LinchPin Documentation

Release 1.0.4

Samvaran Kashyap Rallabandi

Sep 22, 2017

Contents

CHAPTER 1

About LinchPin

Welcome to the LinchPin documentation!

LinchPin is a hybrid cloud orchestration tool. Its intended purpose is managing cloud resources across multiple infrastructures. These resources can be provisioned, decommissioned, and configured all using a topology file and a simple command-line interface.

Additionally, LinchPin provides a Python API (and soon a RESTful API) for managing resources. The cloud management component is backed by *Ansible <https://ansible.com>*. The front-end API manages the interface between the command line (or other interfaces) and calls to the Ansible API.

This documentation covers the current released version of LinchPin (1.0.4). For recent features, we attempt to note in each section the version of LinchPin where the feature was added.

Introduction

Before getting heavily into LinchPin, let's cover some of the basics. The topics below will cover everything needed to get going with LinchPin. For more advanced topics, refer to the *main documentation page*.

- *Installation*
- *Getting Started*
- *Configuration*

See also:

User Mailing List Subscribe and participate. A great place for Q&A

irc.freenode.net #linchpin IRC chat channel

Installation

Topics

Currently, LinchPin can be run from any machine with Python 2.6+ (Python 3.x is currently experimental), and requires Ansible 2.2.1. There are many other dependencies, depending on the provider. The core providers are *OpenStack*, *Amazon EC2*, and *Google Compute Cloud*. If enabled on the host system, *Libvirt* can also be used out of the box.

Refer to your specific operating system for directions on the best method to install Python, if it is not already installed. Many modern operating systems will have Python already installed. This is typically the case in all versions of Linux and OS X, but the version present might be older than the version needed for use with Ansible. You can check the version by typing `python --version`.

If the system installed version of Python is older than 2.6, many systems will provide a method to install updated versions of Python in parallel to the system version (eg. virtualenv).

Minimal Software Requirements

As LinchPin is heavily dependent on Ansible, this is a core requirement. Beyond installing Ansible, there are several packages that need to be installed:

```
* libffi-devel
* openssl-devel
* libyaml-devel
* gmp-devel
* libselinux-python
* make
```

For Fedora/CentOS/RHEL the necessary packages should be installed.

```
$ sudo yum install python-virtualenv libffi-devel \
openssl-devel libyaml-devel gmp-devel libselinux-python make
```

Note: Fedora will present an output suggesting the use of *dnf* as a replacement for yum.

Installing LinchPin

Note: Currently, linchpin is not packaged for any major Operating System. If you'd like to contribute your time to create a package, please contact the linchpin mailing list.

Create a virtualenv to install the package using the following sequence of commands (requires virtualenvwrapper).

```
$ mkvirtualenv linchpin
..snip..
(linchpin) $ pip install linchpin
..snip..
```

Note: mkvirtualenv is optional dependency you can install from *http://virtualenvwrapper.readthedocs.io/en/latest/install.html* , if you would like to use python virtualenv use following commands instead. mkdir linchpin virtualenv linchpin source linchpin/bin/activate

To deactivate the virtualenv.

```
(linchpin) $ deactivate
$
```

Then reactivate the virtualenv.

```
$ workon linchpin
(linchpin) $
```

If testing or docs is desired, additional steps are required.

```
(linchpin) $ pip install linchpin[docs]
(linchpin) $ pip install linchpin[tests]
```

Source Installation

As an alternative, LinchPin can be installed via github. This may be done in order to fix a bug, or contribute to the project.

```
(linchpin) $ git clone git://github.com/CentOS-PaaS-SIG/linchpin
..snip..
(linchpin) $ pip install file://$PWD/linchpin
```

See also:

User Mailing List Subscribe and participate. A great place for Q&A

irc.freenode.net #linchpin IRC chat channel

Getting Started

Topics

Foreword

Now that LinchPin is installed according to *Installation*, it is time to see how it works. This guide is essentially a quick start guide to getting up and running with LinchPin.

LinchPin is a command-line utility, a Python API, and Ansible playbooks. This document focuses on the command-line interface.

Terminology

LinchPin, while it attempts to be a simple tool for provisioning resources, still does have some complexity. To that end, this section attempts to define the minimal bits of terminology needed to understand how to use the `linchpin` command-line utility.

Topology

The *topology* is a set of rules, written in YAML, that define the way the provisioned systems should look after executing linchpin. Generally, the *topology* and *topology_file* values are interchangeable, except where the YAML is specifically indicated. A simple **dummy** topology is shown here.

```
---
topology_name: "dummy_cluster" # topology name
resource_groups:
  -
    resource_group_name: "dummy"
    resource_group_type: "dummy"
    resource_definitions:
      -
        name: "web"
        type: "dummy_node"
        count: 3
```

This topology describes a set of three (3) dummy systems that will be provisioned when *linchpin up* is executed. The names of the systems will be 'web_#.example.net', where # indicates the count (usually *0*, *1*, and *2*). Once provisioned, the resources will be output and stored for reference. The output *resources* data can then be used to generated an inventory, or passed as part of a *linchpin destroy* action.

Inventory Layout

The *inventory_layout* or *layout* mean the same thing, a YAML definition for providing an Ansible static inventory file, based upon the provided topology. A YAML *layout* is stored in a *layout_file*.

```
---
inventory_layout:
  vars:
    hostname: __IP__
  hosts:
```

```
    example-node:
      count: 3
      host_groups:
        - example
  host_groups:
    example:
      vars:
        test: one
```

The above YAML allows for interpolation of the ip address, or hostname as a component of a generated inventory. A host group called *example* will be added to the Ansible static inventory, along with a section called *example:vars* containing *test = one*. The resulting static Ansible inventory is shown here.

```
[example:vars]
test = one

[example]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net

[all]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net
```

PinFile

A *PinFile* takes a *topology* and an optional *layout*, among other options, as a combined set of configurations as a resource for provisioning. An example *Pinfile* is shown.

```
dummy1:
  topology: dummy-cluster.yml
  layout: dummy-layout.yml
```

The *PinFile* collects the given *topology* and *layout* into one place. Many *targets* can be referenced in a single *PinFile*.

The *target* above is named *dummy1*. This *target* is the reference to the *topology* named *dummy-cluster.yml* and *layout* named *dummy-layout.yml*. The *PinFile* can also contain definitions of *hooks* that can be executed at certain pre-defined states.

Running `linchpin`

As stated above, this guide is about using the command-line utility, `linchpin`. First off, simply execute `linchpin` to see some options.

```
$ linchpin
Usage: linchpin [OPTIONS] COMMAND [ARGS]...

  linchpin: hybrid cloud orchestration

Options:
  -c, --config PATH       Path to config file
  -w, --workspace PATH    Use the specified workspace if the familiar Jenkins
                          $WORKSPACE environment variable is not set
```

```
  -v, --verbose           Enable verbose output
  --version               Prints the version and exits
  --creds-path PATH       Use the specified credentials path if WORKSPACE
                          environment variable is not set
  -h, --help              Show this message and exit.

Commands:
  init     Initializes a linchpin project.
  up       Provisions nodes from the given target(s) in...
  destroy  Destroys nodes from the given target(s) in...
```

What can be seen immediately is a simple description, along with options and arguments that can be passed to the command. The three commands found near the bottom of this help are where the focus will be for this document.

