini
/tmp/templates/havana/api-paste.ini
```

Since it was registered with the grizzly release, it first seraches the grizzly directory for nova.conf, then the templates dir.

When writing api-paste.ini, it will find the template in the grizzly directory.

If the object were created with folsom, it would fall back to the base templates dir for its api-paste.ini template.

This system should help manage changes in config files through openstack releases, allowing charms to fall back to the most recently updated config template for a given release

The haproxy.conf, since it is not shipped in the templates dir, will be loaded from the module directory's template directory, eg $CHARM/hooks/charmhelpers/contrib/openstack/templates. This allows us to ship common templates (haproxy, apache) with the helpers.

Context generators

Context generators are used to generate template contexts during hook execution. Doing so may require inspecting service relations, charm config, etc. When registered, a config file is associated with a list of generators. When a template is rendered and written, all context generates are called in a chain to generate the context dictionary passed to the jinja2 template. See context.py for more info.

complete_contexts()
Returns a list of context interfaces that yield a complete context.

register(*config_file*, *contexts*)
Register a config file with a list of context generators to be called during rendering.

render(*config_file*)

set_release(*openstack_release*)
Resets the template environment and generates a new template loader based on a the new openstack release.

write(*config_file*)
Write a single config file, raises if config file is not registered.

write_all()
Write out all registered config files.

class charmhelpers.contrib.openstack.templating.**OSConfigTemplate**(*config_file*, *contexts*)

Bases: object

Associates a config file template with a list of context generators. Responsible for constructing a template context based on those generators.

complete_contexts()
Return a list of interfaces that have atisfied contexts.

context()

charmhelpers.contrib.openstack.templating.**get_loader**(*templates_dir*, *os_release*)
Create a jinja2.ChoiceLoader containing template dirs up to and including os_release. If directory template directory is missing at templates_dir, it will be omitted from the loader. templates_dir is added to the bottom of the search list as a base loading dir.

A charm may also ship a templates dir with this module and it will be appended to the bottom of the search list, eg:

