Charms.Reactive Documentation

Release 1.3.0

Cory Johns

Oct 22, 2019

Contents

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

CHAPTER 1

Overview

Juju is an open source tool for modelling a connected set of applications in a way that allows for that model to be deployed repeatably and consistently across different clouds and substrates. Juju Charms implement the model for individual applications, their configuration, and the relations between them and other applications.

In order for the charm to know what actions to take, Juju informs it of life-cycle events in the form of hooks. These hooks inform the charm of things like the initial installation event, changes to charm config, attachment of storage, and adding and removing of units of related applications. Because managing distributed software is difficult and the exact action to take in response to a life-cycle event can depend on which events have happened in the past, charms.reactive represents a system for setting flags with semantic meaning to the charm and then driving behavior off of the combination of those flags.

The pattern is called "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, certain config values being set, etc. More importantly, you can react to not just individual 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:

```
from charms.reactive import set_flag, clear_flag, when
from charms.reactive.helpers import any_file_changed
from charmhelpers.core import templating, hookenv

@when('db.database.available', 'config.set.admin-pass')
def render_config(pgsql):
    templating.render('app-config.j2', '/etc/app.conf', {
        'db_conn': pgsql.connection_string(),
        'admin_pass': hookenv.config('admin-pass'),
    })
    if any_file_changed(['/etc/app.conf']):
        set_flag('myapp.restart')

@when('myapp.restart')
def restart_service():
```

(continues on next page)

(continued from previous page)

```
    hookenv.service_restart('myapp')
    clear_flag('myapp.restart')
```

CHAPTER 2

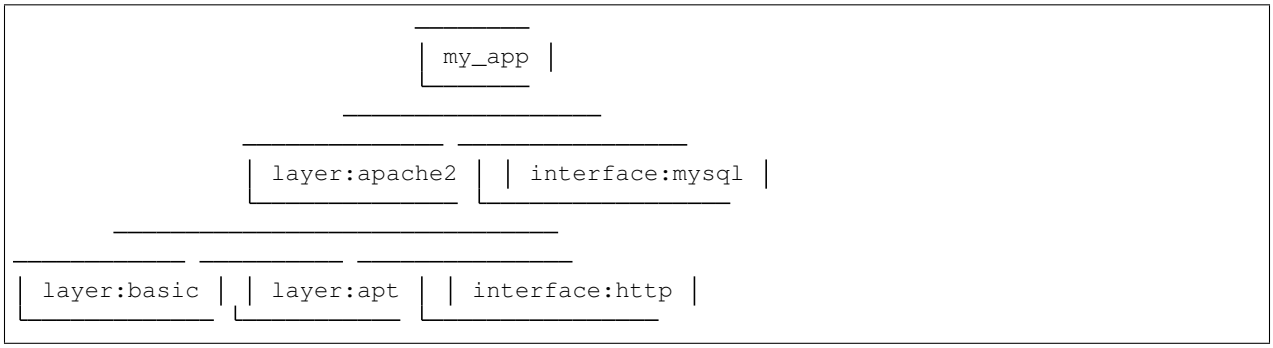
Table of Contents

2.1 Structure of a Reactive Charm

A reactive charm is built using layers, with the "top" layer being called the "charm layer." The charm layer would then reference other layers that it builds upon, which are generally thought of in two types: base layers, and interface layers. The charm indicates which layers it builds upon via its `layer.yaml`, which might look like this:

```
includes:
  - 'layer:apache'
  - 'interface:mysql'
options:
  basic:
    # apt packages required by charm code
    packages:
      - 'unzip'
```

This includes one base layer, `apache2`, and an interface layer, `mysql`. The `apache2` layer itself builds upon two other base layers and another interface layer, so the total hierarchy of the charm would look like this:



The `options` section in the `layer.yaml` allows the charm to set configuration for other layers. In this case, specifying to the `basic` layer that the charm needs the `unzip` package in order to function.

2.1.1 Charm Layer

The charm layer is what most charm authors will be writing, and allows the charm author to focus on just the information and code which is relevant to the charm itself. By including other layers, the charm layer can then rely on those layer to provide common behavior, using documented flags and method calls to communicate with those layers.

A charm layer consists, at a bare minimum, of the following files:

- `metadata.yaml`: This file contains information about the charm, such as the charm name, summary, description, maintainer, and what relations the charm supports.
- `layer.yaml`: This file indicates what other layers this charm builds upon.
- `reactive/<charm_name>.py`: This file, where `<charm_name>` is replaced by the name of the charm (using underscores in place of dashes), is the reactive entry point for the charm. It should contain or import files containing all of the handlers provided by this charm layer.

The charm layer should also contain a few additional files, though some may be optional depending on what features the charm supports:

- `README.md`: This file should document your charm in detail, and is required for the charm to be listed in the Charm Store.
- `copyright`: This file should document what copyright your charm is available under.
- `config.yaml`: For adding configuration options to the charm.
- `icon.svg`: For providing a nice icon for the charm.
- `actions.yaml` and `actions/<action-name>` scripts: For supporting actions in the charm.
- `metrics.yaml`: For collecting metrics about the deployment.