Initialization (init)

Running `linchpin init` will generate the directory structure needed, along with an example *PinFile*, *topology*, and *layout* files. One important option here, is the *–workspace*. When passing this option, the system will use this as the location for the structure. The default is the current directory.

```
$ export WORKSPACE=/tmp/workspace
$ linchpin init
PinFile and file structure created at /tmp/workspace
$ cd /tmp/workspace/
$ tree
.
- credentials
- hooks
- inventories
- layouts
|   - example-layout.yml
- PinFile
- resources
- topologies
    - example-topology.yml
```

At this point, one could execute `linchpin up` and provision a single libvirt virtual machine, with a network named *linchpin-centos71*. An inventory would be generated and placed in `inventories/libvirt.inventory`. This can be known by reading the `topologies/example-topology.yml` and gleaning out the *topology_name* value.

Provisioning (up)

Once a *PinFile*, *topology*, and optionally a *layout* are in place, provisioning can happen.

Note: For this section, the dummy tooling will be used as it is much simpler and doesn't require anything extra to be configured. The dummy provider creates a temporary file, which represents provisioned hosts. If the temporary file does not have any data, hosts have not been provisioned, or they have been recently destroyed.

The dummy *topology*, *layout*, and *PinFile* are shown above in the appropriate sections. The tree would be very simple.

```
$ tree
.
```

```
- inventories
- layouts
|   - dummy-layout.yml
- PinFile
- resources
- topologies
    - dummy-cluster.yml
```

Performing the command `linchpin up` should generate *resources* and *inventory* files based upon the *topology_name* value. In this case, is `dummy_cluster`.

```
$ linchpin up
target: dummy1, action: up

$ ls {resources,inventories}/dummy*
inventories/dummy_cluster.inventory  resources/dummy_cluster.output
```

To verify resources with the dummy cluster, check `/tmp/dummy.hosts`

```
$ cat /tmp/dummy.hosts
web-0.example.net
web-1.example.net
web-2.example.net
```

This is reflected in both the *resources* (not shown) and *inventory* files.

```
[example:vars]
test = one

[example]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net

[all]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net
```

Teardown (destroy)

As expected, LinchPin can also perform *teardown* of *resources*. A teardown action generally expects that resources have been *provisioned*. However, because Ansible is idempotent, `linchpin destroy` will only check to make sure the resources are up. Only if the resources are already up will the teardown happen.

The command `linchpin destroy` will either use *resources* and/or *topology* files to determine the proper *teardown* procedure. The *dummy* Ansible role does not use the resources, only the topology during teardown.

```
$ linchpin destroy
target: dummy1, action: destroy

$ cat /tmp/dummy.hosts
-- EMPTY FILE --
```

Note: The teardown functionality is slightly more limited around ephemeral resources, like networking, storage,

etc. It is possible that a network resource could be used with multiple cloud instances. In this way, performing a `linchpin destroy` does not teardown certain resources. This is dependent on each providers implementation.

See specific implementations for each of the providers.

Multi-Target Actions

LinchPin can *provision* or *teardown* any number of *resources*. If a *PinFile* has multiple *targets*, and is called without a target name, all targets will be executed. Given this PinFile.

```
example:
  topology: example-topology.yml
  layout: example-layout.yml

example2:
  topology: example2-topology.yml
  layout: example2-layout.yml

dummy1:
  topology: dummy-cluster.yml
  layout: dummy-layout.yml
```

A call to `linchpin up` would *provision* and generate an Ansible static *inventory* for each *target*.

```
$ linchpin up
target: dummy1, action: up

target: example2, action: up

target: example, action: up
```

See also:

linchpincli Linchpin Command-Line Interface

User Mailing List Subscribe and participate. A great place for Q&A

irc.freenode.net #linchpin IRC chat channel

Configuration

Before resources can be provisioned in any of the environments through the use of linchpin, the environment must be configured to specify the resources required.

General Configuration

Managing LinchPin requires a few configuration files. Beyond *linchpin.conf*, there are a few other configurations that need to be created. When running linchpin, four different locations are checked for linchpin.conf files. Files are checked in the following order:

1. linchpin/library/path/linchpin.conf
2. /etc/linchpin.conf
3. ~/.config/linchpin/linchpin.conf

4. path/to/workspace/linchpin.conf

The linchpin configuration parser supports overriding and extension of configurations. Therefore, after the files are checked for existence, the existing configuration files are read and if linchpin finds two or more different configuration files to contain the same configuration section header, the header that was parsed more recently will provide the configuration for that section. Therefore, if the user wants to add their own configurations to their linchpin workpace, the the user should add their configurations to a linchpin.conf file in the root of their workspace. This way, their file will be parsed last and their configurations will take precedence over all other configurations.

To add your own configurations, simply create a linchpin.conf file in the root of your workspace using your preferred text editor and write configuration in a *.ini* style. Here's an example:

:: [Section Header] key1 = value1 key2 = value2

Topics

Workspace

Initialization

Running `linchpin init` will generate the directory structure needed, along with an example *PinFile*, *topology*, and *layout* files. One important option here, is the *–workspace*. When passing this option, the system will use this as the location for the structure. The default is the current directory.

```
$ export WORKSPACE=/tmp/workspace
$ linchpin init
PinFile and file structure created at /tmp/workspace
$ cd /tmp/workspace/
$ tree
.
- credentials
- hooks
- inventories
- layouts
|   - example-layout.yml
- PinFile
- resources
- topologies
    - example-topology.yml
```

At this point, one could execute `linchpin up` and provision a single libvirt virtual machine, with a network named *linchpin-centos71*. An inventory would be generated and placed in `inventories/libvirt.inventory`. This can be known by reading the `topologies/example-topology.yml` and gleaning out the *topology_name* value.

PinFile

A *PinFile* takes a *topology* and an optional *layout*, among other options, as a combined set of configurations as a resource for provisioning. An example *Pinfile* is shown.

```
dummy1:
  topology: dummy-cluster.yml
  layout: dummy-layout.yml
```

The *PinFile* collects the given *topology* and *layout* into one place. Many *targets* can be referenced in a single *PinFile*.

The *target* above is named *dummy1*. This *target* is the reference to the *topology* named *dummy-cluster.yml* and *layout* named *dummy-layout.yml*. The *PinFile* can also contain definitions of *hooks* that can be executed at certain pre-defined states.

Topologies

The *topology* is a set of rules, written in YAML, that define the way the provisioned systems should look after executing linchpin. Generally, the *topology* and *topology_file* values are interchangeable, except where the YAML is specifically indicated. A simple **dummy** topology is shown here.

```
---
topology_name: "dummy_cluster" # topology name
resource_groups:
  -
    resource_group_name: "dummy"
    resource_group_type: "dummy"
    resource_definitions:
      -
        name: "web"
        type: "dummy_node"
        count: 3
```

This topology describes a set of three (3) dummy systems that will be provisioned when *linchpin up* is executed. The names of the systems will be 'web_#.example.net', where # indicates the count (usually *0*, *1*, and *2*). Once provisioned, the resources will be output and stored for reference. The output *resources* data can then be used to generated an inventory, or passed as part of a *linchpin destroy* action.

