LinchPin Documentation

Release 1.6.5

Samvaran Kashyap Rallabandi

Dec 03, 2018

Contents

Welcome to the LinchPin documentation!

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

Additionally, LinchPin provides a Python API for managing resources. The cloud management component is backed by Ansible. The front-end API manages the interface between the command line (or other interfaces) and calls to the Ansible API.

This documentation covers LinchPin version (1.6.5). For recent features, see the updated 1.6.5.

CHAPTER 1

Why LinchPin?

LinchPin provides a simple, declarative interface to a repeatable set of resources on cloud providers such as Amazon Web Services, Openstack, and Google Cloud Platform to help improve productivity and performance for you and your team. It's built on top of other proven resources, including Ansible and Python. LinchPin is built with a focus on Continuous Integration and Continuous Delivery tooling, in which its workflow excels.

LinchPin has some very useful features, including inventory generation, hooks, and more. Using these, specific cloud resources can be spun up for testing applications. By creating a single PinFile with your targets in an environment, you can simply run *linchpin up* and have your environment up and configured, ready for you to do your work with very little effort.

1.1 Getting Started

The LinchPin getting started guide will walk you through your first LinchPin project, and show off the basics of the major features LinchPin has to offer.

If you are curious about LinchPin and its features, please read the "*Why LinchPin?*" page.

This getting started guide will use LinchPin with the *dummy* provider. LinchPin can work with many other providers and use cases. After following this tutorial, check out some other providers and use cases.

Before starting, please *install the latest version of LinchPin*.

1.1.1 Installation

Currently, LinchPin can be run from any machine with Python 2.6+ (Python 3.x is currently experimental), and requires Ansible 2.3.1 or newer.

Note: Some providers have additional dependencies. Additional software requirements can be found in the *Providers* documentation.

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 2.3.1 or newer, 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
* gcc
* redhat-rpm-config
* libxml2-python
* libxslt-python
```

For CentOS or RHEL the following packages should be installed:

```
$ sudo yum install python-pip python-virtualenv libffi-devel \
openssl-devel libyaml-devel gmp-devel libselinux-python make \
gcc redhat-rpm-config libxml2-python libxslt-python
```

Attention: CentOS 6 (and likely RHEL 6) require special care during installation. See centos6_install for more detail.

For Fedora 26+ the following packages should be installed:

```
$ sudo dnf install python-virtualenv libffi-devel \
openssl-devel libyaml-devel gmp-devel libselinux-python make \
gcc redhat-rpm-config libxml2-python libxslt-python
```

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..
```

Using mkvirtualenv with Python 3 (now default on some Linux systems) will attempt to link to the *python3* binary. LinchPin isn't fully compatible with Python 3 yet. However, mkvirtualenv provides the `-p` option for specifying the *python2* binary.

```
$ mkvirtualenv linchpin -p $(which python2)
..snip..
(linchpin) $ pip install linchpin
..snip..
```

Note: mkvirtualenv is optional dependency you can install from here. An alternative, virtualenv, also exists. Please refer to the virtualenv documentation for more details.

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]
```

Virtual Environments and SELinux

When using a virtualenv with SELinux enabled, LinchPin may fail due to an error related to with the libselinux-python libraries. This is because the libselinux-python binary needs to be enabled in the Virtual Environment. Because this library affects the filesystem, it isn't provided as a standard python module via pip. The RPM must be installed, then a symlink must occur.

```
(linchpin) $ sudo dnf install libselinux-python
.. snip ..
(linchpin) $ echo ${VIRTUAL_ENV}
/path/to/virtualenvs/linchpin
(linchpin) $ export VENV_LIB_PATH=lib/python2.7/site-packages
(linchpin) $ export LIBSELINUX_PATH=/usr/lib64/python2.7/site-packages # make sure to␣
↪verify this location
(linchpin) $ ln -s ${LIBSELINUX_PATH}/selinux ${VIRTUAL_ENV}/${VENV_LIB_PATH}
(linchpin) $ ln -s ${LIBSELINUX_PATH}/_selinux.so ${VIRTUAL_ENV}/${VENV_LIB_PATH}
```

Note: A script is provided to do this work at scripts/install_selinux_venv.sh

Installing on Fedora 26

Install RPM pre-reqs

```
$ sudo dnf -y install python-virtualenv libffi-devel openssl-devel libyaml-devel gmp-
↪devel libselinux-python make gcc redhat-rpm-config libxml2-python
```

Create a working-directory

```
$ mkdir mywork
$ cd mywork
```

Create linchpin directory, make a virtual environment, activate the virtual environment

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

Make a workspace, and initialize it to prove that linchpin itself works

```
(linchpin) $ mkdir workspace
(linchpin) $ cd workspace
(linchpin) $ linchpin init
PinFile and file structure created at /home/user/workspace
```

Note: The default workspace is $PWD, but can be set using the $WORKSPACE variable.

Installing on RHEL 7.4

Tested on RHEL 7.4 Server VM which was kickstarted and pre-installed with the following YUM package-groups and RPMs:

```
* @core
* @base
* vim-enhanced
* bash-completion
* scl-utils
* wget
```

For RHEL 7, it is assumed that you have access to normal RHEL7 YUM repos via RHSM or by pointing at your own http YUM repos, specifically the following repos or their equivalents:

```
* rhel-7-server-rpms
* rhel-7-server-optional-rpms
```

Install pre-req RPMs via YUM:

```
$ sudo yum install -y libffi-devel openssl-devel libyaml-devel gmp-devel libselinux-
↪python make gcc redhat-rpm-config libxml2-devel libxslt-devel libxslt-python␣
↪libxslt-python
```

To get a working python 2.7 pip and virtualenv either use EPEL

```
$ sudo rpm -ivh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Install python pip and virtualenv:

```
$ sudo yum install -y python2-pip python-virtualenv
```

Create a working-directory

```
$ mkdir mywork
$ cd mywork
```

Create linchpin directory, make a virtual environment, activate the virtual environment

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

Inside the virtualenv, upgrade pip and setuptools because the EPEL versions are too old.

```
(linchpin) $ pip install -U pip
(linchpin) $ pip install -U setuptools
```

Install linchpin

```
(linchpin) $ pip install linchpin
```

Make a workspace, and initialize it to prove that linchpin itself works

```
(linchpin) $ mkdir workspace
(linchpin) $ cd workspace
(linchpin) $ linchpin init
PinFile and file structure created at /home/user/workspace
```

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.

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

linchpin setup : Automatic Dependency installation:

From version 1.6.5 linchpin includes linchpin setup commandline option to automate installations of linchpin dependencies. linchpin setup uses built in ansible-playbooks to carryout the installations.

Usage: .. code-block:: bash

    $ linchpin setup # by default linchpin installs all the dependencies

1.1.2 LinchPin Initialization

```
$ linchpin init simple
Created destination workspace: /tmp/simple
$ cd /tmp/simple
$ linchpin up

.. snip ..

Action 'up' on Target 'simple' is complete

ID: 1
Action: up

Target                       Run ID  uHash   Exit Code
-------------------------------------------------
simple                           1  7735aa          0
```

After running the commands above, LinchPin should be able to provision for you. We'll use *linchpin init* and *linchpin fetch* throughout this tutorial to get you familiar with its inner workings.

It's a minimal setup, using the *dummy* provider. We'll get more into those in the upcoming parts of this tutorial.

Now that *LinchPin* is working, the simple workspace is in place, let's learn more about *Workspaces*.

Note: If you were unable to get LinchPin successfully installed and/or working, please see the troubleshooting documentation.

1.1.3 Workspaces

What is generated is commonly referred to as the *workspace*. The workspace can live anywhere on the filesystem. The default is the current directory. The workspace can also be passed into the `linchpin` command line with the `--workspace (--w)` option, or it can be set with the `$WORKSPACE` environmental variable.

In our *simple* example, the workspaces is */tmp/simple*.

A workspace requires only one file, the *PinFile*. This file is the cornerstone to LinchPin provisioning. It's a YAML file, written with declarative syntax. This means the *PinFile* is written to explain how things should be provisioned *after* running *linchpin up*.

Looking at the simple workspace, you'll see that it has a few other items.

```
$ pwd
/tmp/simple
$ ls
inventories  PinFile  PinFile.json  README.rst  resources
```

Ignoring everything but the *PinFile* for now, it's clear that other files and directories will exist in a workspace. Let's have a closer look at the components of a *PinFile*.

1.1.4 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.

The PinFile in the *simple* workspace is shown below.

```
1   ---
2   simple:
3       topology:
4         topology_name: simple
5         resource_groups:
6           - resource_group_name: simple
7             resource_group_type: dummy
8             resource_definitions:
9               - name: web
10                role: dummy_node
11                count: 2
```

The *PinFile* collects the given *topology* and *layout* into one place. It's grouped together in a *target*.

Note: Each of the lines of this PinFile are numbered to help identify lines discussed throughout this section. Each will be denoted with a superscript[1] next to its description.

Target

In this *PinFile*, the target[2] is the first line *simple*, just like the name of the workspace. The target is what LinchPin performs actions upon. For instance, typing `linchpin up` causes the PinFile to be read, and all targets evaluated. The *simple* target would be found, and then the resources listed would be provisioned.

A target will have subcomponents, which tell *linchpin* what it should do and how. Currently, those are *topology*, *layout*, and hooks. For now, we will just cover the topology and its components.

Topology

A topology[3] consists of several items. First and foremost is the topology_name[4], followed by one or more resource_groups[5]. In this PinFile, there is only one resource group.

Resource Group

A resource group contains several items, minimally, it will have a resource_group_name[6], and a resource_group_type[7]. The main component of a resource group, it its resource_definitions[8] section.

Resource Definitions

Within a resource group, multiple resource definitions can exist. In many cases, there are desires for two different resources to be provisioned within a resource group. In this example, there is only one. Each provider has its own constraints for what is required. There are some common fields, however. In the example above, there is name[9], role[10], and count[11].

Note: The role relates to the ansible role used to perform provisioning. In this case, that's the *dummy_node* role. But many providers have multiple roles.

Definitions help, but lets see it in *action*.

Note: More detail about the PinFile can be found in the *PinFiles* document.

1.1.5 Up

It's time to provision your first LinchPin resources.

```
[/tmp/simple]$ linchpin up
 [WARNING]: Unable to parse /tmp/simple/localhost as an inventory source

 [WARNING]: No inventory was parsed, only implicit localhost is available

Action 'up' on Target 'simple' is complete

ID: 10
Action: up

Target                  Run ID  uHash   Exit Code
-------------------------------------------------
simple                       2  3a4038          0
```

In just a few seconds, the command will finish. Because the *simple* target provides only the *dummy_node* resource, no actual instances are provisioned. However, a representation can be found at `/tmp/dummy.hosts`

```
$ cat /tmp/dummy.hosts
web-3a4038-0.example.net
web-3a4038-1.example.net
```

More importantly, there are several other things to note. First off, The `linchpin` command has two basic actions, *up* and *destroy*. Each should be pretty self-explanatory.

Summary

Upon completion of every action, there is a summary that is provided. This summary provides details which can be used to repeat the process, or for further reporting with `linchpin journal`. Let's cover the process in detail.

uHash

The Unique-ish Hash, or uHash[8] provides a way for each instance to be unique within a set of resources. The uHash is used throughout LinchPin with reporting, idempotency, inventories, etc. The uHash is configurable, but defaults to a sha256 hash of some unique data, trimmed to 6 characters.

Run ID

The Run ID[8] can be used for idempotency. The Run ID is used for a specific target. Passing `-r <run-id>` to `linchpin up` or `linchpin destroy` along with the target will provide an idempotent up or destroy action.

```
$ linchpin up --run-id 2 simple

.. snip ..
```

(continues on next page)

(continued from previous page)

```
Action 'up' on Target 'simple' is complete

ID: 11
Action: up

Target                        Run ID  uHash   Exit Code
------------------------------------------------------
simple                             3  3a4038          0
```

The thing to notice here is that the uHash is the same here as in the original *up* action above. This provides idempotency when provisioning.