An example tree for a charm layer might thus look like this:

```
.
├── README.md
├── metadata.yaml
├── icon.svg
├── config.yaml
├── layer.yaml
├── reactive/
│   └── my_app.py
├── actions.yaml
├── actions/
│   └── do-something
└── copyright
```

2.1.2 Base Layers

Base layers provide functionality that is common across several charms. These layers should provide a set of handlers in `reactive/<layer_name>.py` which will set additional flags that will drive behavior in the charm layer. They may also include a Python module in `lib/charms/layer/<layer_name>.py` which can be imported from the charm layer to provide functions or classes to be used by the charm layer.

Base layers are otherwise identical to charm layers, and can provide things such as actions, config options, metrics, etc. for the charm layer. For example, a base layer might provide an action script, as well as the corresponding defition in the `actions.yaml` file. The `actions.yaml` file from the charm layer will then be merged onto the one provided by the base layer, and both sets of actions will be available.

layer:basic is a useful base layer:

- It provides hooks for other layers to react to such as `install`, `config-changed`, `upgrade-charm`, and `update-status`.
- It provides a *set of useful flags to react to changing config*.
- You can tell it to install *python* and *apt* dependencies of your handlers.

2.1.3 Interface Layers

Interface layers encapsulate the communication protocol over a Juju interface when two applications are related together. These layers will react to applications being related to the charm, and will handle the transfer of data to and from the units of the related application. This ensures that all charms using that interface protocol can effectively communicate with one another.

As with base layers, an interface layer will provide a set of flags to inform the charm layer of the signficant points in the relationship conversation. The interface layer will also provide a class with well-documented methods to use to interact with that relation. Instances of these classes will be automatically created by the framework.

More information about interface layers can be found in the docs.

2.2 Automatic Flags

The reactive framework will automatically set some flags for your charm, based on lifecycle events from Juju. These flags can inform your charm of things such as upgrades, config changes, relation activity, etc.

With a few exceptions, noted below, these flags will be set by the framework but it is up to your charm to clear them, if need be. To avoid conflicts between layers, it is recommended that only the top-level charm layer (or interface layer, in the case of endpoint flags) use any of the automatic flags directly; any base layer should instead use `register_trigger()` to "wrap" the automatic flag with a layer-specific flag that can be safely used within that layer.

The flags that are set by the framework are:

`upgrade.series.in-progress`	This is set when the operator is about to start an OS upgrade, and removed after the operator has completed the upgrade. See *OS Series Upgrades* for more information.
`config.changed`	This is set when any config option has changed.[1]
`config.changed.{option_name}`	This is set for each config option that has changed.[1]
`config.set.{option_name}`	This is set for each config option whose value is not `None`, `False`, or an empty string.[1]
`config.default.{option_name}`	This is set for each config option whose value was changed from its default.[1]
`leadership.is_leader`	This is set when the unit is the leader. The unit will remain the leader for the remainder of the hook, but may not be leader in future hooks.[2]
`leadership.changed`	This is set when any leadership setting has changed.[2]
`leadership.changed.{setting_name}`	This is set for each leadership setting that has changed.[2]
`leadership.set.{setting_name}`	This is set for each leadership setting that has been to set to a value other than `None`.[2]
`endpoint.{endpoint_name}.joined`	This is set when a relation is joined on an endpoint.[3]
`endpoint.{endpoint_name}.changed`	This is set when relation data has changed.[3]
`endpoint.{endpoint_name}.changed.{field}`	This is set for each field of relation data which has changed.[3]
`endpoint.{endpoint_name}.departed`	This is set when a unit leaves a relation.[3]

The `config.*` flags are currently managed by *the base layer* and are automatically cleared a the end of the hook context in which they were set. However, this is expected to change in the future, with the flags being set by this library instead and the automatic clearing behavior changed or removed.

The `leadership.*` flags are currently managed by the leadership layer and the `leadership.changed*` flags are automatically cleared at the end of the hook context in which they were set. If this layer is not included by the charm or one of its base layers, these flags will not be set. However, this is expected to change in the future, with the flags being managed by this library instead and the automatic clearing behavior changed or removed.

See `Endpoint` for more information on the `endpoint.{endpoint_name}.*` flags. The `endpoint.{endpoint_name}.joined` flag is automatically cleared when appropriate.

2.3 The base layer: layer-basic

This is the base layer for all reactive Charms. It provides all of the standard Juju hooks and starts the reactive framework when these hooks get executed. It also bootstraps the charm-helpers and `charms.reactive` libraries, and all of their dependencies for use by the Charm. Check out the code for the basic layer on Github.

