

# Open Source MANO

OSM Hackfest – Session 7a  
Adding day-1/day-2 configuration to your VNF  
Creating your first proxy charm  
Eduardo Sousa (Whitestack)

# What is Juju?

- Juju is an open source modeling tool, composed of a controller, models, and charms, for operating software in the cloud.
- Juju can handle configuration, relationships between services, lifecycle and scaling.
- This ensures that common elements such as databases, messaging systems, key value stores, logging infrastructure and other 'glue' functions are available as charms for automatic integration, reducing the burden on vendors and integrators.

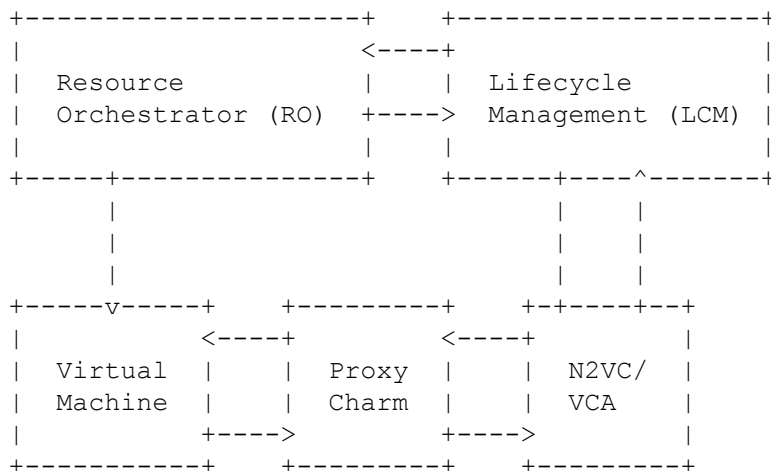
# What is a Charm?

- A charm is a collection of software containing all of the logic to install, configure, and scale cloud-based applications in a repeatable and reliable way.
- Charms are installed on a machine, running a cloud image, and handle the full lifecycle of an application, including day-0, day-1, and day-2 config.
- But...

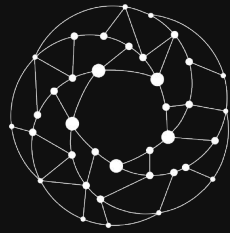
- OSM Release FOUR\* uses “proxy charms”, where the charm is installed into an LXD container, and is only responsible for day-1 and day-2 configuration, executed remotely (typically via ssh).
- Don’t worry! Proxy charm support is being expanded to support more features of full charms, and will still be supported in future releases.

\* Full charm support is a feature targeted for a R4 point release.

Here is a simple diagram showing how a proxy charm fits into the OSM workflow:



- A VNF package is instantiated via the LCM
- The LCM requests a virtual machine from the RO
- The RO instantiates a VM with your VNF image
- The LCM instructs N2VC, using the VCA, to deploy a VNF proxy charm, and tells it how to access your VM (hostname, user name, and password)



Open Source  
**MANO**

# Preparing your development environment

# Install the charm tools

Install charm tools via snap:

```
$ sudo snap install charm  
charm 2.2.3 from 'charms' installed
```

```
$ charm version
```

```
charm 2.2.2  
charm-tools 2.2.3
```

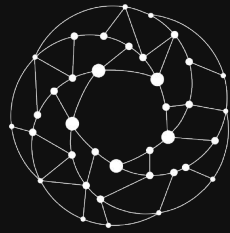
# Setup your Charming environment

Create the directories we'll use for our charm:

```
mkdir -p ~/charms/layers
```

Tell the charm command where our workspace is (for best results, add this to ~/.bashrc):

```
export JUJU_REPOSITORY=~ / charms
```



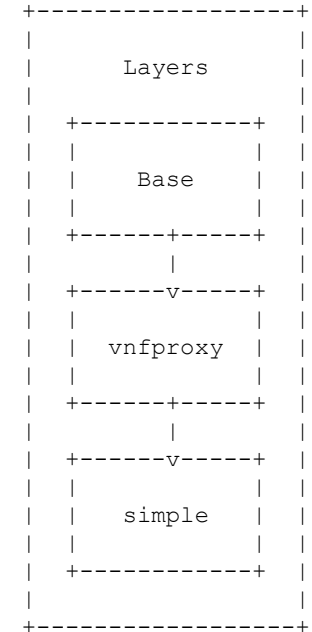
Open Source  
**MANO**

# Understanding charms

- The *Reactive* programming pattern that allows a charm to respond to changes in state, including lifecycle events, in an asynchronous way.
- Lifecycle events may tell the charm to install, start, or stop an application, to perform leadership election, to collect metrics, or to upgrade the charm itself.

# Layers

- Layers are encapsulations of charm code that lend themselves to being reused across charms.
- The Base layer contains the core code needed for other layers to function.
- Vnfproxy is a runtime layer providing common functionality to interoperate with a VNF.
- Simple is the charm layer containing code to manage *your* vnf.



# Creating a VNF Proxy charm

# Change into the layers folder

```
$ cd $JUJU_REPOSITORY/layers
```

# Invoke the charm command to create a charm layer called 'simple'

```
$ charm create simple
```

```
$ cd simple
```

# Anatomy of a charm layer

To the right is the contents of your simple charm.

For the purposes of this example, we will ignore the struck-through files.

```
$JUJU_REPOSITORY/layers
└─ simple
   ├── config.yaml
   ├── icon.svg
   ├── layer.yaml
   ├── metadata.yaml
   ├── reactive
   │   └── simple.py
   ├── README.ex
   └── tests
       ├── 00-setup
       └── 10-deploy
```

# Anatomy of a layer

`layer.yaml` defines which base and runtime layers your charm depends on.