ID

Similar to the Run ID explained above, the Transaction ID, or ID[5], is provided for idempotency. If desired, the entire transaction can be repeated using this value. Unlike the Run ID, however, the Transaction ID can be used to repeat the entire transaction (multiple targets). As with Run ID, passing `-t <tx-id>` will provide idempotent an idempotent up or destroy action.

```
$ linchpin up --tx-id 10

.. snip ..

ID: 12
Action: up

Target                        Run ID  uHash   Exit Code
------------------------------------------------------
simple                             4  3a4038          0
```

Note: All targets are executed when using `-t/--tx-id`. This differs from `-r/--run-id` where only one target can be supplied per Run ID. This is useful when multiple targets are executed from the PinFile.

Exit Code

A common desire is to check the exit code[7]. This is provided as an indicator of the action's success or failure. Commonly, post actions are performed upon resources (eg. configure the system, adding logins, setting up security, etc.)

1.1.6 Destroy

To destroy the previously provisioned resources, use `linchpin destroy`.

```
$ linchpin destroy
 [WARNING]: Unable to parse /tmp/simple/localhost as an inventory source

 [WARNING]: No inventory was parsed, only implicit localhost is available

Action 'destroy' on Target 'simple' is complete
```

(continues on next page)

(continued from previous page)

```
ID: 13
Action: destroy

Target                        Run ID  uHash    Exit Code
-----------------------------------------------
simple                             5  3a4038           0
```

As with `linchpin up`, destroy provides a summary of the action taken. In this case, however, the resources have been terminated and cleaned up. With the *dummy_node* role, the resources are remove from the file.

```
$ cat /tmp/dummy.hosts
$ wc -l /tmp/dummy.hosts
0 /tmp/dummy.hosts
```

This concludes the introduction of the LinchPin getting started tutorial. For more information, see *Providers*.

See also:

Commands (CLI) Linchpin Command-Line Interface

workflow Common LinchPin Workflows

Managing Resources Managing Resources

Providers Providers in Detail

1.2 Documentation

1.2.1 Running LinchPin

This guide will walk you through the basics of using LinchPin. LinchPin is a command-line utility, a Python API, and Ansible playbooks. As this guide is intentionally brief to get you started, a more complete version can be found in the documentation links found to the left in the *index*.

Topics

Running the `linchpin` command

The `linchpin` CLI is used to perform tasks related to managing *resources*. For detail about a specific command, see *Commands (CLI)*.

Getting Help

Getting help from the command line is very simple. Running either `linchpin` or `linchpin --help` will yield the command line help page.

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

  linchpin: hybrid cloud orchestration

Options:
  -c, --config PATH               Path to config file
  -p, --pinfile PINFILE           Use a name for the PinFile different from
                                  the configuration.
  -d, --template-data TEMPLATE_DATA
                                  Template data passed to PinFile template
  -o, --output-pinfile OUTPUT_PINFILE
                                  Write out PinFile to provided location
  -w, --workspace PATH            Use the specified workspace. Also works if
                                  the familiar Jenkins WORKSPACE environment
                                  variable is set
  -v, --verbose                   Enable verbose output
  --version                       Prints the version and exits
  --creds-path PATH               Use the specified credentials path. Also
                                  works if CREDS_PATH environment variable is
                                  set
  -h, --help                      Show this message and exit.
```

(continues on next page)

(continued from previous page)

```
Commands:
  init     Initializes a linchpin project.
  up       Provisions nodes from the given target(s) in...
  destroy  Destroys nodes from the given target(s) in...
  fetch    Fetches a specified linchpin workspace or...
  journal  Display information stored in Run Database...
```

For subcommands, like `linchpin up`, passing the `--help` or `-h` option produces help related to the provided subcommand.

```
$ linchpin up -h
Usage: linchpin up [OPTIONS] TARGETS

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

  targets:    Provision ONLY the listed target(s). If omitted, ALL targets
  in the appropriate PinFile will be provisioned.

  run-id:     Use the data from the provided run_id value

Options:
  -r, --run-id run_id  Idempotently provision using `run-id` data
  -h, --help           Show this message and exit.
```

As can easily be seen, `linchpin up` has additional arguments and options.

Basic Usage

The most basic usage of `linchpin` might be to perform an *up* action. This simple command assumes a *PinFile* in the workspace (current directory by default), with one target *dummy*.

```
$ linchpin up
Action 'up' on Target 'dummy' is complete

Target               Run ID      uHash   Exit Code
-------------------------------------------------
dummy                    75       79b9           0
```

Upon completion, the systems defined in the *dummy* target will be provisioned. An equally basic usage of `linchpin` is the *destroy* action. This command is peformed using the same PinFile and target.

```
$ linchpin destroy
Action 'destroy' on Target 'dummy' is complete

Target               Run ID      uHash   Exit Code
-------------------------------------------------
dummy                    76       79b9           0
```

Upon completion, the systems which were provisioned, are destroyed (or torn down).

Options and Arguments

The most common argument available in `linchpin` is the *TARGET*. Generally, the *PinFile* will have many targets available, but only one or two will be requested.

```
$ linchpin up dummy-new libvirt-new
Action 'up' on Target 'dummy' is complete
Action 'up' on Target 'libvirt' is complete

Target              Run ID     uHash    Exit Code
-------------------------------------------------
dummy                   77      73b1            0
libvirt                 39      dc2c            0
```

In some cases, you may wish to use a different *PinFile*.

```
$ linchpin -p PinFile.json up
Action 'up' on Target 'dummy-new' is complete

Target              Run ID      uHash   Exit Code
-------------------------------------------------
dummy-new           29          c70a            0
```

As you can see, this PinFile had a *target* called `dummy-new`, and it was the only target listed.

Other common options include:

- `--verbose` (`-v`) to get more output
- `--config` (`-c`) to specify an alternate configuration file
- `--workspace` (`-w`) to specify an alternate workspace

Combining Options

The `linchpin` command also allows combinining of general options with subcommand options. A good example of these might be to use the verbose (`-v`) option. This is very helpful in both the `up` and `destroy` subcommands.

```
$ linchpin -v up dummy-new -r 72
using data from run_id: 72
rundb_id: 73
uhash: a48d
calling: preup
hook preup initiated

PLAY [schema check and Pre Provisioning Activities on topology_file] ********

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [common : use linchpin_config if provided] ****************************
```

What can be immediately observed, is that the `-v` option provides more verbose output of a particular task. This can be useful for troubleshooiting or giving more detail about a specitic task. The `-v` option is placed **before** the subcommand. The `-r` option, since it applies directly to the `up` subcommand, it is placed **afterward**. Investigating the `linchpin -help` and `linchpin up --help` can help differentiate if there's confusion.

Common Usage

Verbose Output

```
$ linchpin -v up dummy-new
```

Specify an Alternate PinFile

```
$ linchpin -vp Pinfile.alt up
```

Specify an Alternate Workspace

```
$ export WORKSPACE=/tmp/my_workspace
$ linchpin up libvirt
```

or

```
$ linchpin -vw /path/to/workspace destroy openshift
```

Provide Credentials

```
$ export CREDS_PATH=/tmp/my_workspace
$ linchpin -v up libvirt
```

or

```
$ linchpin -v --creds-path /credentials/path up openstack
```

Note: The value provided to the `--creds-path` option is a directory, NOT a file. This is generally due to the topology containing the filename where the credentials are stored.

Workspaces

Initialization (init)

Running `linchpin init` will generate the *workspace* directory structure, along with an example *PinFile*, *topology*, and *layout* files. Performing the following tasks will generate a simple dummy PinFile, topology, and layout structure.

```
$ pwd
/tmp/workspace
$ linchpin init
PinFile and file structure created at /tmp/workspace
$ tree
.
├── credentials
├── hooks
```

(continues on next page)

(continued from previous page)

```
├── inventories
├── layouts
│   └── dummy-layout.yml
├── PinFile
└── topologies
    └── dummy-topology.yml
```

Resources

With LinchPin, resources are king. Defining, managing, and generating outputs are all done using a declarative syntax. Resources are managed via the *PinFile*. The PinFile can hold two additional files, the *topology*, and *layout*. Linchpin also supports hooks.

Topology

The *topology* is declarative, written in YAML or JSON (v1.5+), and defines how the provisioned systems should look after executing the `linchpin up` command. 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"
        role: "dummy_node"
        count: 1
```

This topology describes a single dummy system that will be provisioned when *linchpin up* is executed. Once provisioned, the resources outputs are stored for reference and later lookup. Additional topology examples can be found in the source code.

Inventory Layout

An *inventory_layout* (or *layout*) is written in YAML or JSON (v1.5+), and defines how the provisioned resources should look in an Ansible static inventory file. The inventory is generated from the resources provisioned by the topology and the layout data. A layout is shown here.

```
---
inventory_layout:
  vars:
    hostname: __IP__
  hosts:
    example-node:
      count: 1
      host_groups:
        - example
```

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. The *all* group always exists, and includes all provisioned hosts.

```
$ cat inventories/dummy_cluster-0446.inventory
[example]
web-0446-0.example.net hostname=web-0446-0.example.net

[all]
web-0446-0.example.net hostname=web-0446-0.example.net
```

Note: A keen observer might notice the filename and hostname are appended with `-0446`. This value is called the *uhash* or unique-ish hash. Most providers allow for unique identifiers to be assigned automatically to each hostname as well as the inventory name. This provides a flexible way to repeat the process, but manage multiple resource sets at the same time.

Advanced layout examples can be found by reading ra_inventory_layouts.

Note: Additional layout examples can be found in the source code.

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.

```
dummy_cluster:
  topology: dummy-topology.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*.

More detail about the PinFile can be found in the *PinFiles* document.

Additional PinFile examples can be found in the source code

Provisioning (up)

Once a *PinFile*, *topology*, and optional *layout* are in place, provisioning can happen. Performing the command `linchpin up` should provision the *resources* and *inventory* files based upon the *topology_name* value. In this case, is `dummy_cluster`.

```
$ linchpin up
target: dummy_cluster, action: up
Action 'up' on Target 'dummy_cluster' is complete

Target              Run ID  uHash       Exit Code
--------------------------------------------------
dummy_cluster       70      0446                0
```

As you can see, the generated inventory file has the right data. This can be used in many ways, which will be covered elsewhere in the documentation.

```
$ cat inventories/dummy_cluster-0446.inventory
[example]
web-0446-0.example.net hostname=web-0446-0.example.net
```

(continues on next page)

(continued from previous page)

```
[all]
web-0446-0.example.net hostname=web-0446-0.example.net
```

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

```
$ cat /tmp/dummy.hosts
web-0446-0.example.net
test-0446-0.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 look up the *resources* and/or *topology* files (depending on the provider) to determine the proper *teardown* procedure. The *dummy* Ansible role does not use the resources, only the topology during teardown.

```
$ linchpin destroy
target: dummy_cluster, action: destroy
Action 'destroy' on Target 'dummy_cluster' is complete

Target               Run ID  uHash       Exit Code
-------------------------------------------------
dummy_cluster        71      0446                0
```

Verify the */tmp/dummy.hosts* file to ensure the records have been removed.

```
$ 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.

Authentication

Some *Providers* require authentication to acquire managing_resources. LinchPin provides tools for these providers to authenticate. The tools are called credentials.

Credentials

Credentials come in many forms. LinchPin wants to let the user control how the credentials are formatted. In this way, LinchPin supports the standard formatting and options for a provider. The only constraints that exist are how to tell LinchPin which credentials to use, and where they credentials data resides. In every case, LinchPin tries to use the data similarly to the way the provider might.

Credentials File

An example credentials file may look like this for aws.

```
$ cat aws.key
[default]
aws_access_key_id=ARYA4IS3THE3NO7FACEB
aws_secret_access_key=0Hy3x899u93G3xXRkeZK444MITtfl668Bobbygls

[herlo_aws1_herlo]
aws_access_key_id=JON6SNOW8HAS7A3WOLF8
aws_secret_access_key=Te4cUl24FtBELL4blowSx9odd0eFp2Aq30+7tHx9
```

See also:

Providers for provider-specific credentials examples.

To use these credentials, the user must tell LinchPin two things. The first is which credentials to use. The second is where to find the credentials data.

Using Credentials

In the topology, a user can specific credentials. The credentials are described by specifying the file, then the profile. As shown above, the filename is 'aws.key'. The user could pick either profile in that file.