1
2
3

2.3.1 Usage

To create a charm layer using this base layer, you need only include it in a `layer.yaml` file.

```
includes: ['layer:basic']
```

This will fetch this layer from interfaces.juju.solutions and incorporate it into your charm layer. You can then add handlers under the `reactive/` directory. Note that **any** file under `reactive/` will be expected to contain handlers, whether as Python decorated functions or executables using the external handler protocol.

2.3.2 Hooks

This layer provides a `hook.template` which starts the reactive framework when the hook is run. During the build process, this template is used to implement all of the following hooks, as well as any necessary relation and storage hooks:

- `config-changed`
- `install`
- `leader-elected`
- `leader-settings-changed`
- `start`
- `stop`
- `upgrade-charm`
- `update-status`
- `pre-series-upgrade`
- `post-series-upgrade`

A layer can implement other hooks (e.g., `metrics`) by putting them in the `hooks` directory.

Note: Because `update-status` is invoked every 5 minutes, you should take care to ensure that your reactive handlers only invoke expensive operations when absolutely necessary. It is recommended that you use helpers like `data_changed` to ensure that handlers run only when necessary.

Note: The charm snap has been the supported way to build charms for a long time, but there is still an old version of charm-tools available via apt on some systems. This old version doesn't properly handle the `hook.template` file, leading to missing hooks when charms are built. If you encounter this issue, please make sure you have the snap installed and remove any copies of the `charm` or `charm-tools` apt packages.

2.3.3 Reactive flags for Charm config

This layer will set the following flags:

- **`config.changed`** Any config option has changed from its previous value. This flag is cleared automatically at the end of each hook invocation.
- **`config.changed.<option>`** A specific config option has changed. **`<option>`** will be replaced by the config option name from `config.yaml`. This flag is cleared automatically at the end of each hook invocation.

- **config.set.<option>** A specific config option has a True or non-empty value set. **<option>** will be replaced by the config option name from `config.yaml`. This flag is cleared automatically at the end of each hook invocation.
- **config.default.<option>** A specific config option is set to its default value. **<option>** will be replaced by the config option name from `config.yaml`. This flag is cleared automatically at the end of each hook invocation.

An example using the config flags would be:

```
@when('config.changed.my-opt')
def my_opt_changed():
    update_config()
    restart_service()
```

2.3.4 Layer Configuration

This layer supports the following options, which can be set in `layer.yaml`:

- **packages** A list of system packages to be installed before the reactive handlers are invoked.

  Note: The `packages` layer option is intended for **charm** dependencies only. That is, for libraries and applications that the charm code itself needs to do its job of deploying and configuring the payload. If the payload (the application you're deploying) itself has dependencies, those should be handled separately, by your Charm using for example the Apt layer

  ---

- **python_packages** A list of Python packages to be installed after the wheelhouse but before the reactive handlers are invoked.

  Note: The `packages` layer option is intended for **charm** dependencies only. That is, for libraries and applications that the charm code itself needs to do its job of deploying and configuring the payload. If the payload (the application you're deploying) itself has dependencies, those should be handled separately, by your Charm using for example the Apt layer

  ---

- **use_venv** If set to true, the charm dependencies from the various layers' `wheelhouse.txt` files will be installed in a Python virtualenv located at `$JUJU_CHARM_DIR/../.venv`. This keeps charm dependencies from conflicting with payload dependencies, but you must take care to preserve the environment and interpreter if using `execl` or `subprocess`.
- **include_system_packages** If set to true and using a venv, include the `--system-site-packages` options to make system Python libraries visible within the venv.

An example `layer.yaml` using these options might be:

```
includes: ['layer:basic']
options:
  basic:
    packages: ['git']
    use_venv: true
    include_system_packages: true
```

2.3.5 Wheelhouse.txt for Charm Python dependencies

`layer-basic` provides two methods to install dependencies of your charm code: `wheelhouse.txt` for python dependencies and the `packages` layer option for apt dependencies.

Each layer can include a `wheelhouse.txt` file with Python requirement lines. *The format of this file is the same as pip's `requirements.txt` file.* For example, this layer's `wheelhouse.txt` includes:

```
pip>=7.0.0,<8.0.0
charmhelpers>=0.4.0,<1.0.0
charms.reactive>=0.1.0,<2.0.0
```

All of these dependencies from each layer will be fetched (and updated) at build time and will be automatically installed by this base layer **before any reactive handlers are run.**

See PyPI for packages under the `charms.` namespace which might be useful for your charm. See the `packages` layer option of this layer for installing `apt` dependencies of your Charm code.

Note: The `wheelhouse.yaml` are intended for **charm** dependencies only. That is, for libraries and applications that the charm code itself needs to do its job of deploying and configuring the payload. If the payload (the application you're deploying) itself has dependencies, those should be handled separately.

2.3.6 Exec.d Support

It is often necessary to configure and reconfigure machines after provisioning, but before attempting to run the charm. Common examples are specialized network configuration, enabling of custom hardware, non-standard disk partitioning and filesystems, adding secrets and keys required for using a secured network.