Inventory Layouts

The *inventory_layout* or *layout* mean the same thing, a YAML definition for providing an Ansible static inventory file, based upon the provided topology. A YAML *layout* is stored in a *layout_file*.

```
---
inventory_layout:
  vars:
    hostname: __IP__
  hosts:
    example-node:
      count: 3
      host_groups:
        - example
  host_groups:
    example:
      vars:
        test: one
```

The above YAML allows for interpolation of the ip address, or hostname as a component of a generated inventory. A host group called *example* will be added to the Ansible static inventory, along with a section called *example:vars* containing *test* = *one*. The resulting static Ansible inventory is shown here.

```
[example:vars]
test = one

[example]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net

[all]
web-2.example.net hostname=web-2.example.net
web-1.example.net hostname=web-1.example.net
web-0.example.net hostname=web-0.example.net
```

Topologies

A topology is a specification of which resources from which environments are being requested from a linchpin run. Since each environment has different sets of requirements, the exact values and structure of a topology file will vary based on where resources are to be provisioned. In this document some broad discussion of topologies will be addressed. More extensive examples pertaining to specific environments will be given in a separate section of the documentation.

Topology

Broadly speaking, a linchpin topology file is a list of resources to be provisioned from each environment. It is possible and a very common use case to list multiple resources, even multiple types of resources, in a single topology file. A less common use case, but still supported, is to provision multiple resources across multiple environments.

The topology file does not designate the format of the output, nor map the particular resources that get provisioned into output inventory groups. That is the work of the layouts file.

Structure

A topology is a YAML file or a JSON file formatted with a single top-level object.

There are two top level keys in a topology.

The first key is named *topology_name* and is a relatively free-form string that identifies the user-friendly name for this particular topology. For best practices, this should resemble the file name and possibly the name of the key from the PinFile.

The second key is the *resource_groups* key. This key is an array of objects.

Resource Group

Each entry in the *resource_group* key array is itself an object hash with three object keys.

The first key is *resource_group_name*, and is a user-friendly name for the resources that will be provisioned from this group definition.

The second key is *res_group_type* and must be a string of a limited set. This set must match to the particular environment. Some environments can define different types of valid values. As an example, the value *duffy* will define a resource type to be provisioned in a Duffy architecture, whereas the value *beaker* will contain definitions of a set of servers to be provisioned in a Beaker environment.

The third key is *res_defs*. This key defines an array of objects. Each of these objects' exact form will be dictated by the value of *res_group_type*. To see more information on the structure of these values, check the example topologies section of this documentation.

Layouts

A layout file is the current mechanism to define mappings between the resources provisioned out of the topology and the Ansible inventory groups that are output.

Topics

Structure

As with a topology file, a layout file is a YAML file or a JSON file with a single root object hash. There is one top-level entry in the hash, named *inventory_layout*. The *inventory_layout* value is itself an object that has a few fields inside of it.

Hosts

The first hash value is *hosts*, which contains an object hash as a value. The keys of that hash are the names of hosts that have been provisioned out of the topology. Each host has two properties, *count* and *host_groups*.

The *count* property says how many of the topology hosts are to share this inventory hostname. For instance, if the host is "webserver" and *count* is 2, then this will generate hosts in the output inventory named "webserver-1" and "webserver-2". This value is optional and defaults to 1 when it's not present.

The *host_groups* field contains an array of Ansible inventory groups into which all the hosts under this hash will get placed. This value is optional and will default to an empty list when not filled. In that case, the host will be named into the inventory with its host vars, and added to default 'all' group.

As an example, assume you provisioned three hosts and you wanted one database and two applicaiton hosts. Your goal is to get to an Ansible inventory that looks like this:

```
[backend]
database

[frontend]
webhost-1
webhost-2

[ldap]
database
webhost-1
```

```
webhost-2

[security_updates]
database
```

Then your hosts object would look like this:

```
hosts:
  database:
    count: 1
    host_groups:
      - backend
      - ldap
      - security_updates
  webhost:
    count: 2
    host_groups:
      - ldap
      - frontend
```

Ansible Variables

Topics

Inputs

The following variables can be set using ansible extra_vars, including in the `[evars]` section of linchpin.conf, to alter linchpin's default behavior.

topology

topology_file A set of rules, written in YAML, that define the way the provisioned systems should look after executing linchpin.

Generally, the *topology* and *topology_file* values are interchangeable, except after the file has been processed.

schema

schema_file JSON description of the format for the topology. *(schema_v3, schema_v4 are still available)*

layout

layout_file YAML definition for providing an ansible (currently) static inventory file, based upon the provided topology.

inventory

inventory_file If layout / layout_file is provided, this will be the location of the resulting ansible inventory.

linchpin_config if passed on the command line with `-c/--config`, should be an ini-style config file with linchpin default configurations (see BUILT-INS below for more information)

resources

resources_file File with the resource outputs in a JSON formatted file. Useful for teardown (destroy,down) actions depending on the provider.

workspace If provided, the above variables will be adjusted and mapped according to this value. Each path will use the following variables:

```
topology / topology_file = /<topologies_folder>
layout / layout_file = /<layouts_folder>
resources / resources_file = /resources_folder>
inventory / inventory_file = /<inventories_folder>

.. note:: schema is not affected by this pathing
```

If the `WORKSPACE` environment variable is set, it will be used here. If it is not, this variable can be set on the command line with `-w/--workspace`, and defaults to the location of the PinFile bring provisioned.

Built-ins

These variables **SHOULD NOT** be changed!

lp_path base path for linchpin playbooks and python api

lpconfig `<lp_path>/linchpin.conf`, unless overridden by *linchpin_config*

Defaults

While the variables here can also be passed as extra-vars, the values are the defaults and it is recommended not to change them. These values are defined in `<lp_path>/linchpin.conf` by default.

async *(boolean, default: False)*

Used to enable asynchronous provisioning/teardown

async_timeout *(int, default: 1000)*

How long the resouce collection (formerly outputs_writer) process should wait

output *(boolean, default: True, previous: no_output)*

Controls whether resources will be written to the resources_file

check_mode *(boolean, default: no)*

This option does nothing at this time, though it may eventually be used for dry-run functionality based upon the provider

schemas_folder *(file_path, default: schemas)*

relative path to schemas

playbooks_folder *(file_path, default: provision)*

relative path to playbooks, only useful to the linchpin API and CLI

layouts_folder *(file_path, default: layouts)*

relative path to layouts

topologies_folder *(file_path, default: topologies)*

relative path to topologies

default_schemas_path *(file_path, default: <lp_path>/defaults/<schemas_folder>)*

default path to schemas, absolute path. Can be overridden by passing schema / schema_file.

default_playbooks_path *(file_path, default: <lp_path>/defaults/playbooks_folder>)*

default path to playbooks location, only useful to the linchpin API and CLI

default_layouts_path *(file_path, default: <lp_path>/defaults/<layouts_folder>)*

default path to inventory layout files

default_topologies_path *(file_path, default: <lp_path>/defaults/<topologies_folder>)*

default path to topology files

default_resources_path *(file_path, default: <lp_path>/defaults/<resources_folder>, formerly: outputs)*

landing location for resources output data

default_inventories_path *(file_path, default: <lp_path>/defaults/<inventories_folder>)*

landing location for inventory outputs

See also:

Glossary Glossary

Example Topologies

Before using Linchpin, here are few Linchpin topology examples.