```
---
topology_name: ec2-new
resource_groups:
  - resource_group_name: "aws"
    resource_group_type: "aws"
    resource_definitions:
      - name: demo-day
        flavor: m1.small
        role: aws_ec2
        region: us-east-1
        image: ami-984189e2
        count: 1
    credentials:
      filename: aws.key
      profile: default
```

The important part in the above topology is the *credentials* section. Adding credentials like this will look up, and use the credentials provided.

Credentials Location

By default, credential files are stored in the *default_credentials_path*, which is `~/.config/linchpin`.

Hint: The *default_credentials_path* value uses the interpolated default_config_path value, and can be overridden in the linchpin.conf.

The credentials path (or `creds_path`) can be overridden in two ways.

It can be passed in when running the linchpin command.

```
$ linchpin -vvv --creds-path /dir/to/creds up aws-ec2-new
```

Note: The `aws.key` file could be placed in the default_credentials_path. In that case passing `--creds-path` would be redundant.

Or it can be set as an environment variable.

```
$ export CREDS_PATH=/dir/to/creds
$ linchpin -v up aws-ec2-new
```

See also:

Commands (CLI) Linchpin Command-Line Interface

workflow Common LinchPin Workflows

Managing Resources Managing Resources

Providers Providers in Detail

1.2.2 General Configuration

Managing LinchPin requires a few configuration files. Most configurations are stored in the linchpin configuration file.

Note: in versions before 1.5.1, the file was called linchpin.conf. This changed in 1.5.1 due to backward compatibility requirements, and the need to load configuration defaults. The linchpin.conf continues to work as expected.

The settings in this file are loaded automatically as defaults.

However, it's possible to override any setting in linchpin. For the command line shell, three different locations are checked for linchpin.conf files. Files are checked in the following order:

1. `/etc/linchpin.conf`
2. `~/.config/linchpin/linchpin.conf`
3. `/path/to/workspace/linchpin.conf`

The LinchPin configuration parser supports overriding and extending configurations. If linchpin finds the same section and setting in more than one file, the header that was parsed more recently will provide the configuration. In this way user can override default configurations. Commonly, this is done by placing a *linchpin.conf* in the root of the *workspace*.

Adding/Overriding a Section

New in version 1.2.0

Adding a section to the configuration is simple. The best approach is to create a linchpin.conf in the appropriate location from the locations above.

Once created, add a section. The section can be a new section, or it can overwrite an existing section.

```
[lp]
# move the rundb_connection to a global scope
rundb_conn = %(default_config_path)s/rundb/rundb-::mac::.json

module_folder = library
rundb_conn = ~/.config/linchpin/rundb-::mac::.json

rundb_type = TinyRunDB
rundb_conn_type = file
rundb_schema = {"action": "",
                "inputs": [],
                "outputs": [],
                "start": "",
                "end": "",
                "rc": 0,
                "uhash": ""}
rundb_hash = sha256

dateformat = %%m/%%d/%%Y %%I:%%M:%%S %%p
default_pinfile = PinFile
```

Warning: For version 1.5.0 and earlier, if overwriting a section, all entries from the entire section must be updated.

Overriding a configuration item

New in version 1.5.1

Each item within a section can be a new setting, or override a default setting, as shown.

```
[lp]
# move the rundb_connection to a global scope
rundb_conn = ~/.config/linchpin/rundb-::mac::.json
```

As can be plainly seen, the configuration has been updated to use a different path to the `rundb_conn`. This section now uses a user-based RunDB, which can be useful in some scenarios.

Useful Configuration Options

These are some configuration options that may be useful to adjust for your needs. Each configuration option listed here is in a format of `section.option`.

Note: For clarity, this would appear in a configuration file where the section is in brackets (eg. `[section]`) and the option would have a `option = value` set within the section.

---

lp.external_providers_path New in version 1.5.0

Default value: `%(default_config_path)s/linchpin-x`

Providers playbooks can be created outside of the core of linchpin, if desired. When using these external providers, linchpin will use the *external_providers_path* to lookup the playbooks and attempt to run them.

See *Providers* for more information.

lp.rundb_conn New in version 1.2.0

Default value:

- v1.2.0: `/home/user/.config/linchpin/rundb-<macaddress>.json`
- v1.2.2+: `/path/to/workspace/.rundb/rundb.json`

The RunDB is a single json file, which records each transaction involving resources. A *run_id* and *uHash* are assigned, along with other useful information. The *lp.rundb_conn* describes the location to store the RunDB so data can be retrieved during execution.

evars._async Updated in version 1.2.0

Default value: `False`

Previous key name: evars.async

Some providers (eg. openstack, aws, ovirt) support asynchronous provisioning. This means that a topology containing many resources would provision or destroy all at once. LinchPin then waits for responses from these asynchronous tasks, and returns the success or failure. If the amount of resources is large, asynchronous tasks reduce the wait time immensely.

Reason for change: Avoiding conflict with existing Ansible variable.

Starting in Ansible 2.4.x, the *async* variable could not be set internally. The *_async* value is now passed in and sets the Ansible *async* variable to its value.

evars.default_credentials_path Default value: `%(default_config_path)s`

Storing credentials for multiple providers can be useful. It also may be useful to change the default here to point to a given location.

Note: The `--creds-path` option, or `$CREDS_PATH` environment variable overrides this option

---

evars.inventory_file Default value: `None`

If the unique-hash feature is turned on, the default inventory_file value is built up by combining the *workspace* path, *inventories_folder topology_name*, the *uhash*, and the *extensions.inventory* configuration value. The resulting file might look like this:

```
/path/to/workspace/inventories/dummy_cluster-049e.inventory
```

It may be desired to store the inventory without the uhash, or define a completely different structure altogether.

ansible.console Default value: `False`

This configuration option controls whether the output from the Ansible console is printed. In the `linchpin` CLI tool, it's the equivalent of the `-v (--verbose)` option.

1.2.3 Commands (CLI)

This document covers the `linchpin` Command Line Interface (CLI) in detail. Each page contains a description and explanation for each component. For an overview, see *Running the linchpin command*.

linchpin init

Running `linchpin init` will generate the *workspace* directory structure, along with an example *PinFile*, *topology*, and *layout* files. Performing the following tasks will generate a simple dummy PinFile, topology, and layout structure.

```
$ pwd
/tmp/workspace
$ linchpin init
PinFile and file structure created at /tmp/workspace
$ tree
.
├── credentials
├── hooks
├── inventories
├── layouts
│   └── dummy-layout.yml
├── PinFile
└── topologies
    └── dummy-topology.yml
```

linchpin up

Once a *PinFile*, *topology*, and optional *layout* are in place, provisioning can happen. Performing the command `linchpin up` should provision the *resources* and *inventory* files based upon the *topology_name* value. In this case, is `dummy_cluster`.

```
$ linchpin up
target: dummy_cluster, action: up
Action 'up' on Target 'dummy_cluster' is complete

Target               Run ID  uHash       Exit Code
-------------------------------------------------
dummy_cluster        70      0446                0
```

As you can see, the generated inventory file has the right data. This can be used in many ways, which will be covered elsewhere in the documentation.

```
$ cat inventories/dummy_cluster-0446.inventory
[example]
web-0446-0.example.net hostname=web-0446-0.example.net

[all]
web-0446-0.example.net hostname=web-0446-0.example.net
```

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

```
$ cat /tmp/dummy.hosts
web-0446-0.example.net
test-0446-0.example.net
```

linchpin 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 look up the *resources* and/or *topology* files (depending on the provider) to determine the proper *teardown* procedure. The *dummy* Ansible role does not use the resources, only the topology during teardown.

```
$ linchpin destroy
target: dummy_cluster, action: destroy
Action 'destroy' on Target 'dummy_cluster' is complete

Target              Run ID  uHash       Exit Code
-------------------------------------------------
dummy_cluster       71      0446                0
```

Verify the */tmp/dummy.hosts* file to ensure the records have been removed.

```
$ 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 also:

Providers

linchpin journal

Upon completion of any provision (up) or teardown (destroy) task, there's a record that is created and stored in the *RunDB*. The `linchpin journal` command displays data about these tasks.

```
$ linchpin journal --help
Usage: linchpin journal [OPTIONS] TARGETS

  Display information stored in Run Database

  view:       How the journal is displayed

              'target': show results of transactions on listed targets
              (or all if omitted)

              'tx': show results of each transaction, with results
              of associated targets used

  (Default: target)

  count:      Number of records to show per target

  targets:    Display data for the listed target(s). If omitted, the latest
  records for any/all targets in the RunDB will be displayed.

  fields:     Comma separated list of fields to show in the display.
  (Default: action, uhash, rc)

  (available fields are: uhash, rc, start, end, action)

Options:
  --view VIEW          Type of view display (default: target)
  -c, --count COUNT    (up to) number of records to return (default: 3)
```

(continues on next page)

(continued from previous page)

```
  -f, --fields FIELDS  List the fields to display
  -h, --help           Show this message and exit.
```

There are two specific ways to view the data using the journal, by 'target' and 'transactions (tx)'.

Target

The default view, 'target', is displayed using the target. The data displayed to the screen shows the last three (3) tasks per target, along with some useful information.

```
$ linchpin journal --view=target dummy-new

Target: dummy-new
run_id       action       uhash         rc
-----------------------------------------
5              up         0658          0
4         destroy         cf22          0
3              up         cf22          0
```

Note: The 'target' view is the default, making the –view optional.

The target view can show more data as well. Fields (`-f, --fields`) and count (`-c, --count`) are useful options.

```
$ linchpin journal dummy-new -f action,uhash,end -c 5

Target: dummy-new
run_id      action       uhash         end
-----------------------------------------
6              up        cd00   12/15/2017 05:12:52 PM
5              up        0658   12/15/2017 05:10:52 PM
4         destroy        cf22   12/15/2017 05:10:29 PM
3              up        cf22   12/15/2017 05:10:17 PM
2         destroy        6d82   12/15/2017 05:10:06 PM
1              up        6d82   12/15/2017 05:09:52 PM
```

It is simple to see that the output now has five (5) records, each containing the run_id, action, uhash, and end date.

The data here can be used to perform idempotent (repetitive) tasks, like running the *up* action on *run_id: 5* again.

```
$ linchpin up dummy-new -r 6
Action 'up' on Target 'dummy-new' is complete

Target                   Run ID  uHash   Exit Code
-------------------------------------------------
dummy-new                     7   cd00           0
```

What might not be immediately obvious, is that the *uhash* on Run ID: 7 is identical to the run_id: 6 shown in the previous `linchpin journal` output. Essentially, the same task was run again.

Note: If LinchPin is configured with the unique-hash feature, and the provider supports naming, resources can have unique names. These features are turned off by default.

The *destroy* action will automatically look up the last task with an *up* action and destroy it. If other resources are needed to be destroyed, a *run_id* should be passed to the task.

```
$ linchpin destroy dummy-new -r 5
Action 'destroy' on Target 'dummy-new' is complete

Target                   Run ID  uHash   Exit Code
-------------------------------------------------
dummy-new                     8   0658           0
```

Transactions

The transaction view, provides data based upon each transaction.

```
$ linchpin journal --view tx --count 1

ID: 130         Action: up

Target                   Run ID  uHash   Exit Code
-------------------------------------------------
dummy-new                   279   920c           0
libvirt                     121   ef96           0

=================================================
```

In the future, the transaction view will also provide output for these items.

linchpin fetch

The `linchpin fetch` command provides a simple way to access a resource from a remote location. One could simply perform a *git clone*, or use *wget* to download a `workspace`. However, `linchpin fetch` makes this process simpler, and includes some tooling to make the workflow smooth.

```
$ linchpin fetch --help
Usage: linchpin fetch [OPTIONS] REMOTE

  Fetches a specified linchpin workspace or component from a remote location

Options:
  -t, --type TYPE              Which component of a workspace to fetch.
                               (Default: workspace)
  -r, --root ROOT              Use this to specify the location of the
                               workspace within the root url. If root is not
                               set, the root of the given remote will be used.
  --dest DEST                  Workspaces destination, the fetched workspace
                               will be relative to this location. (Overrides
                               -w/--workspace)
  --branch REF                 Specify the git branch. Used only with git
                               protocol (eg. master).
  --git                        Remote is a Git repository (default)
  --web                        Remote is a web directory
  --nocache                    Do not check the cached time, just copy the
                               data to the destination
  -h, --help                   Show this message and exit.
```

linchpin validate

Validate Command

The purpose of the validate command is to determine whether topologies and layouts are syntactically valid. If not, it will provide a list of errors that occured during validation

The command *linchpin validate* looks at the topology and layout files for each target in a given PinFile. If the topology is not valid under the current schema, it will attempt to convert the topology to an older schema and try again. If the topology is still invalid, the command will report the topology and a list of errors found.