The reactive framework's base layer invokes this mechanism as early as possible, before any network access is made or dependencies unpacked or non-standard modules imported (including the charms.reactive framework itself).

Operators needing to use this functionality may branch a charm and create an exec.d directory in it. The exec.d directory in turn contains one or more subdirectories, each of which contains an executable called charm-pre-install and any other required resources. The charm-pre-install executables are run, and if successful, state saved so they will not be run again.

```
$JUJU_CHARM_DIR/exec.d/mynamespace/charm-pre-install
```

An alternative to branching a charm is to compose a new charm that contains the exec.d directory, using the original charm as a layer,

A charm author could also abuse this mechanism to modify the charm environment in unusual ways, but for most purposes it is saner to use `charmhelpers.core.hookenv.atstart()`.

2.3.7 General layer info

Layer Namespace

Each layer has a reserved section in the `charms.layer.` Python package namespace, which it can populate by including a `lib/charms/layer/<layer-name>.py` file or by placing files under `lib/charms/layer/<layer-name>/`. (If the layer name includes hyphens, replace them with underscores.) These can be helpers that the layer uses internally, or it can expose classes or functions to be used by other layers to interact with that layer.

For example, a layer named `foo` could include a `lib/charms/layer/foo.py` file with some helper functions that other layers could access using:

```
from charms.layer.foo import my_helper
```

Layer Options

Any layer can define options in its `layer.yaml`. Those options can then be set by other layers to change the behavior of your layer. The options are defined using jsonschema, which is the same way that action paramters are defined.

For example, the `foo` layer could include the following option definitons:

```
includes: ['layer:basic']
defines:  # define some options for this layer (the layer "foo")
  enable-bar:  # define an "enable-bar" option for this layer
    description: If true, enable support for "bar".
    type: boolean
    default: false
```

A layer using `foo` could then set it:

```
includes: ['layer:foo']
options:
  foo:  # setting options for the "foo" layer
    enable-bar: true  # set the "enable-bar" option to true
```

The `foo` layer can then use `charms.layer.options` to get the value for each defined option. For example:

```
from charms import layer

@when('flag')
def do_thing():
  # check the value of the "enable-bar" option for the "foo" layer
  if layer.options.get('foo', 'enable-bar'):
      hookenv.log("Bar is enabled")

  # or get all of the options for the "foo" layer as a dict
  foo_opts = layer.options.get('foo')
```

Note: *charms.layer.options* is, itself, implemented as a layer named [*layer:options*][layer-options] and is automatically included when a layer includes [*layer:basic*][layer-basic]

You can also access layer options in other handlers, such as Bash, using the command-line interface:

```
. charms.reactive.sh

@when 'flag'
function do_thing() {
    if layer_option foo enable-bar; then
        juju-log "Bar is enabled"
        juju-log "bar-value is: $(layer_option foo bar-value)"
    fi
}

reactive_handler_main
```

Note that options of type `boolean` will set the exit code, while other types will be printed out.

2.4 Reactive with Bash or Other Languages

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, which allow you to write handlers using a decorator-like syntax similar to Python handlers. For example:

```
#!/bin/bash
source charms.reactive.sh

@when 'db.database.available' 'admin-pass'
function render_config() {
    db_conn=$(relation_call --flag 'db.database.available' connection_string)
    admin_pass=$(config-get 'admin-pass')
    charms.reactive render_template 'app-config.j2' '/etc/app.conf'
}

@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
```

2.5 Frequently Asked Questions

2.5.1 How do I run and debug reactive charm?

You run a reactive charm by running a hook in the `hooks/` directory. That hook will start the reactive framework and initiate the "cascade of flags".

The hook files in the `hooks/` directory are created by `layer:basic` and by `charm build`. Make sure to include `layer:basic` in your *layer.yaml* file if the hook files aren't present in the *hooks/* directory.

You can find more information about debugging reactive charms in the Juju docs.

Note: Changes to flags are reset when a handler crashes. Changes to flags happen immediately, but they are only persisted at the end of a complete and successful run of the reactive framework. All unpersisted changes are discarded when a hook crashes.

2.5.2 Why doesn't my Charm do anything? Why are there no hooks in the `hooks` directory?

You probably forgot to include *layer-basic* in your `layer.yaml` file. This layer creates the hook files so that the reactive framework starts when a hook runs.

2.5.3 How can I react to configuration changes?

The base layer provides *a set of easy flags* to react to configuration changes. These flags will be automatically managed when you include `layer:basic` in your `layer.yaml` file.

2.5.4 How to remove a flag immediately when a config changes?

You can use `triggers` for this, see *Reactive Triggers* for more info.

