LinchPin Documentation

Release 1.5.2pre1

Samvaran Kashyap Rallabandi

Mar 19, 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 the current released version of LinchPin (1.5.2). For recent features, we attempt to note in each section the version of LinchPin where the feature was added.

CHAPTER 1

Introduction

Before investigating the main components of LinchPin – provisioning, topologies, hooks, layouts, etc.– you'll learn how to get LinchPin installed and cover some basic concepts. We'll also cover how to use the `linchpin` command line interface, some configuration basics, and of course the provisioning providers.

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).

1.1.1 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
```

1.1.2 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>

1.1.3 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.

1.1.4 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
```

1.1.5 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
```

1.2 Getting Started

Now that LinchPin is *installed*, 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

1.2.1 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
                                  Write out PinFile to provided location
  -o, --output-pinfile OUTPUT_PINFILE
                                  Template data passed to PinFile template
  -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.

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.

See also:

FIXME: put link to credentials section here.

1.2.2 The Workspace

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.

An functional workspace can be found in the source code.

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

1.2.3 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 *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

1.2.4 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

[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
```

1.2.5 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.

See also:

Commands (CLI) Linchpin Command-Line Interface

Managing Resources Managing Resources

Providers Available Providers

1.3 Configuration File

Topics

Below is full coverage of each of the sections of the values available in linchpin.conf.

1.3.1 Getting the most current configuration

If you are installing LinchPin from a package manager (pip or RPM), the latest linchpin.conf is already included in the library.

An example linchpin.conf is available on Github.

For in-depth details of all the options, see the *Configuration Reference* document.

1.3.2 Environmental Variables

LinchPin allows configuration adjustments via environment variables in some cases. If these environment variables are set, they will take precedence over any settings in the configuration file.

A full listing of available environment variables, see the *Configuration Reference* document.

1.3.3 Command Line Options

Some configuration options are also present in the command line. Settings passed via the command line will override those passed through the configuration file and the environment.

The full list of options is covered in the *Commands (CLI)* document.

1.3.4 Values by Section

The configuration file is broken into sections. Each section controls a specific functionality in LinchPin.

General Defaults

The following settings are in the `[DEFAULT]` section of the linchpin.conf

> **Warning:** The configurations in this section should NOT be changed unless you know what you are doing.

pkg

This defines the package name. Many components base paths and other information from this setting.

```
pkg = linchpin
```

default_config_path

New in version 1.2.0

Where configuration files might land, such as the workspaces.conf, or credentials. Generally used in combination with other configurations.

```
default_config_path = ~/.config/linchpin
```

external_providers_path

New in version 1.5.0

Developers can provide additional provider playbooks and schemas. This configuration is used to set the path(s) to look for additional providers.

```
external_providers_path = %(default_config_path)s/linchpin-x
```

The following settings are in the `[init]` section of the linchpin.conf

source

Source path of files provided for the `linchpin init` command.

```
source = templates/
```

pinfile

Formal name of the PinFile. Can be changed as desired.

```
pinfile = PinFile
```

The following settings are in the `[lp]` section of the linchpin.conf

module_folder

Load custom ansible modules from this directory

```
module_folder = library
```

rundb_type

New in version 1.2.0

RunDB supports additional drivers, currently the only driver is TinyRunDB, based upon tinydb.

```
rundb_type = TinyRunDB
```

rundb_conn

New in version 1.2.0

The resource path to the RunDB connection. The TinyRunDB version (default) is a file.

Default: `{{ workspace }}/.rundb/rundb.json`

The configuration file has this option commented out. Uncommenting it could enable a system-central rundb, if desired.

```
#rundb_conn = %(default_config_path)s/rundb/rundb-::mac::.json
```

rundb_conn_type

New in version 1.2.0

Set this value if the RunDB resource is anything but a file. This setting is linked to the `rundb_conn` configuration.

```
rundb_conn_type = file
```

rundb_conn_schema

New in version 1.2.0

The schema used to store records in the TinyRunDb. Many other databases will likely not use this value, but must honor the configuration item.

```
rundb_schema = {"action": "",
                "inputs": [],
                "outputs": [],
                "start": "",
                "end": "",
                "rc": 0,
                "uhash": ""}
```

rundb_hash

New in version 1.2.0

Hashing algorithm used to create the uHash.

```
rundb_hash = sha256
```

dateformat

New in version 1.2.0

The dateformat to use when writing out start and end times to the RunDB.