Invalid Topologies

Here is a simple PinFile and topology file. The topology file has some errors and will not validate.

```
---
libvirt-new:
   topology: libvirt-new.yml
   layout: libvirt.yml

libvirt:
   topology: libvirt.yml
   layout: libvirt.yml

libvirt-network:
   topology: libvirt-network.yml
```

```
---
topology_name: libvirt-new
resource_groups:
  - resource_group_name: libvirt-new
    resource_group_type: libvirt
    resource_definitions:
      - role: 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: 1
        arch: x86_64
        ssh_key: libvirt
        networks:
          - name: default
            additional_storage: 10G
        cloud_config:
          users:
            - name: herlo
              gecos: Clint Savage
              groups: wheel
              sudo: ALL=(ALL) NOPASSWD:ALL
              ssh-import-id: gh:herlo
              lock_passwd: true
```

```
$ linchpin validate
topology for target 'libvirt-network' is valid

Topology for target 'libvirt-new' does not validate
topology: 'OrderedDict([('topology_name', 'libvirt-new'), ('resource_groups',␣
↪[OrderedDict([('resource_group_name', 'libvirt-new'), ('resource_group_type',
↪'libvirt'), ('resource_definitions', [OrderedDict([('role', 'libvirt_node'), ('uri',
↪ 'qemu:///system'), ('image_src', 'http://cloud.centos.org/centos/7/images/CentOS-7-
↪x86_64-GenericCloud-1608.qcow2.xz'), ('memory', 2048), ('vcpus', '1'), ('arch',
↪'x86_64'), ('ssh_key', 'libvirt'), ('networks', [OrderedDict([('name', 'default'), (
↪'hello', 'world')])]), ('additional_storage', '10G'), ('cloud_config',␣
↪OrderedDict([('users', [OrderedDict([('name', 'herlo'), ('gecos', 'Clint Savage'), (
↪'groups', 'wheel'), ('sudo', 'ALL=(ALL) NOPASSWD:ALL'), ('ssh-import-id', 'gh:herlo
↪'), ('lock_passwd', True)])])])), ('count', 1)])])])])])'
errors:
      res_defs[0][count]: value for field 'count' must be of type 'integer'
      res_defs[0][networks][0][additional_storage]: field 'additional_storage' could␣
↪not be recognized within the schema provided
      res_defs[0][name]: field 'name' is required

topology for target 'libvirt' is valid under old schema
topology for target 'libvirt-network' is valid
```

The *linchpin validate* command can also provide a list of errors against the old schema with the *–old-schema* flag

```
$ linchpin validate --old-schema

Topology for target 'libvirt-new' does not validate
topology: 'OrderedDict([('topology_name', 'libvirt-new'), ('resource_groups',␣
↪[OrderedDict([('resource_group_name', 'libvirt-new'), ('resource_group_type',
↪'libvirt'), ('resource_definitions', [OrderedDict([('role', 'libvirt_node'), ('uri',
↪ 'qemu:///system'), ('image_src', 'http://cloud.centos.org/centos/7/images/CentOS-7-
↪x86_64-GenericCloud-1608.qcow2.xz'), ('memory', 2048), ('vcpus', '1'), ('arch',
↪'x86_64'), ('ssh_key', 'libvirt'), ('networks', [OrderedDict([('name', 'default'), (
↪'hello', 'world')])]), ('additional_storage', '10G'), ('cloud_config',␣
↪OrderedDict([('users', [OrderedDict([('name', 'herlo'), ('gecos', 'Clint Savage'), (
↪'groups', 'wheel'), ('sudo', 'ALL=(ALL) NOPASSWD:ALL'), ('ssh-import-id', 'gh:herlo
↪'), ('lock_passwd', True)])])])), ('count', 1)])])])])])'
errors:
      res_defs[0][networks][0][additional_storage]: field 'additional_storage' could␣
↪not be recognized within the schema provided
      res_defs[0][name]: field 'name' is required

topology for target 'libvirt' is valid under old schema
topology for target 'libvirt-network' is valid
```

As you can see, validation under both schemas result in an error stating that the field *additional_storage* could not be recognized. In this case, there is simply an indentation error. *additional_storage* is a recognized field within *resource_definitions* but not within the *networks* sub-schema. Other times this unrecognized field may be a spelling error. Both fields also flag the missing "name" field, which is required. Both of these errors must be fixed in order for the topology file to validate. Because making *count* a string only results in an error when validating against the old schema, this field does not have to be changed in order for the topology file to pass validation. However, it is best to change it anyway and keep your topology as up-to-date as possible.

Valid Topologies

The topology below has been fixed so that it will validate under the current schema.

```
---
topology_name: libvirt-new
resource_groups:
  - resource_group_name: libvirt-new
    resource_group_type: libvirt
    resource_definitions:
      - role: libvirt_node
        name: centos71
        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: 1
        arch: x86_64
        ssh_key: libvirt
        networks:
          - name: default
        additional_storage: 10G
        cloud_config:
          users:
            - name: herlo
              gecos: Clint Savage
              groups: wheel
              sudo: ALL=(ALL) NOPASSWD:ALL
              ssh-import-id: gh:herlo
              lock_passwd: true
```

If *linchpin validate* is run on a PinFile containing the topology above, this will be the output:

```
$ linchpin validate
topology for target 'libvirt-new' is valid
topology for target 'libvirt' is valid under old schema
topology for target 'libvirt-network' is valid
```

1.2.4 Managing Resources

Resources in LinchPin generally consist of Virtual Machines, Containers, Networks, Security Groups, Instances, and much more. Detailed below are examples of topoologies, layouts, and PinFiles used to manage resources.

PinFiles

These PinFiles represent many combinations of complexity and providers.

PinFiles are processed top to bottom.

YAML

PinFiles written using YAML format:

- PinFile.dummy.yml

- PinFile.openstack.yml
- PinFile.complex.yml

The combined format is only available in v1.5.0+

- PinFile.combined.yml

JSON

New in version 1.5.0

PinFiles written using JSON format.

- PinFile.dummy.json
- PinFile.aws.json
- PinFile.duffy.json
- PinFile.combined.json
- PinFile.complex.json

Jinja2

New in version 1.5.0

These PinFiles are examples of what can be done with templating using Jinja2.

Beaker Template

This template would be processed with a dictionary containing a key named *arches*.

- PinFile.beaker.template

```
$ linchpin -p PinFile.beaker.template \
    --template-data '{ "arches": [ "x86_64", "ppc64le", "s390x" ]}' up
```

Libvirt Template and Data

This template and data can be processed together.

- PinFile.libvirt-mi.template
- Data.libvirt-mi.yml

```
$ linchpin -vp PinFile.libvirt-mi.template \
    --template-data Data.libvirt-mi.yml up
```

Scripts

New in version 1.5.0

Scripts that generate valid JSON output to STDOUT can be processed and used.

- generate_dummy.sh

```
$ linchpin -vp ./scripts/generate_dummy.sh up
```

Output PinFile

New in version 1.5.0

An output file can be created on an up/destroy action. Simply pass the `--output-pinfile` option with a path to a writable file location.

```
$ linchpin --output-pinfile /tmp/Pinfile.out -vp ./scripts/generate_dummy.sh up
..snip..
$ cat /tmp/Pinfile.out
{
    "dummy": {
        "layout": {
            "inventory_layout": {
                "hosts": {
                    "example-node": {
                        "count": 3,
                        "host_groups": [
                            "example"
                        ]
                    }
                },
                "vars": {
                    "hostname": "__IP__"
                }
            }
        },
        "topology": {
            "topology_name": "dummy_cluster",
            "resource_groups": [
                {
                    "resource_group_name": "dummy",
                    "resource_definitions": [
                        {
                            "count": 3,
                            "type": "dummy_node",
                            "name": "web"
                        },
                        {
                            "count": 1,
                            "type": "dummy_node",
                            "name": "test"
                        }
                    ],
                    "resource_group_type": "dummy"
                }
            ]
        }
    }
}
```

Topologies

These topologies represent many combinations of complexity and providers. Topologies process *resource_definitions* top to bottom according to the file.

Topologies have evolved a little and have a slightly different format between versions. However, older versions still work on v1.5.0+ (until otherwise noted).

The difference is quite minor, except in two providers, beaker and openshift.

Topology Format Pre v1.5.0

```
---
topology_name: "dummy_cluster" # topology name
resource_groups:
  - resource_group_name: "dummy"
    resource_group_type: "dummy"
    resource_definitions:
      - name: "web"
        type: "dummy_node" <-- this is called 'type`
        count: 1
```

v1.5.0+ Topology Format

```
---
topology_name: "dummy_cluster" # topology name
resource_groups:
  - resource_group_name: "dummy"
    resource_group_type: "dummy"
    resource_definitions:
      - name: "web"
        role: "dummy_node" <-- this is called 'role`
        count: 1
```

The subtle difference is in the *resource_definitions* section. In the pre-v1.5.0 topology, the key was *type*, in v1.5.0+, the key is *role*.

Note: Pay attention to the callout in the code blocks above.

For details about the differences in beaker and openshift, see ../topology_incompatibilities.

YAML

New in version 1.5.0

Topologies written using YAML format:

- os-server-new.yml
- libvirt-new.yml
- bkr-new.yml

Older topologies, supported in v1.5.0+

- os-server.yml
- libvirt.yml
- bkr.yml

JSON

New in version 1.5.0

Topologies can be written using JSON format.

- dummy.json

Jinja2

New in version 1.5.0

Topologies can be processed as templates using Jinja2.

Jenkins-Slave Template

This topology template would be processed with a dictionary containing one key named *arch*.

- jenkins-slave.j2

The PinFile.jenkins.yml contains the reference to the *jenkins-slave* topology.

```
jenkins-slave:
  topology: jenkins-slave.yml
  layout: jenkins-slave.yml
```

See also:

Pinfile.jenkins.j2

```
$ linchpin -p PinFile.jenkins --template-data '{ "arch": "x86_64" }' up
```

Layouts

Inventory Layouts (or just *layout*) describe what an Ansible inventory might look like after provisioning. A layout is needed because information about the resources provisioned are unknown in advance.

Layouts, like topologies and PinFiles are processed top to bottom according to the file.

YAML

Layouts written using YAML format:

- aws-ec2.yml
- dummy-new.yml

JSON

New in version 1.5.0

Layouts can be written using JSON format.

- gcloud.json

Jinja2

New in version 1.5.0

Topologies can be processed as templates using Jinja2.

Dummy Template

This layout template would be processed with a dictionary containing one key named *node_count*.

- dummy.json

The PinFile.dummy.json contains the reference to the *dummy.json* layout.

```
{
    "dummy": {
        "topology": "dummy.json",
        "layout": "dummy.json"
    }
}
```

See also:

PinFile.dummy.json

```
$ linchpin -p PinFile.dummy.json --template-data '{ "node_count": 2 }' up
```

Advanced layout examples can be found by reading ra_inventory_layouts.

See also:

Providers

1.2.5 Providers

LinchPin has many default providers. This choose-your-own-adventure page takes you through the basics to ensure success for each.

Openstack

The openstack provider manages multiple types of resources.

os_server

Openstack instances can be provisioned using this resource.

- Topology Example
- Ansible module

Note: Currently, the ansible module used is bundled with LinchPin. However, the variables used are identical to the Ansible os_server module, except for adding a `count` option.

Topology Schema

Within Linchpin, the os_server resource_definition has more options than what are shown in the examples above. For each os_server definition, the following options are available.

Parameter	required	type	ansible value	comments
name	true	string	name	
flavor	true	string	flavor	
image	true	string	image	
region	false	string	region	
count	false	integer	count	
keypair	false	string	key_name	
security_groups	false	string	security_groups	
fip_pool	false	string	floating_ip_pools	
nics	false	string	networks	
userdata	false	string	userdata	
volumes	false	list	volumes	
boot_from_volume	false	string	boot_from_volume	
terminate_volume	false	string	terminate_volume	
volume_size	false	string	volume_size	
boot_volume	false	string	boot_volume	

os_obj

Openstack Object Storage can be provisioned using this resource.

- Topology Example
- Ansible module

os_vol

Openstack Cinder Volumes can be provisioned using this resource.