AWS Topologies

Topics

AWS EC2 Multiple Accounts

```
---
topology_name: "ex_aws_topo"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst"
        flavor: "t1.micro"
        res_type: "aws_ec2"
        region: "us-west-2"
        image: "ami-014cb561"
        count: 1
        keypair: "libra"
    assoc_creds: "master_aws_creds"
  -
    resource_group_name: "testgroup2"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst2"
        flavor: "t1.micro"
        res_type: "aws_ec2"
        region: "us-east-1"
        image: "ami-00a7636d"
        count: 2
        keypair: "libra"
    assoc_creds: "master_aws_creds"
  -
    resource_group_name: "testgroup3"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst2"
        flavor: "t1.micro"
        res_type: "aws_ec2"
        region: "us-east-1"
        image: "ami-00a7636d"
        count: 1
        keypair: "libra"
    assoc_creds: "sk_aws_creds"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
  -
    resource_group_name : "testgroup3"
```

```
    Name: "TestInstanceGroup3"
    test_var1: "test_var1 msg is grp3 hello"
    test_var2: "test_var2 msg is grp3 hello"
    test_var3: "test_var3 msg is grp3 hello"
  -
    resource_group_name : "testgroup4"
    Name: "TestInstanceGroup4"
    test_var1: "test_var1 msg is grp4 hello"
    test_var2: "test_var2 msg is grp4 hello"
    test_var3: "test_var3 msg is grp4 hello"
```

AWS EC2 Keypair

```
---
topology_name: "ex_aws_keypair_topo"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "aws"
    res_defs:
      - res_name: "ex_keypair_sk"
        res_type: "aws_ec2_key"
        region: "us-west-2"
    assoc_creds: "sk_aws_personal"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
```

AWS CFN EXAMPLE1

```
---
topology_name: "ex_cfn_topo"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "cfnsimplestackaws"
        res_type: "aws_cfn"
        region: "us-east-1"
        template_path: "/path/to/cfn_template"
    assoc_creds: "sk_aws_personal"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    cfn_params:
```

```
      KeyName: "sk_key"
      InstanceType: "t2.micro"
```

AWS CFN EXAMPLE2

```
---
topology_name: "ex_cfn_topo2"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "cfnsimplestackaws"
        res_type: "aws_cfn"
        region: "us-east-1"
        template_path: "/path/to/ec2_sample_cfn.template"
    assoc_creds: "sk_aws_personal"
  -
    resource_group_name: "testgroup2"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst2"
        flavor: "t2.micro"
        res_type: "aws_ec2"
        region: "us-east-1"
        image: "ami-fce3c696"
        count: 2
        keypair: "sk_key"
    assoc_creds: "sk_aws_personal"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    cfn_params:
      KeyName: "sk_key"
      InstanceType: "t2.micro"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
```

AWS FULLSTACK EXAMPLE

```
---
topology_name: "ex_aws_full_stack"
site: "testsite"
resource_groups:
  -
    resource_group_name: "testgroup1"
```

```
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst2"
        flavor: "t2.micro"
        res_type: "aws_ec2"
        region: "us-east-1"
        image: "ami-fce3c696"
        count: 1
        keypair: "sk_key"
      -
        res_name: "samvaranbucktest"
        res_type: "aws_s3"
        region: "us-west-2"
      -
        res_name: "ex_keypair_sk"
        res_type: "aws_ec2_key"
        region: "us-west-2"
    assoc_creds: "sk_aws_personal"
  -
    resource_group_name: "testgroup2"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "cfnsimplestackaws"
        res_type: "aws_cfn"
        region: "us-east-1"
        template_path: "/path/to/ec2_sample_cfn.template"
    assoc_creds: "sk_aws_personal"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup1"
    cfn_params:
      KeyName: "sk_key"
      InstanceType: "t2.micro"
```

Note: Source of the above mentioned examples is available here

AWS EC2 Security Groups EXAMPLE

```
---
topology_name: "aws_sg_topology"
resource_groups:
  -
    resource_group_name: "awssgtest"
    res_group_type: "aws"
    res_defs:
```

```
      -
        res_name: "aws_test_sg"
        res_type: "aws_sg"
        description: "AWS Security Group with ssh access"
        region: "us-east-1"
        rules:
       -
         rule_type: "inbound"
         from_port: 8 # type 8 is ICMP echo request
         to_port: -1
         proto: "icmp"
         cidr_ip: "0.0.0.0/0"
       -
         rule_type: "inbound"
         from_port: 22
         to_port: 22
         proto: "tcp"
         cidr_ip: "0.0.0.0/0"
       -
         rule_type: "outbound"
         from_port: "all"
         to_port: "all"
         proto: "all"
         cidr_ip: "0.0.0.0/0"
    assoc_creds: "aws_creds"
resource_group_vars:
  -
    resource_group_name : "awssgtest"
    test_var1: "test_var1 msg is grp1 hello"
```

Note: Source of the above AWS EC2 Security Groups example can be found at Example Topologies

Openstack Topologies

Topics

– *Provision*

Openstack Server

```
---
topology_name: "example_topo"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      -
        res_name: "ha_inst"
        flavor: "m1.small"
        res_type: "os_server"
        image: "rhel-6.5_jeos"
        count: 1
        keypair: "ci-factory"
        networks:
          - "e2e-openstack"
      -
        res_name: "web_inst"
        flavor: "m1.small"
        res_type: "os_server"
        image: "rhel-6.5_jeos"
        count: 1
        keypair: "ci-factory"
        networks:
          - "e2e-openstack"
    assoc_creds: "cios_e2e-openstack"
  -
    resource_group_name: "testgroup2"
    res_group_type: "openstack"
    res_defs:
      - res_name: "ano_inst"
        flavor: "m1.small"
        res_type: "os_server"
        image: "rhel-6.5_jeos"
        count: 1
        keypair: "ci-factory"
        networks:
          - "e2e-openstack"
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    test_var1: "test_var1 msg is grp1 hello "
    test_var2: "test_var2 msg is grp1 hello "
    test_var3: "test_var3 msg is grp1 hello "
  -
    resource_group_name : "testgroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
  -
```

```
    resource_group_name : "testgroup3"
    test_var1: "test_var1 msg is grp3 hello"
    test_var2: "test_var2 msg is grp3 hello"
    test_var3: "test_var3 msg is grp3 hello"
```

Openstack Keypair

```
---
topology_name: "ex_os_keypair"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      - res_name: "ex_keypair_sk"
        res_type: "os_keypair"
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
```

Openstack Cinder Volume

```
---
topology_name: "ex_os_vol"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      - res_name: "test_volume_sk"
        res_type: "os_volume"
        size: 1
        count: 3
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
```

Openstack Swift Container

```
---
topology_name: "ex_os_obj"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      - res_name: "testcontainer_sk"
        res_type: "os_object"
        access: "public"
        count: 2
    assoc_creds: "cios_e2e-openstack"
  -
    resource_group_name: "testgroup2"
    res_group_type: "openstack"
    res_defs:
      - res_name: "testit_sk"
        res_type: "os_object"
        access: "private"
        count: 2
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
```

Openstack Container & Volume

```
---
topology_name: "ex_os_obj_vol"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      - res_name: "test_volume_sk"
        res_type: "os_volume"
        size: 2
        count: 3
      - res_name: "testcontainer_sk"
        res_type: "os_object"
        access: "public"
        count: 3
```

```
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
```

Openstack Full Stack

```
---
topology_name: "ex_os_heat_topo"
site: "qeos"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "openstack"
    res_defs:
      -
        res_name: "ex_keypair_sk"
        res_type: "os_keypair"
      -
        res_name: "os_heat_template_sample"
        res_type: "os_heat"
        template_path: "/path/to/hot_template_sample2.yaml"
      - res_name: "ano_inst"
        flavor: "m1.small"
        res_type: "os_server"
        image: "rhel-6.5_jeos"
        count: 2
        keypair: "ci-factory"
        networks:
          - "e2e-openstack"
    assoc_creds: "cios_e2e-openstack"
  -
    resource_group_name: "testgroup2"
    res_group_type: "openstack"
    res_defs:
      - res_name: "test_volume_sk"
        res_type: "os_volume"
        size: 2
        count: 3
      - res_name: "testcontainer_sk"
        res_type: "os_object"
        access: "public"
        count: 3
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    heat_params:
      key_name: "ci-factory"
      image_id: "rhel-6.5_jeos"
      instance_type: "m1.small"
```

```
      network_name: "e2e-openstack"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
```