```
hooks/charmhelpers/contrib/openstack/templates
```

Parameters

- **(str)** (os_release) – Base template directory containing release sub-directories.
- **(str)** – OpenStack release codename to construct template loader.

Returns jinja2.ChoiceLoader constructed with a list of jinja2.FilesystemLoaders, ordered in descending order by OpenStack release.

charmhelpers.contrib.openstack.utils module

charmhelpers.contrib.openstack.utils.**clean_storage**(*block_device*)

Ensures a block device is clean. That is:

- unmounted
- any lvm volume groups are deactivated
- any lvm physical device signatures removed
- partition table wiped

Parameters **block_device** – str: Full path to block device to clean.

charmhelpers.contrib.openstack.utils.**config_value_changed**(*option*)
Determine if config value changed since last call to this function.

charmhelpers.contrib.openstack.utils.**configure_installation_source**(*rel*)
Configure apt installation source.

charmhelpers.contrib.openstack.utils.**ensure_block_device**(*block_device*)
Confirm block_device, create as loopback if necessary.

Parameters **block_device** – str: Full path of block device to ensure.

Returns str: Full path of ensured block device.

charmhelpers.contrib.openstack.utils.**error_out**(*msg*)

charmhelpers.contrib.openstack.utils.**get_matchmaker_map**(*mm_file='/etc/oslo/matchmaker_ring.json'*)

charmhelpers.contrib.openstack.utils.**get_os_codename_install_source**(*src*)
Derive OpenStack release codename from a given installation source.

charmhelpers.contrib.openstack.utils.**get_os_codename_package**(*package*, *fatal=True*)
Derive OpenStack release codename from an installed package.

charmhelpers.contrib.openstack.utils.**get_os_codename_version**(*vers*)
Determine OpenStack codename from version number.

charmhelpers.contrib.openstack.utils.**get_os_version_codename**(*codename*)
Determine OpenStack version number from codename.

charmhelpers.contrib.openstack.utils.**get_os_version_install_source**(*src*)

charmhelpers.contrib.openstack.utils.**get_os_version_package**(*pkg*, *fatal=True*)
Derive OpenStack version number from an installed package.

charmhelpers.contrib.openstack.utils.**git_clone_and_install**(*projects_yaml*, *core_project*, *depth=1*)
Clone/install all specified OpenStack repositories.

The expected format of projects_yaml is:

repositories:

- **{name: keystone,** repository: 'git://git.openstack.org/openstack/keystone.git', branch: 'stable/icehouse'}
- **{name: requirements,** repository: 'git://git.openstack.org/openstack/requirements.git', branch: 'stable/icehouse'}

directory: /mnt/openstack-git http_proxy: squid-proxy-url https_proxy: squid-proxy-url

The directory, http_proxy, and https_proxy keys are optional.

charmhelpers.contrib.openstack.utils.**git_install_requested**()
Returns true if openstack-origin-git is specified.

charmhelpers.contrib.openstack.utils.**git_pip_venv_dir**(*projects_yaml*)
Return the pip virtualenv path.

charmhelpers.contrib.openstack.utils.**git_src_dir**(*projects_yaml*, *project*)
Return the directory where the specified project's source is located.

charmhelpers.contrib.openstack.utils.**git_yaml_value**(*projects_yaml*, *key*)
Return the value in projects_yaml for the specified key.

charmhelpers.contrib.openstack.utils.**import_key**(*keyid*)

charmhelpers.contrib.openstack.utils.**openstack_upgrade_available**(*package*)
Determines if an OpenStack upgrade is available from installation source, based on version of installed package.

Parameters **package** – str: Name of installed package.

Returns bool: : Returns True if configured installation source offers a newer version of package.

charmhelpers.contrib.openstack.utils.**os_release**(*package*, *base='essex'*)
Returns OpenStack release codename from a cached global. If the codename can not be determined from either an installed package or the installation source, the earliest release supported by the charm should be returned.

charmhelpers.contrib.openstack.utils.**os_requires_version**(*ostack_release*, *pkg*)
Decorator for hook to specify minimum supported release

charmhelpers.contrib.openstack.utils.**save_script_rc**(*script_path='scripts/scriptrc'*, ***env_vars*)
Write an rc file in the charm-delivered directory containing exported environment variables provided by env_vars. Any charm scripts run outside the juju hook environment can source this scriptrc to obtain updated config information necessary to perform health checks or service changes.

charmhelpers.contrib.openstack.utils.**sync_db_with_multi_ipv6_addresses**(*database*, *database_user*, *relation_prefix=None*)

charmhelpers.contrib.peerstorage package

charmhelpers.contrib.peerstorage.**leader_get**(*attribute=None*)
Wrapper to ensure that settings are migrated from the peer relation.

This is to support upgrading an environment that does not support Juju leadership election to one that does.

If a setting is not extant in the leader-get but is on the relation-get peer rel, it is migrated and marked as such so that it is not re-migrated.

charmhelpers.contrib.peerstorage.**peer_echo**(*includes=None*, *force=False*)
Echo filtered attributes back onto the same relation for storage.

This is a requirement to use the peerstorage module - it needs to be called from the peer relation's changed hook.

If Juju leader support exists this will be a noop unless force is True.

charmhelpers.contrib.peerstorage.**peer_retrieve**(*key*, *relation_name='cluster'*)
Retrieve a named key from peer relation *relation_name*.

charmhelpers.contrib.peerstorage.**peer_retrieve_by_prefix**(*prefix*, *relation_name='cluster'*, *delimiter='_'*, *inc_list=None*, *exc_list=None*)
Retrieve k/v pairs given a prefix and filter using {inc,exc}_list

charmhelpers.contrib.peerstorage.**peer_store**(*key*, *value*, *relation_name='cluster'*)
Store the key/value pair on the named peer relation *relation_name*.

charmhelpers.contrib.peerstorage.**peer_store_and_set**(*relation_id=None*, *peer_relation_name='cluster'*, *peer_store_fatal=False*, *relation_settings=None*, *delimiter='_'*, ***kwargs*)
Store passed-in arguments both in argument relation and in peer storage.

It functions like doing relation_set() and peer_store() at the same time, with the same data.

@param relation_id: the id of the relation to store the data on. Defaults to the current relation.

@param peer_store_fatal: Set to True, the function will raise an exception should the peer sotrage not be avialable.

charmhelpers.contrib.peerstorage.**relation_get**(*attribute=None*, *unit=None*, *rid=None*)
Attempt to use leader-get if supported in the current version of Juju, otherwise falls back on relation-get.

Note that we only attempt to use leader-get if the provided rid is a peer relation id or no relation id is provided (in which case we assume we are within the peer relation context).

charmhelpers.contrib.peerstorage.**relation_set**(*relation_id=None*, *relation_settings=None*, ***kwargs*)
Attempt to use leader-set if supported in the current version of Juju, otherwise falls back on relation-set.

Note that we only attempt to use leader-set if the provided relation_id is a peer relation id or no relation id is provided (in which case we assume we are within the peer relation context).

charmhelpers.contrib.python package

charmhelpers.contrib.python.debug module

charmhelpers.contrib.python.debug.**set_trace**(*addr='0.0.0.0'*, *port=4444*)
Set a trace point using the remote debugger

charmhelpers.contrib.python.packages module

charmhelpers.contrib.python.packages.**parse_options**(*given*, *available*)
Given a set of options, check if available

charmhelpers.contrib.python.packages.**pip_create_virtualenv**(*path=None*)
Create an isolated Python environment.

charmhelpers.contrib.python.packages.**pip_install**(*package*, *fatal=False*, *upgrade=False*, *venv=None*, ***options*)
Install a python package

charmhelpers.contrib.python.packages.**pip_install_requirements**(*requirements*, ***options*)
Install a requirements file

charmhelpers.contrib.python.packages.**pip_list**()
Returns the list of current python installed packages

charmhelpers.contrib.python.packages.**pip_uninstall**(*package*, ***options*)
Uninstall a python package

charmhelpers.contrib.python.rpdb module

Remote Python Debugger (pdb wrapper).

class charmhelpers.contrib.python.rpdb.**Rpdb**(*addr='127.0.0.1'*, *port=4444*)
Bases: pdb.Pdb

do_EOF(*arg*)
Stop all operation on continue.

do_c(*arg*)
Stop all operation on continue.

do_cont(*arg*)
Stop all operation on continue.

do_continue(*arg*)
Stop all operation on continue.

do_exit(*arg*)
Stop all operation on continue.

do_quit(*arg*)
Stop all operation on continue.

shutdown()
Revert stdin and stdout, close the socket.

charmhelpers.contrib.python.version module

charmhelpers.contrib.python.version.**current_version**()
Current system python version

charmhelpers.contrib.python.version.**current_version_string**()
Current system python version as string major.minor.micro

charmhelpers.contrib.saltstack package

Charm Helpers saltstack - declare the state of your machines.

This helper enables you to declare your machine state, rather than program it procedurally (and have to test each change to your procedures). Your install hook can be as simple as:

```
{{{
from charmhelpers.contrib.saltstack import (
    install_salt_support,
    update_machine_state,
)
```

```
def install():
    install_salt_support()
    update_machine_state('machine_states/dependencies.yaml')
    update_machine_state('machine_states/installed.yaml')
}}}
```

and won't need to change (nor will its tests) when you change the machine state.

It's using a python package called salt-minion which allows various formats for specifying resources, such as:

```
{{{
/srv/{{ basedir }}:
    file.directory:
        - group: ubunet
        - user: ubunet
        - require:
            - user: ubunet
        - recurse:
            - user
            - group

ubunet:
    group.present:
        - gid: 1500
    user.present:
        - uid: 1500
        - gid: 1500
        - createhome: False
        - require:
            - group: ubunet
}}}
```

The docs for all the different state definitions are at: http://docs.saltstack.com/ref/states/all/

TODO:

- Add test helpers which will ensure that machine state definitions are functionally (but not necessarily logically) correct (ie. getting salt to parse all state defs.
- Add a link to a public bootstrap charm example / blogpost.
- Find a way to obviate the need to use the grains['charm_dir'] syntax in templates.

charmhelpers.contrib.saltstack.**install_salt_support**(*from_ppa=True*)

Installs the salt-minion helper for machine state.

By default the salt-minion package is installed from the saltstack PPA. If from_ppa is False you must ensure that the salt-minion package is available in the apt cache.

charmhelpers.contrib.saltstack.**update_machine_state**(*state_path*)

Update the machine state using the provided state declaration.

charmhelpers.contrib.ssl package

charmhelpers.contrib.ssl.service module

class charmhelpers.contrib.ssl.service.**ServiceCA**(*name*, *ca_dir*, *cert_type='standard'*)
Bases: object

ca_cert

ca_conf

ca_key

create_certificate(*common_name*)

default_ca_expiry = '2190'

default_expiry = '730'

static get_ca(*type='standard'*)

get_ca_bundle()

get_certificate(*common_name*)

get_conf_variables()

get_or_create_cert(*common_name*)

classmethod get_service_cert(*type='standard'*)

init()

signing_conf

charmhelpers.contrib.ssl.**generate_selfsigned**(*keyfile*, *certfile*, *keysize='1024'*, *config=None*, *subject=None*, *cn=None*)

Generate selfsigned SSL keypair

You must provide one of the 3 optional arguments: config, subject or cn If more than one is provided the leftmost will be used

Arguments: keyfile – (required) full path to the keyfile to be created certfile – (required) full path to the certfile to be created keysize – (optional) SSL key length config – (optional) openssl configuration file subject – (optional) dictionary with SSL subject variables cn – (optional) cerfificate common name

Required keys in subject dict: cn – Common name (eq. FQDN)

Optional keys in subject dict country – Country Name (2 letter code) state – State or Province Name (full name) locality – Locality Name (eg, city) organization – Organization Name (eg, company) organizational_unit – Organizational Unit Name (eg, section) email – Email Address

charmhelpers.contrib.storage package

charmhelpers.contrib.storage.linux package

charmhelpers.contrib.storage.linux.ceph module

class charmhelpers.contrib.storage.linux.ceph.**CephBrokerRq**(*api_version=1*)
Bases: object

Ceph broker request.

Multiple operations can be added to a request and sent to the Ceph broker to be executed.

Request is json-encoded for sending over the wire.

The API is versioned and defaults to version 1.

add_op_create_pool(*name*, *replica_count=3*)

request

class charmhelpers.contrib.storage.linux.ceph.**CephBrokerRsp**(*encoded_rsp*)
Bases: object

Ceph broker response.

Response is json-decoded and contents provided as methods/properties.

The API is versioned and defaults to version 1.

exit_code

exit_msg

charmhelpers.contrib.storage.linux.ceph.**ceph_version**()
Retrieve the local version of ceph.

charmhelpers.contrib.storage.linux.ceph.**configure**(*service*, *key*, *auth*, *use_syslog*)
Perform basic configuration of Ceph.

charmhelpers.contrib.storage.linux.ceph.**copy_files**(*src*, *dst*, *symlinks=False*, *ignore=None*)
Copy files from src to dst.

charmhelpers.contrib.storage.linux.ceph.**create_key_file**(*service*, *key*)
Create a file containing key.

charmhelpers.contrib.storage.linux.ceph.**create_keyring**(*service*, *key*)
Create a new Ceph keyring containing key.

charmhelpers.contrib.storage.linux.ceph.**create_pool**(*service*, *name*, *replicas=3*)
Create a new RADOS pool.

charmhelpers.contrib.storage.linux.ceph.**create_rbd_image**(*service*, *pool*, *image*, *sizemb*)
Create a new RADOS block device.

charmhelpers.contrib.storage.linux.ceph.**delete_keyring**(*service*)
Delete an existing Ceph keyring.

charmhelpers.contrib.storage.linux.ceph.**delete_pool**(*service*, *name*)
Delete a RADOS pool from ceph.

charmhelpers.contrib.storage.linux.ceph.**ensure_ceph_keyring**(*service*, *user=None*, *group=None*)
Ensures a ceph keyring is created for a named service and optionally ensures user and group ownership.

Returns False if no ceph key is available in relation state.

charmhelpers.contrib.storage.linux.ceph.**ensure_ceph_storage**(*service*, *pool*, *rbd_img*, *sizemb*, *mount_point*, *blk_device*, *fstype*, *system_services=[]*, *replicas=3*)
NOTE: This function must only be called from a single service unit for the same rbd_img otherwise data loss will occur.

Ensures given pool and RBD image exists, is mapped to a block device, and the device is formatted and mounted at the given mount_point.

If formatting a device for the first time, data existing at mount_point will be migrated to the RBD device before being re-mounted.

All services listed in system_services will be stopped prior to data migration and restarted when complete.

charmhelpers.contrib.storage.linux.ceph.**filesystem_mounted**(*fs*)
Determine whether a filesytems is already mounted.

charmhelpers.contrib.storage.linux.ceph.**get_ceph_nodes**()
Query named relation 'ceph' to determine current nodes.

charmhelpers.contrib.storage.linux.ceph.**get_osds**(*service*)
Return a list of all Ceph Object Storage Daemons currently in the cluster.

charmhelpers.contrib.storage.linux.ceph.**image_mapped**(*name*)
Determine whether a RADOS block device is mapped locally.

charmhelpers.contrib.storage.linux.ceph.**install**()
Basic Ceph client installation.

charmhelpers.contrib.storage.linux.ceph.**make_filesystem**(*blk_device*, *fstype='ext4'*, *timeout=10*)
Make a new filesystem on the specified block device.

charmhelpers.contrib.storage.linux.ceph.**map_block_storage**(*service*, *pool*, *image*)
Map a RADOS block device for local use.

charmhelpers.contrib.storage.linux.ceph.**modprobe**(*module*)
Load a kernel module and configure for auto-load on reboot.

charmhelpers.contrib.storage.linux.ceph.**place_data_on_block_device**(*blk_device*, *data_src_dst*)
Migrate data in data_src_dst to blk_device and then remount.

charmhelpers.contrib.storage.linux.ceph.**pool_exists**(*service*, *name*)
Check to see if a RADOS pool already exists.

charmhelpers.contrib.storage.linux.ceph.**rbd_exists**(*service*, *pool*, *rbd_img*)
Check to see if a RADOS block device exists.

charmhelpers.contrib.storage.linux.loopback module

charmhelpers.contrib.storage.linux.loopback.**create_loopback**(*file_path*)