- Topology Example
- Ansible module

os_sg

Openstack Security Groups can be provisioned using this resource.

- Topology Example
- Ansible Security Group module
- Ansible Security Group Rule module

Additional Dependencies

No additional dependencies are required for the Openstack Provider.

Credentials Management

Openstack provides several ways to provide credentials. LinchPin supports some of these methods for passing credentials for use with openstack resources.

LinchPin honors the openstack environment variables such as `$OS_USERNAME`, `$OS_PROJECT_NAME`, etc.

See the openstack documentation cli documentation for details.

Note: No credentials files are needed for this method. When LinchPin calls the openstack provider, the environment variables are automatically picked up by the openstack Ansible modules, and passed to openstack for authentication.

Openstack provides a simple file structure using a file called clouds.yaml, to provide authentication to a particular tenant. A single clouds.yaml file might contain several entries.

```
clouds:
  devstack:
    auth:
      auth_url: http://192.168.122.10:35357/
      project_name: demo
      username: demo
      password: 0penstack
    region_name: RegionOne
  trystack:
    auth:
      auth_url: http://auth.trystack.com:8080/
      project_name: trystack
      username: herlo-trystack-3855e889
      password: thepasswordissecrte
```

Using this mechanism requires that credentials data be passed into LinchPin.

An openstack topology can have a `credentials` section for each resource_group, which requires the filename, and the profile name.

```
---
topology_name: topo
resource_groups:
  - resource_group_name: openstack
    resource_group_type: openstack
    resource_definitions:
```

(continues on next page)

(continued from previous page)

```
    .. snip ..

  credentials:
    filename: clouds.yaml
    profile: devstack
```

Provisioning with credentials uses the `--creds-path` option. Assuming the `clouds.yaml` file was placed in `~/.config/openstack`, and the topology described above, a provision task could occur.

```
$ linchpin -v --creds-path ~/.config/openstack up
```

Note: The `clouds.yaml` could be placed in the default_credentials_path. In that case passing `--creds-path` would be redundant.

Alternatively, the credentials path can be set as an environment variable,

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

Libvirt

The libvirt provider manages two types of resources.

libvirt_node

Libvirt Domains (or nodes) can be provisioned using this resource.

- Topology Example
- Ansible module

Topology Schema

Within Linchpin, the libvirt_node resource_definition has more options than what are shown in the examples above. For each libvirt_node definition, the following options are available.

Parameter	req'd	type	where used	default	comments
role	true	string	role		
name	true	string	module: name		
vcpus	true	string	xml: vcpus		
memory	true	string	xml: memory	1024	
driver	false	string	xml: driver (kvm, qemu)	kvm	
arch	false	string	xml: arch	x86_64	
boot_dev	false	string	xml: boot_dev	hd	
networks	false	list	xml: networks • name (req) • ip • mac	Assigns the domain to a network by name. Each device is named with an incremented value (eth0) **Note:** Network must exist	
image_src	false	string	virt-install		
network_bridge	false	string	virt-install	virbr0	
ssh_key	false	string	role	resource_group_name	
remote_user	false	string	role	ansible_user_id	
cloud_config	false	list	role	http://cloudinit.readthedocs.io is used here	
additional_storage	false	string	role	1G	
uri	false	string	module: uri	qemu:///system	
count	false	string	N/A		

libvirt_network

Libvirt networks can be provisioned. If a libvirt_network is to be used with a *libvirt_node*, it must precede it.

- Topology Example
- Ansible module

Topology Schema

Within Linchpin, the libvirt_network resource_definition has more options than what are shown in the examples above. For each libvirt_network definition, the following options are available.

Parameter	req'd	type	where used	default	comments
role	true	string	role		
name	true	string	module: name		
uri	false	string	module: name	qemu:///system	
ip	true	string	xml: ip		
dhcp_start	false	string	xml: dhcp_start		
dhcp_end	false	string	xml: dhcp_end		
domain	false	string	xml: domain		Automated DNS for guests
forward_mode	false	string	xml: forward	nat	
forward_dev	false	string	xml: forward		
bridge	false	string	xml: bridge		

Note: This resource will not be torn down during a *destroy* action. This is because other resources may depend on the now existing resource.

Additional Dependencies

The libvirt resource group requires several additional dependencies. The following must be installed.

- libvirt-devel
- libguestfs-tools
- python-libguestfs
- libvirt-python
- python-lxml

For a Fedora 26 machine, the dependencies would be installed using dnf.

```
$ sudo dnf install libvirt-devel libguestfs-tools python-libguestfs
$ pip install linchpin[libvirt]
```

Additionally, because libvirt downloads images, certain SELinux libraries must exist.

- libselinux-python

For a Fedora 26 machine, the dependencies would be installed using dnf.

```
$ sudo dnf install libselinux-python
```

If using a python virtual environment, the selinux libraries must be symlinked. Assuming a virtualenv of `~/venv`, symlink the libraries.

```
$ export LIBSELINUX_PATH=/usr/lib64/python2.7/site-packages
$ ln -s ${LIBSELINUX_PATH}/selinux ~/venv/lib/python2.7/site-packages
$ ln -s ${LIBSELINUX_PATH}/_selinux.so ~/venv/lib/python2.7/site-packages
```

Copying Images

New in version 1.5.1

By default, LinchPin manages the libvirt images in a directory that is accessible only by the root user. However, adjustments can be made to allow an unprivileged user to manage Libvirt via LinchPin. These settings can be modified in the linchpin.conf

This configuration adjustment of *linchpin.conf* may work for the unprivileged user *herlo*.

```
[evars]
libvirt_image_path = ~/libvirt/images/
libvirt_user = herlo
libvirt_become = no
```

The directory will be created automatically by LinchPin. However, the user may need additional rights, like group membership to access Libvirt. Please see https://libvirt.org for any additional configurations.

Credentials Management

Libvirt doesn't require credentials via LinchPin. Multiple options are available for authenticating against a Libvirt daemon (libvirtd). Most methods are detailed here. If desired, the uri for the resource can be set using one of these mechanisms.

By default, however, libvirt requires sudo access to use. To allow users without sudo access to provision libvirt instances, run the following commands on the target machine:

1. Create the libvirt group if it does not exist

```
$ getent group | grep libvirt
$ groupadd -g 7777 libvirt
```

2. Add user account to libvirt group

```
$ usermod -aG libvirt <user>
```

3. Edit libvirtd configuration to add group

```
$ cat <<EOF >>/etc/libvirt/libvirtd.conf
unix_sock_group = "libvirt"
unix_sock_rw_perms = "0770"
EOF
```

4. Restart the libvirtd daemon

```
$ systemctl restart libvirtd
```

The next time the user logs in, they will be able to provision libvirt disks without sudo access

Amazon Web Services

The Amazon Web Services (AWS) provider manages multiple types of resources.

aws_ec2

AWS Instances can be provisioned using this resource.

- Topology Example
- Topology Example w/ VPC
- aws_ec2 module

Topology Schema

Within Linchpin, the aws_ec2 resource_definition has more options than what are shown in the examples above. For each aws_ec2 definition, the following options are available.

Parameter	required	type	ansible value	comments
role	true	string	N/A	
name	true	string	instance_tags	name is set as an instance_tag value.
flavor	true	string	instance_type	
image	true	string	image	
region	false	string	region	
count	false	integer	count	
keypair	false	string	key_name	
security_group	false	string / list	group	
vpc_subnet_id	false	string	vpc_subnet_id	
assign_public_ip	false	string	assign_public_ip	

EC2 Inventory Generation

If an instance has a public IP attached, its hostname in public DNS, if available, will be provided in the generated Ansible inventory file, and if not the public IP address will be provided.

For instances which have a private IP address for VPC usage, the private IP address will be provided since private EC2 DNS hostnames (e.g. **ip-10-0-0-1.ec2.internal**) will not typically be resolvable outside of AWS.

For instances with both a public and private IP address, the public address is always provided instead of the private address, so as to avoid duplicate runs of Ansible on the same host via the generated inventory file.

aws_ec2_key

AWS SSH keys can be added using this resource.

- Topology Example
- ec2_key module

Note: This resource will not be torn down during a *destroy* action. This is because other resources may depend on the now existing resource.

---

aws_s3

AWS Simple Storage Service buckets can be provisioned using this resource.

- Topology Example
- aws_s3 module

Note: This resource will not be torn down during a *destroy* action. This is because other resources may depend on the now existing resource.

---

aws_sg

AWS Security Groups can be provisioned using this resource.

- Topology Example
- ec2_group module

Note: This resource will not be torn down during a *destroy* action. This is because other resources may depend on the now existing resource.

Additional Dependencies

No additional dependencies are required for the AWS Provider.

Credentials Management

AWS provides several ways to provide credentials. LinchPin supports some of these methods for passing credentials for use with AWS resources.

One method to provide AWS credentials that can be loaded by LinchPin is to use the INI format that the AWS CLI tool uses.

Credentials File

An example credentials file may look like this for aws.

```
$ cat aws.key
[default]
aws_access_key_id=ARYA4IS3THE3NO7FACEB
aws_secret_access_key=0Hy3x899u93G3xXRkeZK444MITtfl668Bobbygls

[herlo_aws1_herlo]
aws_access_key_id=JON6SNOW8HAS7A3WOLF8
aws_secret_access_key=Te4cUl24FtBELL4blowSx9odd0eFp2Aq30+7tHx9
```

See also:

Providers for provider-specific credentials examples.

To use these credentials, the user must tell LinchPin two things. The first is which credentials to use. The second is where to find the credentials data.

Using Credentials

In the topology, a user can specific credentials. The credentials are described by specifying the file, then the profile. As shown above, the filename is 'aws.key'. The user could pick either profile in that file.

```
---
topology_name: ec2-new
resource_groups:
  - resource_group_name: "aws"
    resource_group_type: "aws"
    resource_definitions:
      - name: demo-day
```

(continues on next page)

(continued from previous page)

```
        flavor: m1.small
        role: aws_ec2
        region: us-east-1
        image: ami-984189e2
        count: 1
    credentials:
      filename: aws.key
      profile: default
```

The important part in the above topology is the *credentials* section. Adding credentials like this will look up, and use the credentials provided.

Credentials Location

By default, credential files are stored in the *default_credentials_path*, which is `~/.config/linchpin`.

Hint: The *default_credentials_path* value uses the interpolated default_config_path value, and can be overridden in the linchpin.conf.

The credentials path (or `creds_path`) can be overridden in two ways.

It can be passed in when running the linchpin command.

```
$ linchpin -vvv --creds-path /dir/to/creds up aws-ec2-new
```

Note: The `aws.key` file could be placed in the default_credentials_path. In that case passing `--creds-path` would be redundant.

Or it can be set as an environment variable.

```
$ export CREDS_PATH=/dir/to/creds
$ linchpin -v up aws-ec2-new
```

Environment Variables

LinchPin honors the AWS environment variables

Provisioning

Provisioning with credentials uses the `--creds-path` option.

```
$ linchpin -v --creds-path ~/.config/aws up
```

Alternatively, the credentials path can be set as an environment variable,

```
$ export CREDS_PATH="~/.config/aws"
$ linchpin -v up
```

Google Cloud Platform

The Google Cloud Platform (gcloud) provider manages one resource, `gcloud_gce`.

gcloud_gce

Google Compute Engine (gce) instances are provisioned using this resource.

- Topology Example
- Ansible module

Additional Dependencies

No additional dependencies are required for the Google Cloud (gcloud) Provider.

Credentials Management

Google Compute Engine provides several ways to provide credentials. LinchPin supports some of these methods for passing credentials for use with openstack resources.

Environment Variables

LinchPin honors the gcloud environment variables.

Configuration Files

Google Cloud Platform provides tooling for authentication. See https://cloud.google.com/appengine/docs/standard/python/oauth/ for options.

Beaker

The Beaker (bkr) provider manages a single resource, `bkr_server`.

bkr_server

Beaker instances are provisioned using this resource.

- Topology Example

The ansible modules for beaker are written and bundled as part of LinchPin.

- bkr_server.py
- bkr_info.py

Topology Schema

Within Linchpin, the bkr_server resource_definition has more options than what are shown in the examples above. For each bkr_server role definition, the following options are available.