Steps to provision Single Host

Topics

Credentials

- save openstack credentials in standard `clouds.yml` file using below format and save the directory path containing clouds.yml in environment variable `CREDS_PATH`.

```
---
clouds:
  devstack:
    auth:
      username: "admin"
      password: "Secret123"
      project_name: "my-tenant"
      auth_url: "http://192.168.122.33:5000/v2.0"
```

Topology

- create topology file under `$WORKSPACE/topologies/openstack_topology.yml` as show below:

```
---
topology_name: "osp-test"
resource_groups:
```

```
-
  resource_group_name: "lp-test"
  resource_group_type: "openstack"
  resource_definitions:
    - name: "test1"
      type: "os_server"
      flavor: "m1.small"
      image: "rhel-6.5_jeos"
      count: 1
      keypair: "ci-factory"
      networks:
        - "e2e-openstack"
      fip_pool: "192.168.122.1/24"
  credentials:
      filename: "clouds.yml"
      profile: "devstack"
```

Provision

- provision the above topology

```
$ cd $WORKSPACE
$ export CREDS_PATH="/path/to/credential_dir/"
$ linchpin -v up
```

- Alternatively one could pass credentials path as an argument to linchpin

```
$ cd $WORKSPACE
$ linchpin -v --creds-path /path/to/dir_containing_clouds.yml/ up
```

OpenShift Topologies

Topics

Inventory Generation

It is important to note that OpenShift resources do not follow the normal rules of most other providers. When you provision a resource in OpenShift, there is no easy way for Linchpin to introspect any information about the resources

you have spun up. Accessing individual containers and pods directly is a violation of how most people expect OpenShift and container technologies in general to operate. Therefore, no output will be given into the generated Ansible inventory file for an OpenShift provisioning. OpenShift does not even expose a method to address an individual container and create or destroy one. It only exposes the pod level and above for creation, making entering into a particular container impossible.

Additionally, it is possible to use Linchpin to spin up resources in OpenShift that are not even containers, as any item other than an Event which may be created through the API can be created through the OpenShift provider layer in Linchpin. Thus, even if proper destination IP addresses could be introspected from the results, there is no guarantee that what is being created even has such a destination.

Accessing OpenShift Resources

Furthermore, individual containers will typically not expose SSH access to the process space. Such introspection of the containers needs to be done through native OpenShift methods such as the command line client "oc" and its sub commands like "exec" and "rsh". Information on how to access running pods and containers can be found in the external documentation for OpenShift, along with specific information from your cluster's administrator.

Note About Teardown

Again, OpenShift shows its special nature in the teardown step of infrastructure management. Most use cases, as is the case with the example below, will create what is known as a "replication controller". This is an object with the job of monitoring and maintaining multiple copies of a pod running across the cluster. The replication controller provides a very simple way to increase or decrease the quantity of running pods. If it detects that ond of its pods has stopped for any reason, it will attempt to recreate the pod again. This is good, as it gives a layer of automated infrastructure monitoring to ensure the required number of copies are running across the cluster.

However, this configuration creates a difficulty with teardown. If a topology file creates a replication controller with more than 0 pods (the example below creates a ReplicationController with 7 copies of the Jenkins slave pod running) that RC will work to keep the pods up, but it will not teardown those pods when the RC is deleted. Those pods will remain running until they are either killed manually or until their base process crashes. Thus, running "linchpin rise" followed by "linchpin drop" on this ReplicationController will leave seven orphaned pods running in the cluster unless they are cleaned up manually.

One way to avoid this is to "scale down" the RC by setting its number of active pods to 0 before deleting it. This will leave no orphaned pods behind. Alternatively, the pods could be deleted manually after deletion of the RC. Linchpin does not attempt to do the scaling automatically, as there are a vast number of possible scenarios for leaving orphaned items behind in a cluster. Pods are only referenced here as the most likely possibility, and are a clear example of something that could be orphaned on a cluster.

Example Topologies

Each of these topologies has two places where authentication data will need to be inserted. The first is the field named "api_endpoint". This needs to be, minimally, the hostname and port serving the OpenShift cluster API. If the API is behind an additional path element instead of living at the root of the host, this portion can be continued on just as if this is part of a URL fragment.

Secondly, the "api_token" field needs to filled in. This field is time dependent for most users, so it might need to be regenerated on a regular basis. This can be done by executing "oc whoami –token" after an "oc login" command.

OpenShift Instance (Inline)

In this example, the data for a ReplicationController is inserted directly into the topology file. The value under "inline_data" is exactly the same as the data that would be passed into the "oc" command through a file.

```
---
topology_name: openshift
resource_groups:
  - resource_group_name: test1
    res_group_type: openshift
    api_endpoint: example.com:8443
    api_token: someapitoken
    resources:
      - inline_data:
          apiVersion: v1
          kind: ReplicationController
          metadata:
            name: jenkins-slave
            namespace: central-ci-test-ghelling
          spec:
            replicas: 7
            selector:
              name: jenkins-slave
            template:
              metadata:
                labels:
                  name: jenkins-slave
              spec:
                containers:
                  - image: redhatqecinch/jenkins_slave:latest
                    name: jenkins-slave
                    env:
                      - name: JENKINS_MASTER_URL
                        value: http://10.8.172.6/
                      - name: JSLAVE_NAME
                        value: mynode
                restartPolicy: Always
                securityPolicy:
                  runAsUser: 1000090000
```

OpenShift Instance (external)

In this example, the data is not placed into the topology file but a reference to an external yaml file is provided. That file will be read in by Linchpin and uploaded to the OpenShift cluster just as if it had been passed into the "oc" client.

```
---
topology_name: openshift_external
resource_groups:
  - resource_group_name: test-external
    res_group_type: openshift
    api_endpoint: example.com:8443
    api_token: someapitoken
    resources:
      - file_reference: /home/user/openshift/external/resource/file.yaml
      - file_reference: /home/user/openshift/external/resource/cluster.yaml
```

Gcloud Topologies

Topics

Google Cloud Topologies

```
---
topology_name: "ex_gcloud_topo1"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "gcloud"
    res_defs:
      -
        res_name: "testresource"
        flavor: "n1-standard-1"
        res_type: "gcloud_gce"
        region: "us-central1-a"
        image: "centos-7"
        count: 1
    assoc_creds: "gcloudsk"
  -
    resource_group_name: "testgroup2"
    res_group_type: "gcloud"
    res_defs:
      -
        res_name: "testresource2"
        flavor: "n1-standard-1"
        res_type: "gcloud_gce"
        region: "us-central1-a"
        image: "centos-7"
        count: 2
    assoc_creds: "gcloudsk"
resource_group_vars:
  -
    resource_group_name : "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
  -
    resource_group_name : "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp3 hello"
```

Note: Source of the above mentioned examples can be found at Example Topologies

Duffy Topologies

Topics

Simple Duffy Cluster

```
---
topology_name: "duffy_3node_cluster"
resource_groups:
  -
    resource_group_name: "3node"
    res_group_type: "duffy"
    res_defs:
      -
        res_name: "duffy_nodes"
        res_type: "duffy"
        version: 7
        arch: "x86_64"
        count: 3
    assoc_creds: "duffy_creds"
```

Note: the reference to `duffy_creds` defaults to using an assumed file in the user's home directory called `duffy.key`, and points to an internal service at http://admin.ci.centos.org:8080. The credentials themselves are held in the `duffy.key` file.