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

default_pinfile

New in version 1.2.0

The dateformat to use when writing out start and end times to the RunDB.

```
default_pinfile = PinFile
```

Extra Vars

The following settings are in the `[evars]` section of the linchpin.conf

LinchPin sets several *extra_vars* values, which are passed to the provisioning playbooks.

Note: Default paths in playbooks exist. lp_path = <src_dir>/linchpin determined in the load_config method of linchpin.cli.LinchpinCliContext if either of these change, the value in linchpin/templates must also change

_check_mode

Enables the Ansible check_mode, or Dry Run functionality. Most provisioners currently DO NOT support this option

```
_check_mode = False
```

_async

LinchPin supports the Ansible async mode for certain *Providers*. Setting `async = True` here enables the feature.

```
_async = False
```

async_timeout

Works in conjunction with the *async* setting, defaulting the async wait time to `{{ async_timeout }}` in provider playbooks

```
async_timeout = 1000
```

enable_uhash

In older versions of Linchpin, the uhash value was not used. If set to true, the unique-ish hash (uhash) will be appended to instances (for providers who support naming) and the *inventory_path*.

```
enable_uhash = False
```

generate_resources

In older versions of linchpin (<v1.0.4), a *resources* folder exists, which dumped the data that is now stored in the RunDB.

```
generate_resources = True
```

output

Deprecated in version 1.2.0 Removed in version 1.5.0

Horribly named variable, no longer used

```
output = True
```

layouts_folder

Used in lookup for a specific *layout* within a workspace. The PinFile specifies the layout without a path, this is the relative location.

Also used in combination with *default_layouts_path* <*conf_def_layout_path*>, which isn't generally used.

```
layouts_folder = layouts
```

topologies_folder

Used in lookup for a specific *topology* within a workspace. The PinFile specifies the topology without a path, this is the relative location.

Also used in combination with *default_topologies_path*<*conf_def_topo_path*>, which isn't generally used.

```
topologies_folder = topologies
```

roles_folder

New in version 1.5.0

Used in combination with *default_roles_path* <*conf_def_roles_path*> for external provider roles

```
roles_folder = roles
```

inventories_folder

Relative location where inventories will be written. Usually combined with the *default_inventories_path*, but could be relative tothe workspace.

```
_check_mode = False
```

inventories_folder = inventories

hooks_folder

Relative location within the workspace where hooks data is stored

```
hooks_folder = hooks
```

resources_folder

Deprecated in version 1.5.0

Relative location of the resources destination path. Generally combined with the *default_resource_path*

```
resources_folder = resources
```

schemas_folder

Deprecated in version 1.2.0

Relative location of the schemas within the LinchPin codebase

```
schemas_folder = schemas
```

playbooks_folder

Relative location of the Ansible playbooks and roles within the LinchPin codebase

```
playbooks_folder = provision
```

default_schemas_path

Deprecated in version 1.5.0

Used to locate default schemas, usually *schema_v4* or *schema_v3*

```
default_schemas_path = {{ lp_path }}/defaults/%(schemas_folder)s
```

default_topologies_path

Deprecated in version 1.2.0

Default location for topologies in cases where *topology* or *topology_file* is not set.

```
default_topologies_path = {{ lp_path }}/defaults/%(topologies_folder)s
```

default_layouts_path

Deprecated in version 1.2.0

When inventories are processed, layouts are looked up here if *layout_file* is not set

```
default_layouts_path = {{ lp_path }}/defaults/%(layouts_folder)s
```

default_inventories_path

Deprecated in version 1.2.0

When writing out inventories, this path is used if *inventory_file* is not set

```
default_inventories_path = {{ lp_path }}/defaults/%(inventories_folder)s
```

default_resources_path

Deprecated in version 1.2.0

When writing out resources files, this path is used if *inventory_file* is not set

```
default_inventories_path = {{ lp_path }}/defaults/%(inventories_folder)s
```

default_roles_path

When using an external provider, this path points back to some of the core roles needed in the provider's playbook.

```
default_roles_path = {{ lp_path }}/%(playbooks_folder)s/%(roles_folder)s
```