Parameter	required	type	ansible value	default
role	true	string	N/A	
whiteboard	false	string	whiteboard	Provisioned by LinchPin
job_group	false	string	job_group	
cancel_message	false	string	cancel_message	
max_attempts	false	string	max_attempts	
attempt_wait_time	false	integer	attempt_wait_time	
recipesets	false	string	recipesets	see table below

recipesets

Because recipesets is how beaker requests systems, it's a large part of what the topology schema includes. There are several ways to request systems. This table describes the available recipesets options.

Parameter	required	type	sub-field layout options		
distro	false	string	N/A		
family	false	string	N/A		
tags	false	list	list of strings		
name	false	string	N/A		
arch	false	string	N/A		
variant	false	string	N/A		
bkr_data	false	string	N/A		
method	false	string	N/A		
count	false	string	N/A		
ids	false	list	N/A		
taskparam	false	list	list of strings		
keyvalue	false	list	list of strings		
hostrequires	false	list	**param**	**required**	**type**
			tag	true	string
			op	false	string
			value	false	int / string
			type	false	string
		dict	force	false	string
		dict	rawxml	false	string
reserve_duration	false	int	N/A		
repos	false	list	dict baseurl		
install	false	list	list of strings		

Additional Dependencies

The beaker resource group requires several additional dependencies. The following must be installed.

- beaker-client>=23.3

It is also recommended to install the python bindings for kerberos.

- python-krbV

For a Fedora 26 machine, the dependencies could be installed using dnf.

```
$ sudo dnf install python-krbV
$ wget https://beaker-project.org/yum/beaker-server-Fedora.repo
$ sudo mv beaker-server-Fedora.repo /etc/yum.repos.d/
$ sudo dnf install beaker-client
```

Alternatively, with pip, possibly within a virtual environment.

```
$ pip install linchpin[beaker]
```

Credentials Management

Beaker provides several ways to authenticate. LinchPin supports these methods.

- Kerberos
- OAuth2

Note: LinchPin doesn't support the username/password authentication mechanism. It's also not recommended by the Beaker Project, except for initial setup.

Duffy

Duffy is a tool for managing pre-provisioned systems in CentOS' CI environment. The Duffy provider manages a single resource, `duffy_node`.

duffy_node

The `duffy_node` resource provides the ability to provision using the duffy api.

- Topology Example

The ansible module for duffy exists in its own repository.

Using Duffy

Duffy can only be run within the CentOS CI environment. To get access, follow this guide. Once access is granted, the duffy ansible module can be used.

Additional Dependencies

Duffy doesn't require any additional dependencies, but does need to be included in the Ansible library path to work properly. See the ansible documentation for help addding a library path.

Credentials Management

Duffy uses a single file, generally found in the user's home directory, to provide credentials. It contains a single line, which has the API key which is passed to duffy via the API.

For LinchPin to provision, `duffy.key` must exist.

A duffy topology can have a `credentials` section for each resource_group, which requires a filename.

```
---
topology_name: topo
resource_groups:
  - resource_group_name: duffy
    resource_group_type: duffy
    resource_definitions:

      .. snip ..

    credentials: duffy.key
```

By default, the location searched for the `duffy.key` is the user's home directory, as stated above. However, the credentials path can be set using `--creds-path` option. Assuming the `duffy.key` file was placed in `~/.config/duffy`, using the topology described above, a provisioning task could occur.

```
$ linchpin -v --creds-path ~/.config/duffy up
```

Alternatively, the credentials path can be set as an environment variable,

```
$ export CREDS_PATH="~/.config/duffy"
$ linchpin -v up
```

oVirt

The ovirt provider manages a single resource, `ovirt_vms`.

ovirt_vms

oVirt Domains/VMs can be provisioned using this resource.

- Topology Example
- Ansible module

Additional Dependencies

There are no known additional dependencies for using the oVirt provider for LinchPin.

Credentials Management

An oVirt topology can have a `credentials` section for each resource_group, which requires the filename, and the profile name.

Consider the following file, named `ovirt_creds.yml`.

```
clouds:
  ge2:
    auth:
      ovirt_url: http://192.168.122.10/
      ovirt_username: demo
      ovirt_password: demo
```

An oVirt topology can have a `credentials` section for each resource_group, which requires the filename and profile name.

```
---
topology_name: topo
resource_groups:
  - resource_group_name: ovirt
    resource_group_type: ovirt
    resource_definitions:

      .. snip ..

    credentials:
      filename: ovirt_creds.yml
      profile: ge2
```

Provisioning

Provisioning with credentials uses the `--creds-path` option. Assuming the credentials file was placed in `~/.config/ovirt`, and the topology described above, a provision task could occur.

```
$ linchpin -v --creds-path ~/.config/ovirt up
```

Alternatively, the credentials path can be set as an environment variable,

```
$ export CREDS_PATH="~/.config/ovirt"
$ linchpin -v up
```

Openshift

The openshift provider manages two resources, `openshift_inline`, and `openshift_external`. However, both of the resource types are managed by module k8s Ansible module. Usage of either one will result in redirection to k8s module with different parameters.

Prior to linchpin 1.6.5, The Ansible module for openshift is written and bundled as part of LinchPin. * openshift.py

After 1.6.5 bundled ansible module is being replaced by upstream ansible kubernetes module. Refer: K8s module. Linchpin supports all the attributes mentioned in k8s module.

openshift_inline

Openshift instances can be provisioned using this resource. Resources are detail inline. * Topology Example

Example PinFile:

openshift_external

Openshift instances can be provisioned using this resource. Resources are detail in an external file.

Example PinFile:

Topology Schema:

openshift_inline and opeshift_external resource definitions in linchpin follow the schema identical to ansible k8s module. The following parameters are allowed in a linchpin topology:

Additional Dependencies

There are no known additional dependencies for using the openshift provider for LinchPin. Since openshift client dependecy is included as part of linchpin's core requirements.

Credentials Management

An openshift topology can have a `credentials` section for each resource_group, which requires the *api_endpoint*, and the *api_token* values. Openshift honors –creds-path in linchpin. The credential file passed needs to be formatted as follows. Further, it also honors all the evironment variables that are supported by ansible k8s module. Refer: K8s module. Linchpin defaults to environment variables if the credentials section is ommited or the –creds-path does not contain the openshift credential file.

```
---
default:
  api_endpoint: https://192.168.42.115:8443
  api_token: 4_6A86rcZqdVBIbPwJQnsz33mO35O_PnSH2okk8_190
# optional parameters
# api_version: v1  # defaults to version 1
# cert_file: /path/to/cert_file
# context: contextname
# key_file: /path/to/key_file
# kube_config: /path/to/kube_config
# ssl_ca_cert: /path/to/ssl_ca_cert
# username: username # not needed when api_token is used
# password: ******** # not needed when api_token is used
# verify_ssl: no #defaults to no. Needs to be set to yes when ssl_ca_cert is used

test:
  api_endpoint: https://192.168.42.115:8443
  api_token: 4_6A86rcZqdVBIbPwJQnsz33mO35O_PnSH2okk8_190
```

```
---
topology_name: topo
resource_groups:
  - resource_group_name: openshift
    resource_group_type: openshift
    resource_definitions:
      - name: openshift
```

(continues on next page)

(continued from previous page)

```
        role: openshift_inline
        definition:

      .. snip ..

    credentials:
      filename: name_of_credsfile.yaml  # fetched from --creds-path is provided
      profile: name_of_profile # defaults to 'default' profile in cred_file
```

Tid bits :

How to get to know API_ENDPOINT and API_TOKEN:

Once the openshift cluster is up and running try logging into openshift using the following command

After login run following command to get the API_ENDPOINT:

Run the following command to get API_TOKEN

Make sure your openshift user has permissions to create resources:

Openshift by default imposes many restrictions on users when it comes to creation . One can always manage roles to get appropriate roles. if its just a development environment please use following command to give admin user privileges to user .. code-block:

```
oc adm policy add-cluster-role-to-user cluster-admin <username> --as=system:admin
```

Refer: Openshift role management.

1.2.6 Advanced Topics

Provisioning in LinchPin is a fairly simple process. However, LinchPin also provides some very flexible and powerful features. These features can sometimes be complex, which means most users will likely not use them. Those features are covered here.

Inventory Layouts

When generating an inventory, LinchPin provides some very flexible options. From the simple *Layouts* to much more complex options, detailed here.

inventory_file

New in version 1.5.2

When an *layout* is provided in the PinFile, LinchPin automatically generates a static inventory for Ansible. The inventory filename is dynamically generated based upon a few factors. However, the value can be overridden simply by adding the `inventory_file` option.

```
---
inventory_layout:
  inventory_file: /path/to/dummy.inventory
  vars:
  .. snip ..
```

Using LinchPin or Ansible variables

New in version 1.5.2

It's likely that the inventory file is based upon specific Linchpin (or Ansible) variables. In this case, the values need to be wrapped as raw values. This allows LinchPin to read the string in unparsed and pass it to the Ansible parser.

```
inventory_layout:
  inventory_file: "{% raw -%}{{ workspace }}/inventories/dummy-new-{{ uhash }}.
↪inventory{%- endraw %}"
```

Using Environment variables

Additionally, using environment variables requires the raw values.

```
host_groups:
  all:
    vars:
      ansible_user: root
      ansible_private_key_file: |
          "{% raw -%}{{ lookup('env', 'TESTLP') | default('/tmp', true) }}/CSS/
↪keystore/css-central{%- endraw %}"
```

The RunDB Explained

Attention: Much of the information below began in v1.2.0 and later. However, much of the data did not exist until later on, generally in version 1.5.0 or later. Some cases, where noted, the data is only planned, and does not yet exist.

The RunDB is the central database which stores transactions and target-based runs each time any LinchPin action is performed. The RunDB stores detailed data, including inputs like topology, inventory layout, hooks; and outputs like resource return data, ansible inventory filename and data, etc.

RunDB Storage

The RunDB is stored using a JSON format by default. TinyDB currently provides the backend. It is a NOSQL database, which writes out transactional records to a single file. Other databases could provide a backend, as long as a driver is written and included.

TinyDB is included in a class called TinyRunDB. TinyRunDB is an implementation of a parent class, called BaseDB, which in turn is a subclass of the abstract RunDB class.

Records are the main way for items to be stored in the RunDB. There are two types of records stored in the RunDB, target, and transaction.

Transaction Records

Each time any action (eg. `linchpin up`) occurs using linchpin, a transaction record is stored. The transaction records are stored in the 'linchpin' table. The main constraint to this is that a target called *linchpin* cannot be used.

Transaction Records consist of a Transaction ID *(tx_id)*, the action and a target information for each target acted upon during the specified transaction. A single record could have multiple targets listed.

```
"136": {
    "action": "up",
    "targets": [
        {
            "dummy-new": {
                "290": {
                    "rc": 0,
                    "uhash": "27e1"
                }
            },
            "libvirt-new": {
                "225": {
                    "rc": 0,
                    "uhash": "d88c"
                }
            }
        }
    ]
},
```

In every case, the target data included is the name, run-id, return code (rc), and uhash. The `linchpin journal` provides a transaction view to show this data in human readable format.

```
$ linchpin journal --view tx -t 136

ID: 136                     Action: up

Target              Run ID  uHash   Exit Code
-------------------------------------------
dummy-new              290   27e1           0
libvirt-new            225   d88c           0

===========================================
```

Target Records

Target Records are much more detailed. Generally, the target records correspond to a specific Run ID (*run_id*). These can also be referenced via the `linchpin journal` command, using the target (default) view.

```
$ linchpin journal dummy-new --view target

Target: dummy-new
run_id          action          uhash       rc
-----------------------------------------------
225                up           f9e5        0
224           destroy           89ea        0
223                up           89ea        0
```

The target record data is where the detail lies. Each record contains several sections, followed by possibly several sub-sections. A complete target record is very large. Let's have a look at record 225 for the 'dummy-new' target.


```
"225": {
    "action": "up",
    "end": "03/27/2018 12:18:21 PM",
    "inputs": [
        {
            "topology_data": {
                "resource_groups": [
                    {
                        "resource_definitions": [
                            {
                                "count": 3,
                                "name": "web",
                                "role": "dummy_node"
                            },
                            {
                                "count": 1,
                                "name": "test",
                                "role": "dummy_node"
                            }
                        ],
                        "resource_group_name": "dummy",
                        "resource_group_type": "dummy"
                    }
                ],
                "topology_name": "dummy_cluster"
            }
        },
        {
            "layout_data": {
                "inventory_layout": {
                    "hosts": {
                        "example-node": {
                            "count": 3,
                            "host_groups": [
                                "example"
                            ]
                        },
                        "test-node": {
                            "count": 1,
                            "host_groups": [
                                "test"
                            ]
                        }
                    },
                    "inventory_file": "{{ workspace }}/inventories/dummy-new-{{ uhash␣
↪}}.inventory",
                    "vars": {
                        "hostname": "__IP__"
                    }
                }
            }
        },
        {
            "hooks_data": {
                "postup": [
```


(continues on next page)

(continued from previous page)

```
                    {
                        "actions": [
                            "echo hello"
                        ],
                        "name": "hello",
                        "type": "shell"
                    }
                ]
            }
        }
    ],
    "outputs": [
        {
            "resources": [
                {
                    "changed": true,
                    "dummy_file": "/tmp/dummy.hosts",
                    "failed": false,
                    "hosts": [
                        "web-f9e5-0.example.net",
                        "web-f9e5-1.example.net",
                        "web-f9e5-2.example.net"
                    ]
                },
                {
                    "changed": true,
                    "dummy_file": "/tmp/dummy.hosts",
                    "failed": false,
                    "hosts": [
                        "test-f9e5-0.example.net"
                    ]
                }
            ]
        }
    ],
    "rc": 0,
    "start": "03/27/2018 12:18:02 PM",
    "uhash": "f9e5",
    "cfgs": [
        {
            "evars": []
        },
        {
            "magics": []
        },
        {
            "user": []
        }
    ]
},
```

As might be gleaned from looking at the JSON, there are a few main sections. Some of these sections, have subsections. The main sections include:

```
* action
* start
```

(continues on next page)

(continued from previous page)

```
* end
* uhash
* rc
* inputs
* outputs
* cfgs
```

Most of these sections are self-explanatory, or can be easily determined. However, there are three that may need further explanation.