Create a loopback device for a given backing file.

Returns str: Full path to new loopback device (eg, /dev/loop0)

charmhelpers.contrib.storage.linux.loopback.**ensure_loopback_device**(*path*, *size*)
Ensure a loopback device exists for a given backing file path and size. If it a loopback device is not mapped to file, a new one will be created.

TODO: Confirm size of found loopback device.

Returns str: Full path to the ensured loopback device (eg, /dev/loop0)

charmhelpers.contrib.storage.linux.loopback.**loopback_devices**()
Parse through 'losetup -a' output to determine currently mapped loopback devices. Output is expected to look like:

/dev/loop0: [0807]:961814 (/tmp/my.img)

Returns dict: a dict mapping {loopback_dev: backing_file}

charmhelpers.contrib.storage.linux.lvm module

charmhelpers.contrib.storage.linux.lvm.**create_lvm_physical_volume**(*block_device*)
Initialize a block device as an LVM physical volume.

Parameters **block_device** – str: Full path of block device to initialize.

charmhelpers.contrib.storage.linux.lvm.**create_lvm_volume_group**(*volume_group*, *block_device*)
Create an LVM volume group backed by a given block device.

Assumes block device has already been initialized as an LVM PV.

Parameters **volume_group** – str: Name of volume group to create.

Block_device str: Full path of PV-initialized block device.

charmhelpers.contrib.storage.linux.lvm.**deactivate_lvm_volume_group**(*block_device*)
Deactivate any volume gruop associated with an LVM physical volume.

Parameters **block_device** – str: Full path to LVM physical volume

charmhelpers.contrib.storage.linux.lvm.**is_lvm_physical_volume**(*block_device*)
Determine whether a block device is initialized as an LVM PV.

Parameters **block_device** – str: Full path of block device to inspect.

Returns boolean: True if block device is a PV, False if not.

charmhelpers.contrib.storage.linux.lvm.**list_lvm_volume_group**(*block_device*)
List LVM volume group associated with a given block device.

Assumes block device is a valid LVM PV.

Parameters **block_device** – str: Full path of block device to inspect.

Returns str: Name of volume group associated with block device or None

charmhelpers.contrib.storage.linux.lvm.**remove_lvm_physical_volume**(*block_device*)
Remove LVM PV signatures from a given block device.

Parameters **block_device** – str: Full path of block device to scrub.

charmhelpers.contrib.storage.linux.utils module

charmhelpers.contrib.storage.linux.utils.**is_block_device**(*path*)
Confirm device at path is a valid block device node.

Returns boolean: True if path is a block device, False if not.

charmhelpers.contrib.storage.linux.utils.**is_device_mounted**(*device*)
Given a device path, return True if that device is mounted, and False if it isn't.

Parameters **device** – str: Full path of the device to check.

Returns boolean: True if the path represents a mounted device, False if it doesn't.

charmhelpers.contrib.storage.linux.utils.**zap_disk**(*block_device*)
Clear a block device of partition table. Relies on sgdisk, which is installed as pat of the 'gdisk' package in Ubuntu.

Parameters **block_device** – str: Full path of block device to clean.

charmhelpers.contrib.templating package

charmhelpers.contrib.templating.contexts module

A helper to create a yaml cache of config with namespaced relation data.

charmhelpers.contrib.templating.contexts.**dict_keys_without_hyphens**(*a_dict*)
Return the a new dict with underscores instead of hyphens in keys.

charmhelpers.contrib.templating.contexts.**juju_state_to_yaml**(*yaml_path*, *namespace_separator=':'*, *allow_hyphens_in_keys=True*, *mode=None*)
Update the juju config and state in a yaml file.

This includes any current relation-get data, and the charm directory.

This function was created for the ansible and saltstack support, as those libraries can use a yaml file to supply context to templates, but it may be useful generally to create and update an on-disk cache of all the config, including previous relation data.

By default, hyphens are allowed in keys as this is supported by yaml, but for tools like ansible, hyphens are not valid [1].

[1] http://www.ansibleworks.com/docs/playbooks_variables.html#what-makes-a-valid-variable-name

charmhelpers.contrib.templating.contexts.**update_relations**(*context*, *namespace_separator=':'*)
Update the context with the relation data.

charmhelpers.contrib.templating.pyformat module

Templating using standard Python str.format() method.

charmhelpers.contrib.templating.pyformat.**render**(*template*, *extra={}*, ***kwargs*)
Return the template rendered using Python's str.format().

charmhelpers.contrib.unison package

charmhelpers.contrib.unison.**collect_authed_hosts**(*peer_interface*)
Iterate through the units on peer interface to find all that have the calling host in its authorized hosts list

charmhelpers.contrib.unison.**create_private_key**(*user*, *priv_key_path*)

charmhelpers.contrib.unison.**create_public_key**(*user*, *priv_key_path*, *pub_key_path*)

charmhelpers.contrib.unison.**ensure_user**(*user*, *group=None*)

charmhelpers.contrib.unison.**get_homedir**(*user*)

charmhelpers.contrib.unison.**get_keypair**(*user*)

charmhelpers.contrib.unison.**run_as_user**(*user*, *cmd*, *gid=None*)

charmhelpers.contrib.unison.**ssh_authorized_peers**(*peer_interface*, *user*, *group=None*, *ensure_local_user=False*)
Main setup function, should be called from both peer -changed and -joined hooks with the same parameters.

charmhelpers.contrib.unison.**sync_path_to_host**(*path*, *host*, *user*, *verbose=False*, *cmd=None*, *gid=None*, *fatal=False*)

Sync path to an specific peer host

Propagates exception if operation fails and fatal=True.

charmhelpers.contrib.unison.**sync_to_peer**(*host*, *user*, *paths=None*, *verbose=False*, *cmd=None*, *gid=None*, *fatal=False*)

Sync paths to an specific peer host

Propagates exception if any operation fails and fatal=True.

charmhelpers.contrib.unison.**sync_to_peers**(*peer_interface*, *user*, *paths=None*, *verbose=False*, *cmd=None*, *gid=None*, *fatal=False*)

Sync all hosts to an specific path

The type of group is integer, it allows user has permissions to operate a directory have a different group id with the user id.

Propagates exception if any operation fails and fatal=True.

charmhelpers.contrib.unison.**write_authorized_keys**(*user*, *keys*)

charmhelpers.contrib.unison.**write_known_hosts**(*user*, *hosts*)

charmhelpers.fetch package

charmhelpers.fetch.archiveurl module

class charmhelpers.fetch.archiveurl.**ArchiveUrlFetchHandler**

Bases: *charmhelpers.fetch.BaseFetchHandler*

Handler to download archive files from arbitrary URLs.

Can fetch from http, https, ftp, and file URLs.

Can install either tarballs (.tar, .tgz, .tbz2, etc) or zip files.

Installs the contents of the archive in $CHARM_DIR/fetched/.

can_handle(*source*)

download(*source*, *dest*)

Download an archive file.

Parameters

- **source** (*str*) – URL pointing to an archive file.
- **dest** (*str*) – Local path location to download archive file to.

download_and_validate(*url*, *hashsum*, *validate='sha1'*)

install(*source*, *dest=None*, *checksum=None*, *hash_type='sha1'*)

Download and install an archive file, with optional checksum validation.

The checksum can also be given on the *source* URL's fragment. For example:

```
handler.install('http://example.com/file.tgz#sha1=deadbeef')
```

Parameters

- **source** (*str*) – URL pointing to an archive file.
- **dest** (*str*) – Local destination path to install to. If not given, installs to *$CHARM_DIR/archives/archive_file_name*.
- **checksum** (*str*) – If given, validate the archive file after download.
- **hash_type** (*str*) – Algorithm used to generate *checksum*. Can be any hash alrgorithm supported by hashlib, such as md5, sha1, sha256, sha512, etc.

charmhelpers.fetch.archiveurl.**splitpasswd**(*user*)
urllib.splitpasswd(), but six's support of this is missing

charmhelpers.fetch.archiveurl.**splituser**(*host*)
urllib.splituser(), but six's support of this seems broken

charmhelpers.fetch.bzrurl module

exception charmhelpers.fetch.**AptLockError**
Bases: exceptions.Exception

class charmhelpers.fetch.**BaseFetchHandler**
Bases: object

Base class for FetchHandler implementations in fetch plugins

base_url(*url*)
Return url without querystring or fragment

can_handle(*source*)
Returns True if the source can be handled. Otherwise returns a string explaining why it cannot

install(*source*)
Try to download and unpack the source. Return the path to the unpacked files or raise UnhandledSource.

parse_url(*url*)

exception charmhelpers.fetch.**SourceConfigError**
Bases: exceptions.Exception

exception charmhelpers.fetch.**UnhandledSource**
Bases: exceptions.Exception

charmhelpers.fetch.**add_source**(*source*, *key=None*)
Add a package source to this system.

@param source: a URL or sources.list entry, as supported by add-apt-repository(1). Examples:

```
ppa:charmers/example
deb https://stub:key@private.example.com/ubuntu trusty main
```

In addition: 'proposed:' may be used to enable the standard 'proposed' pocket for the release. 'cloud:' may be used to activate official cloud archive pockets, such as 'cloud:icehouse' 'distro' may be used as a noop

@param key: A key to be added to the system's APT keyring and used to verify the signatures on packages. Ideally, this should be an ASCII format GPG public key including the block headers. A GPG key id may also be used, but be aware that only insecure protocols are available to retrieve the actual public key from a public keyserver placing your Juju environment at risk. ppa and cloud archive keys are securely added automtically, so sould not be provided.

charmhelpers.fetch.**apt_cache**(*in_memory=True*)
Build and return an apt cache

charmhelpers.fetch.**apt_hold**(*packages*, *fatal=False*)

charmhelpers.fetch.**apt_install**(*packages*, *options=None*, *fatal=False*)
Install one or more packages

charmhelpers.fetch.**apt_mark**(*packages*, *mark*, *fatal=False*)
Flag one or more packages using apt-mark

charmhelpers.fetch.**apt_purge**(*packages*, *fatal=False*)
Purge one or more packages

charmhelpers.fetch.**apt_unhold**(*packages*, *fatal=False*)

charmhelpers.fetch.**apt_update**(*fatal=False*)
Update local apt cache

charmhelpers.fetch.**apt_upgrade**(*options=None*, *fatal=False*, *dist=False*)
Upgrade all packages

charmhelpers.fetch.**configure_sources**(*update=False*, *sources_var='install_sources'*, *keys_var='install_keys'*)
Configure multiple sources from charm configuration.

The lists are encoded as yaml fragments in the configuration. The frament needs to be included as a string. Sources and their corresponding keys are of the types supported by add_source().

Example config:

install_sources: |

- "ppa:foo"
- "http://example.com/repo precise main"

install_keys: |

- null
- "a1b2c3d4"

Note that 'null' (a.k.a. None) should not be quoted.

charmhelpers.fetch.**filter_installed_packages**(*packages*)
Returns a list of packages that require installation

charmhelpers.fetch.**install_from_config**(*config_var_name*)

charmhelpers.fetch.**install_remote**(*source*, **args*, ***kwargs*)
Install a file tree from a remote source

The specified source should be a url of the form: scheme://[host]/path[#[option=value][&...]]

Schemes supported are based on this modules submodules. Options supported are submodule-specific. Additional arguments are passed through to the submodule.

For example:

```
dest = install_remote('http://example.com/archive.tgz',
                      checksum='deadbeef',
                      hash_type='sha1')
```

This will download *archive.tgz*, validate it using SHA1 and, if the file is ok, extract it and return the directory in which it was extracted. If the checksum fails, it will raise *charmhelpers.core.host.ChecksumError*.

charmhelpers.fetch.**plugins**(*fetch_handlers=None*)

charmhelpers.payload package

charmhelpers.payload.archive module

exception charmhelpers.payload.archive.**ArchiveError**
Bases: exceptions.Exception

charmhelpers.payload.archive.**archive_dest_default**(*archive_name*)

charmhelpers.payload.archive.**extract**(*archive_name*, *destpath=None*)

charmhelpers.payload.archive.**extract_tarfile**(*archive_name*, *destpath*)
Unpack a tar archive, optionally compressed

charmhelpers.payload.archive.**extract_zipfile**(*archive_name*, *destpath*)
Unpack a zip file

charmhelpers.payload.archive.**get_archive_handler**(*archive_name*)

charmhelpers.payload.execd module

charmhelpers.payload.execd.**default_execd_dir**()

charmhelpers.payload.execd.**execd_module_paths**(*execd_dir=None*)
Generate a list of full paths to modules within execd_dir.

charmhelpers.payload.execd.**execd_preinstall**(*execd_dir=None*)
Run charm-pre-install for each module within execd_dir.

charmhelpers.payload.execd.**execd_run**(*command*, *execd_dir=None*, *die_on_error=False*, *stderr=None*)
Run command for each module within execd_dir which defines it.

charmhelpers.payload.execd.**execd_submodule_paths**(*command*, *execd_dir=None*)
Generate a list of full paths to the specified command within exec_dir.

Tools for working with files injected into a charm just before deployment.

charmhelpers.cli package

charmhelpers.cli.commands module

This module loads sub-modules into the python runtime so they can be discovered via the inspect module. In order to prevent flake8 from (rightfully) telling us these are unused modules, throw a ' # noqa' at the end of each import so that the warning is suppressed.

charmhelpers.cli.host module

charmhelpers.cli.host.**mounts**()
List mounts

class charmhelpers.cli.**CommandLine**
Bases: object

argument_parser = None

exit_code = 0

formatter = None

no_output(*decorated*)
Subcommand is not expected to return a value, so don't print a spurious None.

run()
Run cli, processing arguments and executing subcommands.

subcommand(*command_name=None*)
Decorate a function as a subcommand. Use its arguments as the command-line arguments

subcommand_builder(*command_name*, *description=None*)
Decorate a function that builds a subcommand. Builders should accept a single argument (the subparser instance) and return the function to be run as the command.

subparsers = None

test_command(*decorated*)
Subcommand is a boolean test function, so bool return values should be converted to a 0/1 exit code.

class charmhelpers.cli.**OutputFormatter**(*outfile=<open file '<stdout>', mode 'w'>*)
Bases: object

add_arguments(*argument_parser*)

csv(*output*)
Output data as excel-compatible CSV

format_output(*output*, *fmt='raw'*)

json(*output*)
Output data in JSON format

py(*output*)
Output data as a nicely-formatted python data structure

raw(*output*)
Output data as raw string (default)

supported_formats

tab(*output*)
Output data in excel-compatible tab-delimited format

yaml(*output*)
Output data in YAML format

charmhelpers.cli.**describe_arguments**(*func*)
Analyze a function's signature and return a data structure suitable for passing in as arguments to an argparse parser's add_argument() method.

charmhelpers.coordinator package

charmhelpers.coordinator module

The coordinator module allows you to use Juju's leadership feature to coordinate operations between units of a service.

Behavior is defined in subclasses of coordinator.BaseCoordinator. One implementation is provided (coordinator.Serial), which allows an operation to be run on a single unit at a time, on a first come, first served basis. You can trivially define more complex behavior by subclassing BaseCoordinator or Serial.

author Stuart Bishop <stuart.bishop@canonical.com>

Services Framework Usage

Ensure a peer relation is defined in metadata.yaml. Instantiate a BaseCoordinator subclass before invoking ServiceManager.manage(). Ensure that ServiceManager.manage() is wired up to the leader-elected, leader-settings-changed, peer relation-changed and peer relation-departed hooks in addition to any other hooks you need, or your service will deadlock.

Ensure calls to acquire() are guarded, so that locks are only requested when they are really needed (and thus hooks only triggered when necessary). Failing to do this and calling acquire() unconditionally will put your unit into a hook loop. Calls to granted() do not need to be guarded.

For example:

```
from charmhelpers.core import hookenv, services
from charmhelpers import coordinator

def maybe_restart(servicename):
    serial = coordinator.Serial()
    if needs_restart():
        serial.acquire('restart')
    if serial.granted('restart'):
        hookenv.service_restart(servicename)

services = [dict(service='servicename',
                 data_ready=[maybe_restart])]

if __name__ == '__main__':
    _ = coordinator.Serial()  # Must instantiate before manager.manage()
    manager = services.ServiceManager(services)
    manager.manage()
```

You can implement a similar pattern using a decorator. If the lock has not been granted, an attempt to acquire() it will be made if the guard function returns True. If the lock has been granted, the decorated function is run as normal:

```
from charmhelpers.core import hookenv, services
from charmhelpers import coordinator

serial = coordinator.Serial()  # Global, instatiated on module import.

def needs_restart():
    [ ... Introspect state. Return True if restart is needed ... ]

@serial.require('restart', needs_restart)
def maybe_restart(servicename):
    hookenv.service_restart(servicename)
```

```
services = [dict(service='servicename',
                 data_ready=[maybe_restart])]

if __name__ == '__main__':
    manager = services.ServiceManager(services)
    manager.manage()
```

Traditional Usage

Ensure a peer relationis defined in metadata.yaml.

If you are using charmhelpers.core.hookenv.Hooks, ensure that a BaseCoordinator subclass is instantiated before calling Hooks.execute.

If you are not using charmhelpers.core.hookenv.Hooks, ensure that a BaseCoordinator subclass is instantiated and its handle() method called at the start of all your hooks.

For example:

```
import sys
from charmhelpers.core import hookenv
from charmhelpers import coordinator

hooks = hookenv.Hooks()

def maybe_restart():
    serial = coordinator.Serial()
    if serial.granted('restart'):
        hookenv.service_restart('myservice')

@hooks.hook
def config_changed():
    update_config()
    serial = coordinator.Serial()
    if needs_restart():
        serial.acquire('restart'):
        maybe_restart()

# Cluster hooks must be wired up.
@hooks.hook('cluster-relation-changed', 'cluster-relation-departed')
def cluster_relation_changed():
    maybe_restart()

# Leader hooks must be wired up.
@hooks.hook('leader-elected', 'leader-settings-changed')
def leader_settings_changed():
    maybe_restart()

[ ... repeat for *all* other hooks you are using ... ]

if __name__ == '__main__':
    _ = coordinator.Serial()  # Must instantiate before execute()
    hooks.execute(sys.argv)
```

You can also use the require decorator. If the lock has not been granted, an attempt to acquire() it will be made if the guard function returns True. If the lock has been granted, the decorated function is run as normal:

```
from charmhelpers.core import hookenv

hooks = hookenv.Hooks()
serial = coordinator.Serial()  # Must instantiate before execute()

@require('restart', needs_restart)
def maybe_restart():
    hookenv.service_restart('myservice')

@hooks.hook('install', 'config-changed', 'upgrade-charm',
            # Peer and leader hooks must be wired up.
            'cluster-relation-changed', 'cluster-relation-departed',
            'leader-elected', 'leader-settings-changed')
def default_hook():
    [...]
    maybe_restart()

if __name__ == '__main__':
    hooks.execute()
```

Details

A simple API is provided similar to traditional locking APIs. A lock may be requested using the acquire() method, and the granted() method may be used do to check if a lock previously requested by acquire() has been granted. It doesn't matter how many times acquire() is called in a hook.

Locks are released at the end of the hook they are acquired in. This may be the current hook if the unit is leader and the lock is free. It is more likely a future hook (probably leader-settings-changed, possibly the peer relation-changed or departed hook, potentially any hook).

Whenever a charm needs to perform a coordinated action it will acquire() the lock and perform the action immediately if acquisition is successful. It will also need to perform the same action in every other hook if the lock has been granted.

Grubby Details

Why do you need to be able to perform the same action in every hook? If the unit is the leader, then it may be able to grant its own lock and perform the action immediately in the source hook. If the unit is the leader and cannot immediately grant the lock, then its only guaranteed chance of acquiring the lock is in the peer relation-joined, relation-changed or peer relation-departed hooks when another unit has released it (the only channel to communicate to the leader is the peer relation). If the unit is not the leader, then it is unlikely the lock is granted in the source hook (a previous hook must have also made the request for this to happen). A non-leader is notified about the lock via leader settings. These changes may be visible in any hook, even before the leader-settings-changed hook has been invoked. Or the requesting unit may be promoted to leader after making a request, in which case the lock may be granted in leader-elected or in a future peer relation-changed or relation-departed hook.

This could be simpler if leader-settings-changed was invoked on the leader. We could then never grant locks except in leader-settings-changed hooks giving one place for the operation to be performed. Unfortunately this is not the case with Juju 1.23 leadership.

But of course, this doesn't really matter to most people as most people seem to prefer the Services Framework or similar reset-the-world approaches, rather than the twisty maze of attempting to deduce what should be done based on what hook happens to be running (which always seems to evolve into reset-the-world anyway when the charm grows beyond the trivial).

I chose not to implement a callback model, where a callback was passed to acquire to be executed when the lock is granted, because the callback may become invalid between making the request and the lock being granted due to an upgrade-charm being run in the interim. And it would create restrictions, such no lambdas, callback defined at the top level of a module, etc. Still, we could implement it on top of what is here, eg. by adding a defer decorator that stores a pickle of itself to disk and have BaseCoordinator unpickle and execute them when the locks are granted.

class charmhelpers.coordinator.**BaseCoordinator**(*relation_key='coordinator'*, *peer_relation_name=None*)

Bases: object

acquire(*lock*)

Acquire the named lock, non-blocking.

The lock may be granted immediately, or in a future hook.

Returns True if the lock has been granted. The lock will be automatically released at the end of the hook in which it is granted.

Do not mindlessly call this method, as it triggers a cascade of hooks. For example, if you call acquire() every time in your peer relation-changed hook you will end up with an infinite loop of hooks. It should almost always be guarded by some condition.

grant(*lock*, *unit*)

Maybe grant the lock to a unit.

The decision to grant the lock or not is made for $lock by a corresponding method grant_$lock, which you may define in a subclass. If no such method is defined, the default_grant method is used. See Serial.default_grant() for details.

granted(*lock*)

Return True if a previously requested lock has been granted

grants = None

handle()

initialize()

msg(*msg*)

Emit a message. Override to customize log spam.

released(*unit*, *lock*, *timestamp*)

Called on the leader when it has released a lock.

By default, does nothing but log messages. Override if you need to perform additional housekeeping when a lock is released, for example recording timestamps.

relid = None

relname = None

request_timestamp(*lock*)

Return the timestamp of our outstanding request for lock, or None.

Returns a datetime.datetime() UTC timestamp, with no tzinfo attribute.

requested(*lock*)

Return True if we are in the queue for the lock

requests = None

require(*lock*, *guard_func*, **guard_args*, ***guard_kw*)

Decorate a function to be run only when a lock is acquired.

The lock is requested if the guard function returns True.

The decorated function is called if the lock has been granted.

class charmhelpers.coordinator.**Serial**(*relation_key='coordinator'*, *peer_relation_name=None*)
Bases: *charmhelpers.coordinator.BaseCoordinator*

default_grant(*lock*, *unit*, *granted*, *queue*)
Default logic to grant a lock to a unit. Unless overridden, only one unit may hold the lock and it will be granted to the earliest queued request.

To define custom logic for $lock, create a subclass and define a grant_$lock method.

unit is the unit name making the request.

granted is the set of units already granted the lock. It will never include *unit*. It may be empty.

queue is the list of units waiting for the lock, ordered by time of request. It will always include *unit*, but *unit* is not necessarily first.

Returns True if the lock should be granted to *unit*.

class charmhelpers.coordinator.**Singleton**
Bases: type

CHAPTER 5

Indices and tables

- genindex
- modindex

Python Module Index

C

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

R

S

T

U

V

W

Y

Z