Beaker Topologies

Topics

Beaker Server

```
---
topology_name: beaker
resource_groups:
  - resource_group_name: test1
```

```
    res_group_type: beaker
    job_group: your-beaker-group
    whiteboard: Arbitrary Job whiteboard string
    recipesets:
      - distro: RHEL-6.5
        arch: x86_64
        keyvalue:
          - MEMORY>1000
          - DISKSPACE>20000
        hostrequires:
          - tag: processors
            op: ">="
            value: 4
          - tag: device
            op: "="
            type: "network"
        count: 1
```

Note: Source of the above Beaker example can be found at Example Topologies

Requiring Specific Hosts

By default, any host available to your beaker user can be selected for use in a given job. If a specific host, or hosts, is desired, `hostrequires` filters can be used to refine the hosts selected for use in a given job.

Force a Specific Host

The reservation of a specific hostname can be done with the `force` keyword nested within a recipeset's `hostrequires` mapping. Additional filtering, such as a `keyvalue` or `hostrequires` filter, is silently ignored by beaker when the hostname to reserve is forced. Because of this, using the `force` argument is mutually exclusive to using any other filters.

For example:

```
hostrequires:
  force: beaker.machine.hostname
```

Select from a named System Pool

Beaker also supports provisioning from a named system pool:

```
hostrequires:
  - tag: pool
    op: "="
    value: system-pool-name
```

This filter will automatically select a system from the named system pool, but unlike the `force` keyword additional filters will also be applied.

Note: The "op" keyword of a hostrequires filter should be quoted when the operator contains symbols, such as "==", "!=", or ">=".

Libvirt Topologies

Topics

Simple Libvirt Topology

```
---
topology_name: "libvirt_simple"
resource_groups:
  -
    resource_group_name: "simple"
    res_group_type: "libvirt"
    res_defs:

      - res_name: "centos72"
        res_type: "libvirt_node"
        driver: 'qemu'
        uri: 'qemu:///system'
        image_src: 'file:///tmp/linchpin_centos71.img'
        count: 2
        memory: 2048
        vcpus: 2
        networks:
          - name: linchpin-centos72

      - res_name: "centos71"
        res_type: "libvirt_node"
        uri: 'qemu:///system'
        count: 1
        image_src: 'http://cloud.centos.org/centos/7/images/CentOS-7-x86_64-
↪GenericCloud-1608.qcow2.xz'
        memory: 2048
        vcpus: 2
        arch: x86_64
        networks:
          - name: linchpin-centos71
```

Note: Each set of nodes can only be assigned one network at this time.

Note: The above topology assumes both networks exist and are running at provision time. If they are not, or do not

exist, they will not be created and will fail.

Complete Libvirt Topology

```
---
topology_name: "libvirt_test"
resource_groups:
  -
    resource_group_name: "libvirt1"
    res_group_type: "libvirt"
    res_defs:

      - res_name: "linchpin-centos72"
        res_type: "libvirt_network"
        ip: 192.168.77.100
        dhcp_start: 192.168.77.101
        dhcp_end: 192.168.77.112

      - res_name: "linchpin-centos74"
        res_type: "libvirt_network"

      - res_name: "centos72"
        res_type: "libvirt_node"
        uri: 'qemu:///system'
        count: 2
        memory: 2048
        vcpus: 2
        networks:
          - name: linchpin-centos72

      - res_name: "centos74"
        res_type: "libvirt_node"
        uri: 'qemu://libvirt.example.com/system'
        memory: 1024
        vcpus: 1
        networks:
          - name: linchpin-centos74
```

Note: as compared with the `simple` topology above, this topology defines and enables the network(s) with the res_type of libvirt_network.

Note: The `linchpin-centos72` network will support dhcp, with a defined pool.

Note: The `linchpin-centos74` is providing only the network definition. Each defined node would need to manually configure its own ip address.

Note: Libvirt provisioning does not yet support `assoc_creds` as simple adjustments can be made to a hypervisor to accommodate authentication.

Hybrid Topologies

Topics

Hybrid Topology1 (os_heat_aws_s3_gce)

```
---
topology_name: "ex_os_heat_aws_s3_gce_topo"
site: "testsite"
resource_groups:
  -
    resource_group_name: "testgroup1"
    res_group_type: "aws"
    res_defs:
      -
        res_name: "ha_inst2"
        flavor: "t2.micro"
        res_type: "aws_ec2"
        region: "us-east-1"
        image: "ami-fce3c696"
        count: 1
        keypair: "sk_key"
      -
        res_name: "samvaranbucktest"
        res_type: "aws_s3"
        region: "us-west-2"
    assoc_creds: "sk_aws_personal"
  -
    resource_group_name: "testgroup2"
    res_group_type: "openstack"
    res_defs:
      - res_name: "ano_inst"
        flavor: "m1.small"
        res_type: "os_server"
        image: "rhel-6.5_jeos"
        count: 1
        keypair: "ci-factory"
        networks:
          - "e2e-openstack"
    assoc_creds: "cios_e2e-openstack"
  -
    resource_group_name: "testgroup3"
    res_group_type: "gcloud"
    res_defs:
      -
        res_name: "testresourcesme"
        flavor: "n1-standard-1"
        res_type: "gcloud_gce"
        region: "us-central1-a"
        image: "debian-8"
        count: 1
```

```
    assoc_creds: "gcloudsk"
  -
    resource_group_name: "testgroup4"
    res_group_type: "openstack"
    res_defs:
      -
        res_name: "os_heat_template_sample"
        res_type: "os_heat"
        template_path: "/root/clients/heat_clients/hot_template_sample2.yaml"
    assoc_creds: "cios_e2e-openstack"
resource_group_vars:
  -
    resource_group_name: "testgroup1"
    Name: "TestInstanceGroup1"
    test_var1: "test_var1 msg is grp1 hello"
    test_var2: "test_var2 msg is grp1 hello"
    test_var3: "test_var3 msg is grp1 hello"
  -
    resource_group_name: "testgroup2"
    Name: "TestInstanceGroup2"
    test_var1: "test_var1 msg is grp2 hello"
    test_var2: "test_var2 msg is grp2 hello"
    test_var3: "test_var3 msg is grp2 hello"
  -
    resource_group_name: "testgroup3"
    Name: "TestInstanceGroup3"
    test_var1: "test_var1 msg is grp3 hello"
    test_var2: "test_var2 msg is grp3 hello"
    test_var3: "test_var3 msg is grp3 hello"
  -
    resource_group_name: "testgroup4"
    Name: "TestInstanceGroup4"
    heat_params:
      key_name: "ci-factory"
      image_id: "rhel-6.5_jeos"
      instance_type: "m1.small"
      network_name: "e2e-openstack"
```

Note: Source of the above mentioned examples can be found at Example Topologies

oVirt Topologies

Topics

oVirt Virtual Machines

```
---
topology_name: "oVirt_vms_example_topology"
resource_groups:
  -
    resource_group_name: "golden_env_mixed"
    resource_group_type: "ovirt"
    resource_definitions:
      -
        res_name: "virtio_1_0"
        res_type: "ovirt_vms"
        template: "golden_mixed_virtio_template"
        cluster: "golden_env_mixed_1"
      -
        res_name: "virtio_1_1"
        res_type: "ovirt_vms"
        template: "golden_mixed_virtio_template"
        cluster: "golden_env_mixed_1"

    credentials:
      filename: "ex_ovirt_creds.yml"
      profile: "ge2"
```

Note: Source of the above mentioned examples can be found at Example Topologies

Python API Reference

This page contains the list of project's modules

linchpin module

The linchpin module contains calls to implement the Command Line Interface within linchpin. It uses the Click command line interface composer.

linchpin.**init**()
Initializes a linchpin project, which generates an example PinFile, and creates the necessary directory structure for topologies and layouts.