Inputs

The RunDB stored all inputs in the "inputs" section.

```
"inputs": [
    {
        "topology_data": {
            "resource_groups": [
                {
                    "resource_definitions": [
                        {
                            "count": 3,
                            "name": "web",
                            "role": "dummy_node"
                        },
                        {
                            "count": 1,
                            "name": "test",
                            "role": "dummy_node"
                        }
                    ],
                    "resource_group_name": "dummy",
                    "resource_group_type": "dummy"
                }
            ],
            "topology_name": "dummy_cluster"
        }
    },
    {
        "layout_data": {
            "inventory_layout": {
                "hosts": {
                    "example-node": {
                        "count": 3,
                        "host_groups": [
                            "example"
                        ]
                    },
                    "test-node": {
                        "count": 1,
                        "host_groups": [
                            "test"
                        ]
                    }
                },
```

(continues on next page)

(continued from previous page)

```
                "inventory_file": "{{ workspace }}/inventories/dummy-new-{{ uhash }}.
↪inventory",
                "vars": {
                    "hostname": "__IP__"
                }
            }
        }
    },
    {
        "hooks_data": {
            "postup": [
                {
                    "actions": [
                        "echo hello"
                    ],
                    "name": "hello",
                    "type": "shell"
                }
            ]
        }
    }
],
```

Currently, the *inputs* section has three sub-sections, *topology_data*, *layout_data*, and *hooks_data*. These three sub-sections hold relevant data. The use of this data is generally for record-keeping, and more recently to allow for reuse of the data with linchpin up/destroy actions.

Additionally, some of this data is used to create the outputs, which are stored in the *outputs* section.

Outputs

Going forward, the *outputs* section will contain much more data than is displayed below. Items like *ansible_inventory*, and *user_data* will also appear in the database. These will be provided in future development.

```
"outputs": [
    {
        "resources": [
            {
                "changed": true,
                "dummy_file": "/tmp/dummy.hosts",
                "failed": false,
                "hosts": [
                    "web-f9e5-0.example.net",
                    "web-f9e5-1.example.net",
                    "web-f9e5-2.example.net"
                ]
            },
            {
                "changed": true,
                "dummy_file": "/tmp/dummy.hosts",
                "failed": false,
                "hosts": [
                    "test-f9e5-0.example.net"
                ]
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    }
],
```

The lone sub-section is *resources*. For the *dummy-new* target, the data provided is simplistic. However, for providers like openstack or aws, the resources become quite large and extensive. Here is a snippet of an openstack resources sub-section.

```
"resources": [
    {
        "changed": true,
        "failed": false,
        "ids": [
            "fc96e134-4a68-4aaa-a053-7f53cae21369"
        ],
        "openstack": [
            {
                "OS-DCF:diskConfig": "MANUAL",
                "OS-EXT-AZ:availability_zone": "nova",
                "OS-EXT-STS:power_state": 1,
                "OS-EXT-STS:task_state": null,
                "OS-EXT-STS:vm_state": "active",
                "OS-SRV-USG:launched_at": "2017-11-27T19:43:54.000000",
                "OS-SRV-USG:terminated_at": null,
                "accessIPv4": "10.8.245.175",
                "accessIPv6": "",
                "addresses": {
                    "atomic-e2e-jenkins-test": [
                        {
                            "OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:ba:0e:5e",
                            "OS-EXT-IPS:type": "fixed",
                            "addr": "172.16.171.15",
                            "version": 4
                        },
                        {
                            "OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:ba:0e:5e",
                            "OS-EXT-IPS:type": "floating",
                            "addr": "10.8.245.175",
                            "version": 4
                        }
                    ]
                },
                "adminPass": "<REDACTED>",
                "az": "nova",
                "cloud": "",
                "config_drive": "",
                "created": "2017-11-27T19:43:47Z",
                "disk_config": "MANUAL",
                "flavor": {
                    "id": "2",
                    "name": "m1.small"
                },
                "has_config_drive": false,
                "hostId": "20a84eb5691c546defeac6b2a5b4586234aed69419641215e0870a64",
                "host_id": "20a84eb5691c546defeac6b2a5b4586234aed69419641215e0870a64
↪",
```

(continues on next page)

(continued from previous page)


```
 "id": "fc96e134-4a68-4aaa-a053-7f53cae21369",
"image": {
    "id": "eae92800-4b49-4e81-b876-1cc61350bf73",
    "name": "CentOS-7-x86_64-GenericCloud-1612"
},
"interface_ip": "10.8.245.175",
"key_name": "ci-factory",
"launched_at": "2017-11-27T19:43:54.000000",
"location": {
    "cloud": "",
    "project": {
        "domain_id": null,
        "domain_name": null,
        "id": "6e65fbc3161648e78fde849c7abbd30f",
        "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
    },
    "region_name": "",
    "zone": "nova"
},
"metadata": {},
"name": "database-44ee-1",
"networks": {},
"os-extended-volumes:volumes_attached": [],
"power_state": 1,
"private_v4": "172.16.171.15",
"progress": 0,
"project_id": "6e65fbc3161648e78fde849c7abbd30f",
"properties": {
    "OS-DCF:diskConfig": "MANUAL",
    "OS-EXT-AZ:availability_zone": "nova",
    "OS-EXT-STS:power_state": 1,
    "OS-EXT-STS:task_state": null,
    "OS-EXT-STS:vm_state": "active",
    "OS-SRV-USG:launched_at": "2017-11-27T19:43:54.000000",
    "OS-SRV-USG:terminated_at": null,
    "os-extended-volumes:volumes_attached": []
},
"public_v4": "10.8.245.175",
"public_v6": "",
"region": "",
"security_groups": [
    {
        "description": "Default security group",
        "id": "1da85eb2-3c51-4729-afc4-240e187a30ce",
        "location": {
            "cloud": "",
            "project": {
                "domain_id": null,
                "domain_name": null,
                "id": "6e65fbc3161648e78fde849c7abbd30f",
                "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
            },
.. snip ..
```


Note: The data above continues for several more pages, and would take up too much space to document. A savvy

user might cat the rundb file and pipe it to the python 'json.tool' module.

Each provider returns a large structure like this as results of the provisioning (up) process. For the teardown, the data can be large, but is generally more succinct.

Context Distiller

New in version 1.5.2

The purpose of the Context Distiller is to take outputs from provisioned resources and provide them to a user as a json file.

The distiller currently supports the following roles:

```
* os_server
* aws_ec2
* bkr_server
* dummy_node (for testing)
```

For each role, the distiller collects specific fields from the resource data.

Note: Please be aware that this feature is planned to integrated with other tooling to make extracting resource data more flexible in the future.

Enabling the Distiller

To enable the Context Distiller, the following must be set in the linchpin.conf.

```
[lp]
distill_data = True

# disable generating the resources file
[evars]
generate_resources = False
```

Note: Other settings may already be in these sections. If that is the case, just add these settings to the proper section.

Hint: It may not be immediately obvious, as LinchPin uses the *RunDB* data to return resource data from a run. In this way, the resource data can be stored somewhere and retrieved at any time by future tooling. Because of this, the resources file is disabled. In this way, the resource data is stored solely in the RunDB for easy retrieval.

Fields to Retreive

Warning: Modifying the distilled fields can cause unexpected results. MODIFY THIS DATA AT YOUR OWN RISK!

Within the linchpin.constants file, the *[distiller]* section exists. Described within this section is how each role gathers the applicable data to distill.

```
[distiller]
bkr_server = id,url,system
dummy_node: hosts
aws_ec2 = instances.id,instances.public_ip,instances.private_ip,instances.public_dns_
↪name,instances.private_dns_name,instances.tags:name
os_server = servers.id,servers.interface_ip,servers.name,servers.private_v4,servers.
↪public_v4
```

If the distiller is enabled, the *bkr_server* role will distill the id, url, and system values for each instance provisioned during the transaction.

Output

The distiller creates one file, placed in `<workspace>/resources/linchpin.distilled`. Each time an 'up' transaction is performed, the distilled data is overwritten.

If no output is recorded, it's likely that the provisioning didn't complete successfully, or an error occurred during data collection. The data is still available in the RunDB.

This is the output for the *aws_ec2* role, using the *aws-ec2-new* target, which provisioned two instances.

```
{
    "aws-ec2-new": [
        {
            "id": "i-0d8616a3d08a67f38",
            "name": "demo-day",
            "private_dns_name": "ip-172-31-18-177.us-west-2.compute.internal",
            "private_ip": "172.31.18.177",
            "public_dns_name": "ec2-54-202-80-27.us-west-2.compute.amazonaws.com",
            "public_ip": "54.202.80.27"
        },
        {
            "id": "i-01112909e184530fc",
            "name": "demo-night",
            "private_dns_name": "ip-172-31-20-190.us-west-2.compute.internal",
            "private_ip": "172.31.20.190",
            "public_dns_name": "ec2-54-187-172-80.us-west-2.compute.amazonaws.com",
            "public_ip": "54.187.172.80"
        }
    ]
}
```

1.3 Developer Information

The following information may be useful for those wishing to extend LinchPin.

1.3.1 Python API Reference

This page contains the list of project's modules

Linchpin API and Context Modules

The linchpin module provides the base API for managing LinchPin, Ansible, and other useful aspects for provisioning.

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

bind_to_hook_state(*callback*)
Function used by LinchpinHooksclass to add callbacks

Parameters **callback** – callback function

do_action(*provision_data*, *action='up'*, *run_id=None*, *tx_id=None*)
This function takes provision_data, and executes the given action for each target within the provision_data disctionary.

Parameters **provision_data** – PinFile data as a dictionary, with

target information

Parameters

- **action** – Action taken (up, destroy, etc). (Default: up)
- **run_id** – Provided run_id to duplicate/destroy (Default: None)
- **tx_id** – Provided tx_id to duplicate/destroy (Default: None)

do_validation(*provision_data*, *old_schema=False*)
This function takes provision_data, and attempts to validate the topologies for that data

Parameters **provision_data** – PinFile data as a dictionary, with

target information

generate_inventory(*resource_data*, *layout*, *inv_format='cfg'*, *topology_data={}*, *config_data={}*)

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.

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)

get_pf_data_from_rundb(*targets*, *run_id=None*, *tx_id=None*)
This function takes the action and provision_data, returns the pinfile data

Parameters

- **targets** – A list of targets for which to get the data
- **targets** – Tuple of target(s) for which to gather data.