Example: clear the flag `apt.sources_configured` immediately when the `install_sources` config option changes.

```
register_trigger(when='config.changed.install_sources',
                 clear_flag='apt.sources_configured')
```

2.5.5 How to run a handler even if the flag it reacts to has since been cleared?

Take the following case:

```
@when('service.stopped')
def restart_service():
    restart_my_service()
    clear_flag('service.stopped')

@when_all('service.stopped',
          'endpoint.clients.connected')
def notify_related_units():
    clients = from_flag('endpoint.clients.connected')
    clients.notify_service_stopped()
```

The `notify_related_units` handler will never get invoked because the `restart_handler` will get invoked first and it removes the `service.stopped` state. If this is not the desired behavior, if you need to notify the clients even when the service has been restarted by another handler, then you can use a `trigger` to create a new state specifically for the `notify_related_units` handler:

```
register_trigger(when='service.stopped',
                 set_flag='clients.need_notification')

@when('service.stopped')
def restart_service():
    restart_my_service()
    clear_flag('service.stopped')

@when_all('clients.need_notification',
          'endpoint.clients.connected')
def notify_related_units():
    clients = from_flag('endpoint.clients.connected')
```

(continues on next page)

(continued from previous page)

```
    clients.notify_service_stopped()
    clear_flag('clients.need_notification')
```

See *Reactive Triggers* for more information.

2.6 Patterns

When creating charms, layers, or interface layers with reactive, some common patterns can come up. This page documents some of them.

2.6.1 Request / Response

A common pattern in interface layers is for one charm to generate individual requests, which then need to be paired with a specific response from the other charm. This can be tricky to accomplish with relation data, due to the fact that a given unit can only publish its data for the entire relation, rather than a specific remote unit, plus the fact that a given unit may want to submit multiple requests. The framework provides some base classes to assist with this pattern.

An interface layer would first define a request and response type, which inherit from `BaseRequest` and `BaseResponse` respectively, and which each define a set of `Field` attributes to hold the data for the request and response. Each field can provide a description, for documentation purposes.

Note: The request class must explicitly point to the class which implements the associated response, via the `RESPONSE_CLASS` attribute, so that the correct class can be used when creating responses to requests.

For example:

```
from charms.reactive import BaseRequest, BaseResponse, Field


class CertResponse(BaseResponse):
    signed_cert = Field(description="""
                        The text of the public certificate signed by the CA.
                        """)

class CertRequest(BaseRequest):
    RESPONSE_CLASS = CertResponse  # point to response implementation

    csr_data = Field(description="""
                     The text of the generated Certificate Signing Request.
                     """)
```

Then, the interface layer would define endpoint classes which inherit from `RequesterEndpoint` and `ResponderEndpoint` rather than directly from `Endpoint`. These classes would point to the appropriate request implementation to use via the `REQUEST_CLASS` attribute, and they would inherit various properties and methods for interacting with the requests and responses (although it may make sense for them to wrap some of these with methods of their own more specialized for their specific needs).

For example:

```
from charms.reactive import RequesterEndpoint, ResponderEndpoint
```

(continues on next page)

(continued from previous page)

```
class CertRequester(RequesterEndpoint):
    REQUEST_CLASS = CertRequest  # point to request implementation

    @property
    def related_cas(self):
        """
        A list of the related CAs which can sign certs.
        """
        return self.relations

    def send_csr(self, related_ca, csr_data):
        """
        Send a CSR to the specified related CA.

        Returns the created request.
        """
        return CertRequest.create(relation=related_ca,
                                  csr_data=csr_data)


class CertResponder(ResponderEndpoint):
    REQUEST_CLASS = CertRequest  # point to request implementation

    # no additional implementation needed beyond the inherited properties / methods
```

Charms using this interface layer could then submit requests and provide responses.

For example, a client charm might look something like:

```
@when('endpoint.certs.joined')
@when_not('charm.cert_requested')
def request_cert():
    cert_provider = endpoint_from_name('certs')
    if len(cert_provider.related_cas) == 0:
        return
    if len(cert_provider.related_cas) > 1:
        status.blocked('Too many CAs')
        return
    ca = cert_provider.related_cas[0]
    csr_data = generate_csr()
    request = cert_provider.send_csr(ca, csr_data)
    unitdata.kv().set('current_cert_request', request.request_id)  # for reissues
    set_flag('charm.cert_requested')

@when('endpoint.certs.all_responses')
def write_cert():
    cert_provider = endpoint_from_name('certs')
    current_request = unitdata.kv().get('current_cert_request')  # handle reissues
    response = cert_provider.response_by_field(request_id=current_request)
    CERT_PATH.write_text(response.signed_cert)
```

And the corresponding provider charm might look something like:

```
@when('endpoint.cert_clients.new_requests')
def sign_certs():
    cert_clients = endpoint_from_name('cert_clients')
```

(continues on next page)

(continued from previous page)

```
    for request in cert_clients.new_requests:
        signed_cert = sign_cert(request.csr_data)
        request.respond(signed_cert=signed_cert)
```