Parameters **ctx** – Context object defined by the click.make_pass_decorator method

linchpin.**up**()
Provisions nodes from the given target(s) in the given PinFile.

Parameters

- **ctx** – Context object defined by the click.make_pass_decorator method
- **pinfile** – path to pinfile (Default: ctx.workspace)
- **targets** – Provision ONLY the listed target(s). If omitted, ALL targets in the appropriate PinFile will be provisioned.

linchpin.**rise**()
DEPRECATED. Use 'up'

linchpin.**destroy**()
Destroys nodes from the given target(s) in the given PinFile.

Parameters

- **ctx** – Context object defined by the click.make_pass_decorator method
- **pinfile** – path to pinfile (Default: ctx.workspace)
- **targets** – Destroy ONLY the listed target(s). If omitted, ALL targets in the appropriate PinFile will be destroyed.

linchpin.**drop**()
DEPRECATED. Use 'destroy'.

There are now two functions, *destroy* and *down* which perform node teardown. The *destroy* functionality is the default, and if drop is used, will be called.

The *down* functionality is currently unimplemented, but will shutdown and preserve instances. This feature will only work on providers that support this option.

linchpin.api module

This page contains the list of project's modules

class linchpin.api.**LinchpinAPI**(*ctx*)

__init__(*ctx*)
LinchpinAPI constructor

Parameters **ctx** – context object from api/context.py

lp_up(*pinfile*, *targets='all'*)
This function takes a list of targets, and provisions them according to their topology. If an layout argument is provided, an inventory will be generated for the provisioned nodes.

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to provision.

lp_destroy(*pinfile*, *targets='all'*)
This function takes a list of targets, and performs a destructive teardown, including undefining nodes, according to the target.

See also:

lp_down - currently unimplemented

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to destroy.

lp_down(*pinfile*, *targets='all'*)
This function takes a list of targets, and performs a shutdown on nodes in the target's topology. Only providers which support shutdown from their API (Ansible) will support this option.

CURRENTLY UNIMPLEMENTED

See also:

lp_destroy

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to provision.

run_playbook(*pinfile*, *targets=[]*, *playbook='up'*)
This function takes a list of targets, and executes the given playbook (provison, destroy, etc.) for each provided target.

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to run. (default: 'all')

find_topology(*topology*)
Find the topology to be acted upon. This could be pulled from a registry.

Parameters **topology** – name of topology from PinFile to be loaded

get_cfg(*section=None*, *key=None*, *default=None*)
Get cfgs value(s) by section and/or key, or the whole cfgs object

Parameters

- **section** – section from ini-style config file
- **key** – key to get from config file, within section
- **default** – default value to return if nothing is found.

Does not apply if section is not provided.

set_cfg(*section*, *key*, *value*)
Set a value in cfgs. Does not persist into a file, only during the current execution.

Parameters

- **section** – section within ini-style config file
- **key** – key to use
- **value** – value to set into section within config file

get_evar(*key=None*, *default=None*)
Get the current evars (extra_vars)

Parameters

- **key** – key to use
- **default** – default value to return if nothing is found

(default: None)

set_evar(*key*, *value*)
Set a value into evars (extra_vars). Does not persist into a file, only during the current execution.

Parameters

- **key** – key to use
- **value** – value to set into evars

lp_rise (*pinfile*, *targets='all'*)
DEPRECATED

An alias for lp_up. Used only for backward compatibility.

lp_drop (*pinfile*, *targets*)
DEPRECATED

An alias for lp_destroy. Used only for backward compatibility.

class linchpin.api.context.**LinchpinContext**
LinchpinContext object, which will be used to manage the cli, and load the configuration file.

get_cfg (*section=None*, *key=None*, *default=None*)
Get cfgs value(s) by section and/or key, or the whole cfgs object

Parameters

- **section** – section from ini-style config file
- **key** – key to get from config file, within section
- **default** – default value to return if nothing is found.

Does not apply if section is not provided.

get_evar (*key=None*, *default=None*)
Get the current evars (extra_vars)

Parameters

- **key** – key to use
- **default** – default value to return if nothing is found

(default: None)

load_config (*lpconfig=None*)
Create self.cfgs from the linchpin configuration file.

Note: Overrides load_config in linchpin.api.LinchpinContext

These are the only hardcoded values, which are used to find the config file. The search path consists of the following:

```
* /linchpin/library/path/linchpin.conf
* /etc/linchpin.conf
* ~/.config/linchpin/linchpin.conf
* path/to/workspace/linchpin.conf
```

Linchpin will continuously override and extend the configuration as newer configurations are added and modified. Alternatively, a full path to the linchpin configuration file can be passed.

Parameters **lpconfig** – absolute path to a linchpin config (default: None)

load_global_evars ()
Instantiate the evars variable, then load the variables from the 'evars' section in linchpin.conf. This will then be passed to invoke_linchpin, which passes them to the Ansible playbook as needed.

log (*msg*, ***kwargs*)
Logs a message to a logfile

Parameters

- **msg** – message to output to log
- **level** – keyword argument defining the log level

log_debug(*msg*)
Logs a DEBUG message

log_info(*msg*)
Logs an INFO message

log_state(*msg*)
Logs nothing, just calls pass

> **Attention:** state messages need to be implemented in a subclass

pinfile
getter function for pinfile name

set_cfg(*section*, *key*, *value*)
Set a value in cfgs. Does not persist into a file, only during the current execution.

Parameters

- **section** – section within ini-style config file
- **key** – key to use
- **value** – value to set into section within config file

set_evar(*key*, *value*)
Set a value into evars (extra_vars). Does not persist into a file, only during the current execution.

Parameters

- **key** – key to use
- **value** – value to set into evars

setup_logging()
Setup logging to the console only

> **Attention:** Please implement this function in a subclass

workspace
getter function for workspace

linchpin.api.utils.**yaml2json**(*pf*)
parses yaml file into json object

linchpin.cli module

This page contains the list of project's modules

class linchpin.cli.**LinchpinCli**(*ctx*)

__init__(*ctx*)
Set some variables, pass to parent class

lp_up (*pinfile*, *targets='all'*)
This function takes a list of targets, and provisions them according to their topology. If an layout argument is provided, an inventory will be generated for the provisioned nodes.

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to provision.

lp_destroy (*pinfile*, *targets='all'*)
This function takes a list of targets, and performs a destructive teardown, including undefining nodes, according to the target.

See also:

lp_down - currently unimplemented

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to destroy.

lp_down (*pinfile*, *targets='all'*)
This function takes a list of targets, and performs a shutdown on nodes in the target's topology. Only providers which support shutdown from their API (Ansible) will support this option.