Edit `layer.yaml` to include the `vnfproxy` layer:

```
includes: ['layer:basic', 'layer:vnfproxy']
options:
  basic:
    use_venv: false
```

```
$JUJU_REPOSITORY/layers
└─ simple
   └─ config.yaml
   └─ icon.svg
   └─ layer.yaml
   └─ metadata.yaml
   └─ reactive
      └─ simple.py
   └─ README.ex
   └─ tests
      └─ 00-setup
      └─ 10-deploy
```

# Anatomy of a layer

Edit `metadata.yaml` with the name and description of your charm:

```
name: simple
summary: A simple VNF proxy charm
maintainer: Name <user@domain.tld>
subordinate: false
series: ['xenial']
```

```
$JUJU_REPOSITORY/layers
└─ simple
   └─ config.yaml
   └─ icon.svg
   └─ layer.yaml
   └─ metadata.yaml
   └─ reactive
      └─ simple.py
   └─ README.ex
   └─ tests
      └─ 00-setup
      └─ 10-deploy
```

# Building your first charm

```
$ charm build
```

```
build: Destination charm directory: ~/charms/builds/simple
```

```
build: Please add a `repo` key to your layer.yaml, with a url from which your layer can be cloned.
```

```
build: Processing layer: layer:basic
```

```
build: Processing layer: layer:sshproxy
```

```
build: Processing layer: layer:vnfproxy
```

```
build: Processing layer: simple (from .)
```

```
proof: W: Includes template README.ex file
```

```
proof: W: README.ex includes boilerplate: Step by step instructions on using the charm:
```

```
proof: W: README.ex includes boilerplate: You can then browse to http://ip-address to configure the service.
```

```
proof: W: README.ex includes boilerplate: - Upstream mailing list or contact information
```

```
proof: W: README.ex includes boilerplate: - Feel free to add things if it's useful for users
```

```
proof: I: all charms should provide at least one thing
```

# Examining the compiled charm

The ``charm build`` command takes all of the layers defined in `layer.yaml`, combines them into a single charm, and caches the dependencies in the ``wheelhouse`` directory for faster installation.

```
$ ls $JUJU_REPOSITORY/builds/simple
```

actions	bin	copyright	hooks	layer.yaml	Makefile
reactive	README.md	simple	tox.ini	actions.yaml	config.yaml
deps	icon.svg	lib	README.ex	metadata.yaml	tests
requirements.txt		wheelhouse			

# Adding an action

Actions are functions that can be called automatically when a VNF is initialized (day-1 configuration) or on-demand by the operator (day-2 configuration).

In OSM terminology, we know these as config primitives.

# Define an action

Let's create `actions.yaml` in the root of the simple charm:

```
touch:
  description: "Touch a file on the VNF."
  params:
    filename:
      description: "The name of the file to touch."
      type: string
      default: ""
  required:
    - filename
```

# Create the action handler

\$ mkdir actions

Create `actions/touch`, with the contents to the right.

When you're done, mark the script executable:

\$ chmod +x actions/touch

```
#!/usr/bin/env python3
import sys
sys.path.append('lib')
from charms.reactive import main, set_flag
from charmhelpers.core.hookenv import action_fail, action_name

set_flag('actions.{}'.format(action_name()))

try:
    main()
except Exception as e:
    action_fail(repr(e))
```

**Note:** The same content has to be used for every action in the charm layer. It is only a boilerplate script to invoke the reactive framework

# Handle the action

Edit

``reactive/simple.py``.

This is where all reactive states are handled.

```
from charmhelpers.core.hookenv import (  
    action_get,  
    action_fail,  
    action_set,  
    status_set,  
)  
  
from charms.reactive import (  
    clear_flag,  
    set_flag,  
    when,  
    when_not,  
)  
import charms.sshproxy
```

# Handle the action

## Edit

``reactive/simple.py``.

This is where all reactive states are handled.

```
# Set the charm's state to active so the LCM knows
# it's ready to work.
@when_not('simple.installed')
def install_simple_proxy_charm():
    set_flag('simple.installed')
    status_set('active', 'Ready!')
```

# Handle the action

## Edit

``reactive/simple.py``.

This is where all reactive states are handled.

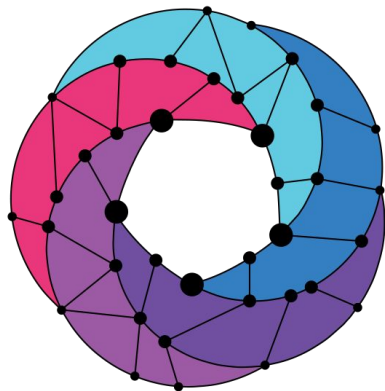
```
# Define what to do when the `touch` primitive is invoked.
@when('actions.touch')
def touch():
    err = ''
    try:
        filename = action_get('filename')
        cmd = ['touch {}'.format(filename)]
        result, err = charms.sshproxy._run(cmd)
    except:
        action_fail('command failed:' + err)
    else:
        action_set({'output': result})
    finally:
        clear_flag('actions.touch')
```

**Note:** For every action in the charm layer you need to add a `@when` decorator and the function to be run

# That's it!

We're ready to compile the charm with our new action:

\$ charm build



# Open Source MANO

Find us at:

[osm.etsi.org](https://osm.etsi.org)  
[osm.etsi.org/wikipub](https://osm.etsi.org/wikipub)