2.7 Reactive API Documentation

BaseRequest	Base class for requests using the request / response pattern.
BaseResponse	Base class for responses using the request / response pattern.
Endpoint	New base class for creating interface layers.
Field	Defines a Field property for a Request or Response object.
RelationBase	A base class for relation implementations.
RequesterEndpoint	Base class for Endpoints that create requests in the request / response pattern.
ResponderEndpoint	Base class for Endpoints that respond to requests in the request / response pattern.
all_flags_set	Assert that all desired_flags are set
any_file_changed	Check if any of the given files have changed since the last time this was called.
any_flags_set	Assert that any of the desired_flags are set
clear_flag	Clear / deactivate a flag.
collect_metrics	Register the decorated function to run for the collect_metrics hook.
data_changed	Check if the given set of data has changed since the previous call.
endpoint_from_flag	The object used for interacting with relations tied to a flag, or None.
endpoint_from_name	The object used for interacting with the named relations, or None.
get_flags	Return a list of all flags which are set.
hook	Register the decorated function to run when the current hook matches any of the hook_patterns.
is_data_changed	Check if the given set of data has changed since the last time *data_changed* was called.
is_flag_set	Assert that a flag is set
main	This is the main entry point for the reactive framework.
meter_status_changed	Register the decorated function to run when a meter status change has been detected.
not_unless	Assert that the decorated function can only be called if the desired_flags are active.
register_trigger	Register a trigger to set or clear a flag when a given flag is set.
scopes	These are the recommended scope values for relation implementations.
set_flag	Set the given flag as active.

Continued on next page

Table 1 – continued from previous page

`toggle_flag`	Helper that calls either `set_flag()` or `clear_flag()`, depending on the value of *should_set*.
`when`	Alias for `when_all()`.
`when_all`	Register the decorated function to run when all of `desired_flags` are active.
`when_any`	Register the decorated function to run when any of `desired_flags` are active.
`when_file_changed`	Register the decorated function to run when one or more files have changed.
`when_none`	Register the decorated function to run when none of `desired_flags` are active.
`when_not`	Alias for `when_none()`.
`when_not_all`	Register the decorated function to run when one or more of the `desired_flags` are not active.

2.8 Internals and Advanced

2.8.1 Discovery and Dispatch of Reactive Handlers

Reactive handlers are loaded from any file under the `reactive` directory, as well as any interface layers you are using. Handlers can be decorated blocks in Python, or executable files following the *`ExternalHandler`* protocol. Handlers can be split amongst several files, which is particularly useful for layers, as each layer can define its own file containing handlers so as not to conflict with files from other layers.

Once all of the handlers are loaded, all `@hook` handlers will be executed, in a non-determined order. In general, only one layer or relation stub should have a matching `@hook` block for each hook, which should then set appropriate semantically meaningful flags that the other layers can react to. If there are multiple handlers that match for a given hook, there is no guarantee which order they will execute in. Hook handlers should live in the layer that is most appropriate for them. The base or runtime layer will probably handle the install and upgrade hooks, relation stubs will handle all of the relation hooks, etc.

After all of the hook handlers have run, other handlers are dispatched based on the flags set by the hook handlers and any flags from previous runs. Various hook invocations can each set their appropriate flags, and the reactive handlers will be triggered when all of the appropriate flags are set, regardless of when and in which order they are each set.

All handlers are tested and matching handlers queued before invoking the first handler. Thus, flags set by a handler will not trigger new matching handlers until after all of the current set of matching handlers are done. This allows you to ensure some ordering of otherwise non-determined handler invocation by chaining flags (e.g., handler_A sets flag_B, which triggers handler_B which then sets flag_C, which triggers handler_C, and so on).

Note, however, that removing a flag causes the remaining set of matched handlers to be re-tested. This ensures that a handler is never invoked when the flag is no longer active.

2.8.2 Reactive Triggers

Coupling Flags with Triggers

In general, it is best to be explicit about setting or clearing a flag. This makes the code more maintainable and easier to follow and reason about. However, rarely, due to the fact that handlers for a given flag are independent and thus there are no guarantees about the order in which they may execute, it is sometimes necessary to enforce that two flags must be set at the same time or that one must be cleared if the other is set.

As an example of when this might be necessary, consider a charm which provides two config values, one that determines the location from which resources should be fetched, with a default location provided by the charm, and another which indicates that a particular feature be installed and enabled. If the charm is deployed and fetches all of the resources, it might set a flag that indicates that all resources are available and any installation can proceed. However, if both resource location and feature flag config options are changed at the same time, the handlers might be invoked in an order that causes the feature installation to happen before the resource change has been observed, leading to the feature using the wrong resource. This problem is particularly intractable if the layer managing the resource location and readiness options is different than the layer managing the feature option, such as with the apt layer.

Triggers provide a mechanism for a flag to indicate that when a particular flag is set, another specific flag should be either set or cleared. To use a trigger, you simply have to register it, which can be done from inside a handler, or at the top level of your handlers file:

```
from charms.reactive.flags import register_trigger
from charms.reactive.flags import set_flag
from charms.reactive.decorators import when


register_trigger(when='flag_a',
                 set_flag='flag_b')


@when('flag_b')
def handler():
    do_something()
    register_trigger(when='flag_a',
                     clear_flag='flag_c')
    set_flag('flag_c')
```

When a trigger is registered, then as soon as the flag given by `when` is set, the other flag is set or cleared at the same time. Thus, there is no chance that another handler will run in between.