CURRENTLY UNIMPLEMENTED

See also:

lp_destroy

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to provision.

run_playbook (*pinfile*, *targets=[]*, *playbook='up'*)
This function takes a list of targets, and executes the given playbook (provison, destroy, etc.) for each provided target.

Parameters

- **pinfile** – Provided PinFile, with available targets,
- **targets** – A tuple of targets to run. (default: 'all')

find_topology (*topology*)
Find the topology to be acted upon. This could be pulled from a registry.

Parameters **topology** – name of topology from PinFile to be loaded

get_cfg (*section=None*, *key=None*, *default=None*)
Get cfgs value(s) by section and/or key, or the whole cfgs object

Parameters

- **section** – section from ini-style config file
- **key** – key to get from config file, within section
- **default** – default value to return if nothing is found.

Does not apply if section is not provided.

set_cfg(*section*, *key*, *value*)
Set a value in cfgs. Does not persist into a file, only during the current execution.

Parameters

- **section** – section within ini-style config file
- **key** – key to use
- **value** – value to set into section within config file

get_evar(*key=None*, *default=None*)
Get the current evars (extra_vars)

Parameters

- **key** – key to use
- **default** – default value to return if nothing is found

(default: None)

set_evar(*key*, *value*)
Set a value into evars (extra_vars). Does not persist into a file, only during the current execution.

Parameters

- **key** – key to use
- **value** – value to set into evars

lp_rise(*pinfile*, *targets='all'*)
DEPRECATED

An alias for lp_up. Used only for backward compatibility.

lp_drop(*pinfile*, *targets*)
DEPRECATED

An alias for lp_destroy. Used only for backward compatibility.

class linchpin.cli.context.**LinchpinCliContext**
Context object, which will be used to manage the cli, and load the configuration file

get_cfg(*section=None*, *key=None*, *default=None*)
Get cfgs value(s) by section and/or key, or the whole cfgs object

Parameters

- **section** – section from ini-style config file
- **key** – key to get from config file, within section
- **default** – default value to return if nothing is found.

Does not apply if section is not provided.

get_evar(*key=None*, *default=None*)
Get the current evars (extra_vars)

Parameters

- **key** – key to use
- **default** – default value to return if nothing is found

(default: None)

load_config(*lpconfig=None*)

load_global_evars()
Instantiate the evars variable, then load the variables from the 'evars' section in linchpin.conf. This will then be passed to invoke_linchpin, which passes them to the Ansible playbook as needed.

log(*msg*, ***kwargs*)
Logs a message to a logfile or the console

Parameters

- **msg** – message to log
- **lvl** – keyword argument defining the log level
- **msg_type** – keyword argument giving more flexibility.

Note: Only msg_type *STATE* is currently implemented.

log_debug(*msg*)
Logs a DEBUG message

log_info(*msg*)
Logs an INFO message

log_state(*msg*)
Logs a message to stdout

pinfile
getter function for pinfile name

set_cfg(*section*, *key*, *value*)
Set a value in cfgs. Does not persist into a file, only during the current execution.

Parameters

- **section** – section within ini-style config file
- **key** – key to use
- **value** – value to set into section within config file

set_evar(*key*, *value*)
Set a value into evars (extra_vars). Does not persist into a file, only during the current execution.

Parameters

- **key** – key to use
- **value** – value to set into evars

setup_logging()
Setup logging to a file, console, or both. Modifying the *linchpin.conf* appropriately will provide functionality.

workspace
getter function for workspace

Glossary

The following is a list of terms used throughout the LinchPin documentation.

async *(boolean, default: False)*

Used to enable asynchronous provisioning/teardown

async_timeout *(int, default: 1000)*

How long the resouce collection (formerly outputs_writer) process should wait

check_mode *(boolean, default: no)*

This option does nothing at this time, though it may eventually be used for dry-run functionality based upon the provider

default_schemas_path *(file_path, default: <lp_path>/defaults/<schemas_folder>)*

default path to schemas, absolute path. Can be overridden by passing schema / schema_file.

default_playbooks_path *(file_path, default: <lp_path>/defaults/playbooks_folder>)*

default path to playbooks location, only useful to the linchpin API and CLI

default_layouts_path *(file_path, default: <lp_path>/defaults/<layouts_folder>)*

default path to inventory layout files

default_topologies_path *(file_path, default: <lp_path>/defaults/<topologies_folder>)*

default path to topology files

default_resources_path *(file_path, default: <lp_path>/defaults/<resources_folder>, formerly: outputs)*

default landing location for resources output data

default_inventories_path *(file_path, default: <lp_path>/defaults/<inventories_folder>)*

default landing location for inventory outputs

hook Certan scripts can be called when a particular *hook* has been referenced in the *PinFile*. The currently available hooks are *preup*, *postup*, *predestroy*, and *postdestroy*.

inventory

inventory_file If layout / layout_file is provided, this will be the location of the resulting ansible inventory.

linchpin_config if passed on the command line with `-c/--config`, should be an ini-style config file with linchpin default configurations (see BUILT-INS below for more information)

layout

layout_file YAML definition for providing an ansible (currently) static inventory file, based upon the provided topology.

layouts_folder *(file_path, default: layouts)*

relative path to layouts

lp_path base path for linchpin playbooks and python api

lpconfig `<lp_path>/linchpin.conf`, unless overridden by *linchpin_config*

output *(boolean, default: True, previous: no_output)*

Controls whether resources will be written to the resources_file

PinFile A YAML file consisting of a *topology* and an optional *layout*, among other options. This file is used by the `linchpin` command-line, or Python API to determine what resources are needed for the current action.

playbooks_folder *(file_path, default: provision)*

relative path to playbooks, only useful to the linchpin API and CLI

provider A set of platform actions grouped together, which is provided by an external Ansible module. *openstack* would be a provider.

provision An action taken when resources are to be made available on a particular provider platform. Usually corresponds with the `linchpin up` command.

resources

resources_file File with the resource outputs in a JSON formatted file. Useful for teardown (destroy,down) actions depending on the provider.

schema JSON description of the format for the topology.

(schema_v3, schema_v4 are still available)

schemas_folder *(file_path, default: schemas)*

relative path to schemas

target Specified in the *PinFile*, the *target* references a *topology* and optional *layout* to be acted upon from the command-line utility, or Python API.

teardown An action taken when resources are to be made unavailable on a particular provider platform. Usually corresponds with the `linchpin destroy` command.

topologies_folder *(file_path, default: topologies)*

relative path to topologies

topology

topology_file A set of rules, written in YAML, that define the way the provisioned systems should look after executing linchpin.

Generally, the *topology* and *topology_file* values are interchangeable, except after the file has been processed.

topology_name Within a *topology_file*, the *topology_name* provides a way to identify the set of resources being acted upon.

workspace If provided, the above variables will be adjusted and mapped according to this value. Each path will use the following variables:

```
topology / topology_file = /<topologies_folder>
layout / layout_file = /<layouts_folder>
resources / resources_file = /resources_folder>
inventory / inventory_file = /<inventories_folder>
```

If the `WORKSPACE` environment variable is set, it will be used here. If it is not, this variable can be set on the command line with `-w/--workspace`, and defaults to the location of the PinFile bring provisioned.

Note: schema is not affected by this pathing

See also:

Ansible Variables Ansible Variables

Source Code LinchPin Source Code

User Mailing List Subscribe and participate. A great place for Q&A

irc.freenode.net #linchpin IRC chat channel

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

A

C

D

F

G

H

I

L

O

P

R

S

T

U

W

Y