- **run_id** – run_id associated with target (Default: None)
- **tx_id** – tx_id for which to gather data (Default: None)

get_run_data(*tx_id*, *fields*, *targets=()*)
Returns the RunDB for data from a specified field given a tx_id. The fields consist of the major sections in the RunDB (target view only). Those fields are action, start, end, inputs, outputs, uhash, and rc.

Parameters

- **tx_id** – tx_id to search
- **fields** – Tuple of fields to retrieve for each record requested.
- **targets** – Tuple of targets to search from within the tx_ids

hook_state
getter function for hook_state property of the API object

lp_journal(*view='target'*, *targets=[]*, *fields=None*, *count=1*, *tx_ids=None*)

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_rundb()
Configures the run database parameters, sets them into extra_vars

validate_layout(*layout*)
Validate the provided layout against the schema

Parameters **layout** – layout dictionary

validate_topology(*topology*)
Validate the provided topology against the schema

;param topology: topology dictionary

class linchpin.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(*search_path=None*)
Update self.cfgs from the linchpin configuration file (linchpin.conf).

NOTE: Must be implemented by a subclass

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

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

linchpin.ansible_runner.**ansible_runner**(*playbook_path*, *module_path*, *extra_vars*, *inventory_src='localhost'*, *verbosity=1*, *console=True*)
Uses the Ansible API code to invoke the specified linchpin playbook :param playbook: Which ansible playbook to run (default: 'up') :param console: Whether to display the ansible console (default: True)

linchpin.ansible_runner.**ansible_runner_24x**(*playbook_path*, *extra_vars*, *options=None*, *inventory_src='localhost'*, *console=True*)

linchpin.ansible_runner.**ansible_runner_2x**(*playbook_path*, *extra_vars*, *options=None*, *inventory_src='localhost'*, *console=True*)

linchpin.ansible_runner.**suppress_stdout**(**args*, ***kwds*)
This context manager provides tooling to make Ansible's Display class not output anything when used

class linchpin.callbacks.**PlaybookCallback**(*display=None*, *options=None*, *ansible_version=2.3*)
Playbook callback

v2_runner_on_failed(*result*, ***kwargs*)
Save failed result

v2_runner_on_ok(*result*)
Save ok result

LinchPin Command-Line API

The linchpin.cli module provides an API for writing a command-line interface, the *LinchPin Command Line Shell implementation* being the reference implementation.

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

find_include(*filename*, *ftype='topology'*)
Find the included file to be acted upon.

Parameters

- **filename** – name of file from to be loaded
- **ftype** – the file type to locate: topology, layout (default: topology)

lp_destroy(*targets=()*, *run_id=None*, *tx_id=None*)
This function takes a list of targets, and performs a destructive teardown, including undefining nodes, according to the target(s).

See also:

lp_down - currently unimplemented

Parameters

- **targets** – A tuple of targets to destroy.
- **run_id** – An optional run_id to use
- **tx_id** – An optional tx_id to use

lp_down (*pinfile*, *targets=()*, *run_id=None*)
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.

lp_fetch (*src*, *root=''*, *fetch_type='workspace'*, *fetch_protocol='FetchGit'*, *fetch_ref=None*, *dest_ws=None*, *nocache=False*)
Fetch a workspace from git, http(s), or a local directory, and generate a provided workspace

Parameters

- **src** – The URL or URI of the remote directory
- **root** – Used to specify the location of the workspace within the remote. If root is not set, the root of the given remote will be used.
- **fetch_type** – Specifies which component(s) of a workspace the user wants to fetch. Types include: topology, layout, resources, hooks, workspace. (default: workspace)
- **fetch_protocol** – The protocol to use to fetch the workspace. (default: git)
- **fetch_ref** – Specify the git branch. Used only with git protocol (eg. master). If not used, the default branch will be used.
- **dest_ws** – Workspaces destination, the workspace will be relative to this location.

  If *dest_ws* is not provided and *-r/–root* is provided, the basename will be the name of the workspace within the destination. If no root is provided, a random workspace name will be generated. The destination can also be explicitly set by using -w (see linchpin –help).
- **nocache** – If true, don't copy from the cache dir, unless it's longer than the configured fetch.cache_days (1 day) (default: False)

lp_init (*providers=['libvirt']*)
Initializes a linchpin project. Creates the necessary directory structure, includes PinFile, topologies and layouts for the given provider. (Default: Dummy. Other providers not yet implemented.)

Parameters **providers** – A list of providers for which templates

(and a target) will be provided into the workspace. NOT YET IMPLEMENTED

lp_setup (*providers='all'*)
This function takes a list of providers, and setsup the dependencies :param providers:

A tuple of providers to install dependencies

lp_up (*targets=()*, *run_id=None*, *tx_id=None*, *inv_f='cfg'*)
This function takes a list of targets, and provisions them according to their topology.

Parameters

- **targets** – A tuple of targets to provision
- **run_id** – An optional run_id if the task is idempotent

- **tx_id** – An optional tx_id if the task is idempotent

lp_validate(*targets=()*, *old_schema=False*)
This function takes a list of targets, and validates their topology.

Parameters **targets** – A tuple of targets to provision

:param old_schema Denotes whether schema should be validated with the old schema rather than the new one!/usr/bin/env python

pf_data
getter for pinfile template data

pinfile
getter function for pinfile name

workspace
getter function for context workspace

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

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

The following paths are used to find the config file. The search path defaults to the first-found order:

```
* /etc/linchpin.conf
* /linchpin/library/path/linchpin.conf
* <workspace>/linchpin.conf
```

An alternate search_path can be passed.

Parameters **search_path** – A list of paths to search a linchpin config

(default: None)

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

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

workspace
getter function for workspace

LinchPin Command Line Shell implementation

The linchpin.shell module contains calls to implement the Command Line Interface within linchpin. It uses the Click command line interface composer. All calls here interface with the *LinchPin Command-Line API* API.

class linchpin.shell.click_default_group.**DefaultGroup**(**args*, ***kwargs*)
Invokes a subcommand marked with *default=True* if any subcommand not chosen.

Parameters **default_if_no_args** – resolves to the default command if no arguments passed.

command(**args*, ***kwargs*)
A shortcut decorator for declaring and attaching a command to the group. This takes the same arguments as *command()* but immediately registers the created command with this instance by calling into add_command().

format_commands(*ctx*, *formatter*)
Extra format methods for multi methods that adds all the commands after the options.

get_command(*ctx*, *cmd_name*)
Given a context and a command name, this returns a Command object if it exists or returns *None*.

list_commands(*ctx*)
Provide a list of available commands. Anything deprecated should not be listed

parse_args(*ctx*, *args*)
Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by make_context().

resolve_command(*ctx*, *args*)

set_default_command(*command*)
Sets a command function as the default command.

LinchPin Hooks API

The linchpin.hooks module manages the hooks functionality within LinchPin.

class linchpin.hooks.**ActionBlockRouter**(*name*, **args*, ***kwargs*)
Proxy pattern implementation for fetching actionmanagers by name

class linchpin.hooks.**LinchpinHooks**(*api*)

prepare_ctx_params()
prepares few context parameters based on the current target_data that is being set. these parameters are based topology name.

prepare_inv_params()

run_actions(*action_blocks*, *tgt_data*, *is_global=False*)
Runs actions inside each action block of each target

Parameters

- **action_blocks** – list of action_blocks each block constitues to a type of hook
- **tgt_data** – data specific to target, which can be dict of

topology , layout, outputs, inventory :param is_global: scope of the hook

example: action_block: - name: do_something

type: shell actions:

- echo ' this is 'postup' operation Hello hai how r u ?'

run_hooks(*state*, *is_global=False*)
Function to run hook all hooks from Pinfile based on the state :param state: hook state (currently, preup, postup, predestroy, postdestroy) :param is_global: whether the hook is global (can be applied to multiple targets)

run_inventory_gen(*data*)

rundb

LinchPin Extra Modules

These are modules not documented elsewhere in the LinchPin API, but may be useful to a developer.

class linchpin.utils.dataparser.**DataParser**

load_pinfile(*pinfile*)

parse_json_yaml(*data*, *ordered=True*)
parses yaml file into json object

process(*file_w_path*, *data=None*)
Processes the PinFile and any data (if a template) using Jinja2. Returns json of PinFile, topology, layout, and hooks.

Parameters

- **file_w_path** – Full path to the provided file to process
- **data** – A JSON representation of data mapped to a Jinja2 template in file_w_path

render(*template*, *context*, *ordered=True*)
Performs the rendering of template and context data using Jinja2.

Parameters

- **template** – Full path to the Jinja2 template
- **context** – A dictionary of variables to be rendered againt the template

run_script(*script*)

write_json(*provision_data*, *pf_outfile*)

exception linchpin.exceptions.**ActionError**(**args*, ***kwargs*)

exception linchpin.exceptions.**ActionManagerError**(**args*, ***kwargs*)

exception linchpin.exceptions.**HookError**(**args*, ***kwargs*)

exception linchpin.exceptions.**LinchpinError**(**args*, ***kwargs*)

exception linchpin.exceptions.**SchemaError**(**args*, ***kwargs*)

exception linchpin.exceptions.**StateError**(**args*, ***kwargs*)

exception linchpin.exceptions.**TopologyError**(**args*, ***kwargs*)

exception linchpin.exceptions.**ValidationError**(**args*, ***kwargs*)

class linchpin.exceptions.**ValidationErrorHandler**(*tree=None*)

messages = {0: '{0}', 1: 'document is missing', 2: "field '{field}' is required", 3

class linchpin.fetch.**FetchHttp**(*ctx*, *fetch_type*, *src*, *dest*, *cache_dir*, *root=''*, *root_ws=''*, *ref=None*)

call_wget(*fetch_dir=None*)

fetch_files()

class linchpin.fetch.**FetchGit**(*ctx*, *fetch_type*, *src*, *dest*, *cache_dir*, *root=''*, *root_ws=''*, *ref=None*)

call_clone(*fetch_dir=None*)

fetch_files()

See also:

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

LinchPin on Github Code Contributions and Latest Software

webchat.freenode.net #linchpin IRC chat channel

LinchPin on PyPi Latest Release of LinchPin

1.4 FAQs

Below is a list of Frequently Asked Questions (FAQs), with answers. Feel free to submit yours in a Github issue.

1.5 Community

LinchPin has a small, but vibrant community. Come help us while you learn a skill.

See also:

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

LinchPin on Github Code Contributions and Latest Software

webchat.freenode.net #linchpin IRC chat channel

LinchPin on PyPi Latest Release of LinchPin

1.6 Glossary

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

_async *(boolean, default: False)*

Used to enable asynchronous provisioning/teardown. Sets the Ansible *async* magic_var.

async_timeout *(int, default: 1000)*

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

_check_mode/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

evars

extra_vars Variables that can be passed into Ansible playbooks from external sources. Used in linchpin via the linchpin.conf *[evars]* section.

hook Certain 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 is provided, this will be the location of the resulting ansible inventory

inventories_folder A configuration entry in linchpin.conf which stores the relative location where inventories are stored.

linchpin_config

lpconfig 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

inventory_layout 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

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

Controls whether resources will be written to the resources_file

PinFile

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

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

resource_definitions In a topology, a resource_definition describes what the resources look like when provisioned. This example shows two different dummy_node resources, the resource named *web* will get 3 nodes, while and the resource named *test* will get 1 resource.

```
resource_definitions:
  - name: "web"
    type: "dummy_node"
    count: 3
  - name: "test"
    type: "dummy_node"
    count: 1
```

resource_group_type For each resource group, the type is defined by this value. It's used by the LinchPin API to determine which provider playbook to run.

resources

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

run_id

run-id An integer identifier assigned to each task.

- The run_id can be passed to `linchpin up` for idempotent provisioning
- The run_id can be passed to `linchpin destroy` to destroy any previously provisioned resources.

rundb

RunDB A simple json database, used to store the *uhash* and other useful data, including the *run_id* and output data.

schema JSON description of the format for the topology.

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

destroy 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.

uhash

uHash Unique-ish hash associated with resources on a provider basis. Provides unique resource names and data if desired. The uhash must be enabled in linchpin.conf if desired.

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:

Source Code LinchPin Source Code

Note: Releases are formatted using semanting versioning. If the release shown above is a pre-release version, the content listed may not be supported. Use latest for the most up-to-date documentation.

—

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

See also:

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

LinchPin on Github Code Contributions and Latest Software

webchat.freenode.net #linchpin IRC chat channel

LinchPin on PyPi Latest Release of LinchPin

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W