Keep in mind that since triggers are implicit, they should be used sparingly. Most use cases can be better modeled by explicitly setting and clearing flags.

Example uses of triggers

Remove flags immediately when config changes

In the `apt` layer, the `install_sources` config option specifies which repositories and ppa's to use for installing a package, so these need to be added before installing any package. This is easy to do with flags: you create a handler that adds the sources and then sets the flag `apt.sources_configured`. The handler that installs the packages reacts to that flag with `@when('apt.sources_configured')`. This works perfectly the first time but what happens if the `install_sources` config option gets changed after they are first configured? Then the `apt.sources_configured` flag needs to be cleared immediately before any new packages are installed. This is where triggers come in: You create a trigger that unsets the *apt.sources_configured* flag when the *install_sources* config changes.

```
register_trigger(when='config.changed.install_sources',
                 clear_flag='apt.sources_configured')


@when_not('apt.sources_configured')
def sources_handler():
    configure_sources()
    set_state('apt.sources_configured')
```

(continues on next page)

(continued from previous page)

```
@when_all('apt.needs_update',
          'apt.sources_configured')
def update():
    charms.apt.update()
    clear_flag('apt.sources_configured')


@when('apt.queued_installs')
@when_not('apt.needs_update')
def install_queued():
    charms.apt.install_queued()
    clear_flag('apt.queued_installs')


@when_not('apt.queued_installs')
def ensure_package_status():
   charms.apt.ensure_package_status()
```

2.8.3 OS Series Upgrades

Upgrades of the operating system's series, or version, are difficult to automate in a general fashion, so most of the work is done manually by the operator and the role that the charm plays is somewhat limited. However, the charm does need to ensure that during the upgrade, all of the application services on the unit are disabled and stopped so that nothing runs while the operator is making changes that could break the application, even if the machine is rebooted one or more times.

When the operator is about to initiate an OS upgrade, they will run:

```
juju upgrade-series <machine> prepare <target-series>
```

The framework will then set the `upgrade.series.in-progress` flag, which will give the charm one and only one chance to disable and stop its application services in preparation for the upgrade. Once that flag is set and the charm's handlers have had a chance to respond, Juju will no longer run any charm code for the duration of the upgrade.

Once the operator has completed the upgrade, they will run:

```
juju upgrade-series <machine> complete
```

Juju will once again enable the charm code to run, and the framework will re-bootstrap the charm environment to ensure that it is setup properly for the new OS series. it will then remove the `upgrade.series.in-progress` flag. At this point, the charm should check the new OS series and perform any necessary migration the application may require to run on the new OS (unless that was to be performed manually by the operator). Finally, the charm should re-enable and start its application services.

Note that it is likely that the charm will need an additional self-managed flag to track whether the application services were disabled. The handlers might look something along the lines of:

```
@when('charm.application.started')
@when('upgrade.series.in-progress')
def disable_application():
    stop_app_services()
    disable_app_services()
    set_flag('charm.application.disabled')
```

(continues on next page)

(continued from previous page)

```
@when('charm.application.disabled')
@when_not('upgrade.series.in-progress')
def enable_application():
    enable_app_services()
    start_app_services()
    clear_flag('charm.application.disabled')
```

2.8.4 charms.reactive.bus

Summary

BrokenHandlerException	
ExternalHandler	A variant Handler for external executable actions (such as bash scripts).
FlagWatch	
Handler	Class representing a reactive flag handler.
discover	Discover handlers based on convention.
dispatch	Dispatch registered handlers.

Reference

exception charms.reactive.bus.**BrokenHandlerException**(*path*)
Bases: Exception

class charms.reactive.bus.**ExternalHandler**(*filepath*)
Bases: *charms.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 charms.reactive.bus.**FlagWatch**
Bases: object

classmethod change(*flag*)

classmethod commit()

classmethod iteration(*i*)

key = 'reactive.state_watch'

classmethod reset()

classmethod watch(*watcher*, *flags*)

class charms.reactive.bus.**Handler**(*action*, *suffix=None*)
Bases: object

Class representing a reactive flag 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_post_callback(*callback*)
Add a callback to be run after the action is invoked.

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

classmethod clear()
Clear all registered handlers.

classmethod get(*action*, *suffix=None*)
Get or register a handler for the given action.

Parameters

- **action** (*func*) – Callback that is called when invoking the Handler
- **suffix** (*func*) – Optional suffix for the handler's ID

classmethod get_handlers()
Get all registered handlers.

has_args
Whether or not this Handler has had any args added via *add_args()*.

id()

invoke()
Invoke this handler.

register_flags(*flags*)
Register flags as being relevant to this handler.

Relevant flags will be used to determine if the handler should be re-invoked due to changes in the set of active flags. If this handler has already been invoked during this *dispatch()* run and none of its relevant flags have been set or removed since then, then the handler will be skipped.

This is also used for linting and composition purposes, to determine if a layer has unhandled flags.

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

charms.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.

charms.reactive.bus.**dispatch**(*restricted=False*)
Dispatch registered handlers.

When dispatching in restricted mode, only matching hook handlers are executed.

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 flags will not trigger new handlers until the next iteration, to ensure that chained flags are invoked in a predictable order.
- Removed flags will cause the current set of matched handlers to be re-tested, to ensure that no handler is invoked after its matching flag has been removed.
- Other than the guarantees mentioned above, the order in which matching handlers are invoked is undefined.
- Flags 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()`.

Changelog

2.9 1.3.0

Monday Aug 26 2019

- Add pattern for request / response Endpoints (#215)
- Update link to the juju docs on debugging (#214)
- Separate layer-options sidenote from the main text (#211)

2.10 1.2.1

Wednesday Apr 3 2019

- Fix errant str.format handling of flags in expand_name (#210)
- Remove departed flag when joined (#209)

2.11 1.2.0

Wednesday Feb 6 2019

- Add ability to trigger on flag being cleared (#205)
- Add documentation for python_packages layer option (#204)
- Fix docs on upgrade series for final syntax (#203)
- Add OS Series Upgrades to main index (#202)
- Turn on flag and handler log tracing for all charms (#200)
- Update docs around hook.template and call out removing apt package (#199)

2.12 1.1.2

Thursday Oct 4 2018

- Adjust imports to work with Python 3.4 (#194)
- Adjust tests to work with older Ubuntu 14.04 (trusty) packages
- Update CI for charm-tools snap confinement change.

2.13 1.1.1

Friday Sep 28 2018

- Add is_data_changed to export list (#193)

2.14 1.1.0

Friday Sep 28 2018

- Flag and handler trace logging (#191)
- Add non-destructive version of data_changed (#188)

2.15 1.0.0

Wednesday Aug 8 2018

- Preliminary support for operating system series upgrades (#183)
- Hotfix for Python 3.4 incompatibility (#181)
- Hotfix adding missed backwards compatibility alias (#176)
- Documentation updates, including merging in core layer docs (#186)
- Acknowledgment by version number that this is mature software (and has been for quite some time).

2.16 0.6.3

Tuesday Apr 24 2018

- Export endpoint_from_name as well (#174)
- Rename Endpoint.joined to Endpoint.is_joined (#168)
- Only pass one copy of self to Endpoint method handlers (#172)
- Make Endpoint.from_flag return None for unset flags (#173)
- Fix hard-coded version in docs config (#167)
- Fix documentation of unit_name and application_name on RelatedUnit (#165)
- Fix setdefault on Endpoint data collections (#163)

2.17 0.6.2

Friday Feb 23 2018

- Hotfix for issue #161 (#162)
- Add diagram showing endpoint workflow and all_departed_units example to docs (#157)
- Fix doc builds on RTD (#156)

2.18 0.6.1

- Separate departed units from joined in Endpoint (#153)
- Add deprecated placeholder for RelationBase.from_state (#148)

2.19 0.6.0

- Endpoint base for easier interface layers (#123)
- Public API is now only documented via the top level charms.reactive namespace. The internal organization of the library is not part of the public API.
- Added layer-basic docs (#144)
- Fix test error from juju-wait snap (#143)
- More doc fixes (#140)
- Update help output in charms.reactive.sh (#136)
- Multiple docs fixes (#134)
- Fix import in triggers.rst (#133)
- Update README (#132)
- Fixed test, order doesn't matter (#131)
- Added FAQ section to docs (#129)

- Deprecations:
  - relation_from_name (renamed to endpoint_from_name)
  - relation_from_flag (renamed to endpoint_from_flag)
  - RelationBase.from_state (use endpoint_from_flag instead)

2.20 0.5.0

- Add flag triggers (#121)
- Add integration test to Travis to build and deploy a reactive charm (#120)
- Only execute matching hooks in restricted context. (#119)
- Rename "state" to "flag" and deprecate "state" name (#112)
- Allow pluggable alternatives to RelationBase (#111)
- Deprecations:
  - State
  - StateList
  - set_state (renamed to set_flag)
  - remove_state (renamed to clear_flag)
  - toggle_state (renamed to toggle_flag)
  - is_state (renamed to is_flag_set)
  - all_states (renamed to all_flags)
  - any_states (renamed to any_flags)
  - get_states (renamed to get_flags)
  - get_state
  - only_once
  - relation_from_state (renamed to relation_from_flag)

2.21 0.4.7

- Move docs to ReadTheDocs because PythonHosted is deprecated
- Fix cold loading of relation instances (#106)

2.22 0.4.6

- Correct use of templating.render (fixes #93)
- Add comments to bash reactive wrappers
- Use the standard import mechanism with module discovery

Python Module Index

C

Index