default_roles_path = {{ lp_path }}/%(playbooks_folder)s/%(roles_folder)s

schema_v3

Deprecated in v1.5.0

Full path to the location of the `schema_v3.json` file, which is used to perform validation of the topology.

```
_check_mode = False
```

schema_v3 = %(default_schemas_path)s/schema_v3.json

schema_v4

Deprecated in v1.5.0

Full path to the location of the `schema_v4.json` file, which is used to perform validation of the topology.

```
schema_v4 = %(default_schemas_path)s/schema_v4.json
```

default_credentials_path

If the `--creds-path` option or `$CREDS_PATH` environment variable are not set, use this location to look up credentials files defined in a topology.

```
default_credentials_path = %(default_config_path)s
```

inventory_path

New in version 1.5.0

The *inventory_path* is used to set the value of the resulting inventory file which is generated by LinchPin. This value is dynamically generated by default.

Note: This should not be confused with the *inventory_file* which is an input to the LinchPin ansible playbooks.

```
#inventory_path = {{ workspace }}/{{inventories_folder}}/happy.inventory
```

default_ssh_key_path

New in version 1.2.0

Used solely in the *libvirt provider* <*prov_libvirt*>. Could be used to set a default location for ssh keys that might be passed via a cloud-config setup.

```
default_ssh_key_path = ~/.ssh
```

libvirt_image_path

Where to store the libvirt images for copying/booting instances. This can be adjusted to a user accessible location if permissions are an issue.

Note: Ensure the *libvirt_user* and *libvirt_become* options below are also adjusted according to proper permissions.

```
libvirt_image_path = /var/lib/libvirt/images/
```

libvirt_user

What user to use to access the libvirt services.

Note: Specifying *root* means that linchpin will attempt to access the libvirt service as the *root* user. If the linchpin user is not root, sudo without password must be setup.

```
libvirt_user = root
```

libvirt_become

When using root or any privileged user, this must be set to yes.

Note: If the linchpin user is not root, sudo without password must also be setup.

```
libvirt_become = yes
```

Initialization Settings

The following settings are in the `[init]` section of the linchpin.conf.

These settings specifically pertain to *linchpin init*, which stores templates in the source code to generate a simple example workspace.

source

Location of templates stored in the source code. The structure is built to resemble the directory structure explained in *linchpin init*.

```
source = templates/
```

pinfile

Formal name of the *PinFile*. Can be changed as desired.

```
pinfile = PinFile
```

Logger Settings

The following settings are in the `[logger]` section of the linchpin.conf.

Note: These settings are ONLY used for the Command Line Interface. The API configures a console output only logger and expects the functionality to be overwritten in subclasses.

enable

Whether logging to a file is enabled

```
enable = True
```

file

Name of the file to write the log. A relative or full path is acceptable.

```
file = linchpin.log
```

format

The format in which logs are written. See *https://docs.python.org/2/library/logging.html#logrecord-attributes* for more detail and available options.

```
format = %%(levelname)s %%(asctime)s %%(message)s
```

dateformat

How to display the date in logs. See *https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior* for more detail and available options.

Note: This action was never implemented.

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

level

Default logging level

```
level = logging.DEBUG
```

The following settings are in the `[console]` section of the linchpin.conf.

Each of these settings is for logging output to the console, except for Ansible output.

format

The format in which console output is written. See *https://docs.python.org/2/library/logging.html#logrecord-attributes* for more detail and available options.

```
format = %%(message)s
```

level

Default logging level

```
level = logging.INFO
```

Hooks Settings

The following settings are in the `[states]` section of the linchpin.conf.

These settings define the state names, which are useful in hooks.

preup

Define the name of the so called *preup* state. This state is set and executed prior to the 'up' action being called, but after the PinFile data is loaded.

```
preup = preup
```

predestroy

Define the name of the so called *predestroy* state. This state is set and executed prior to the 'destroy' action being called, but after the PinFile data is loaded.

```
predestroy = predestroy
```

postup

Define the name of the so called *postup* state. This state is set and executed after to the 'up' action is completed, and after the *postinv* state is executed.

```
postup = postup
```

postdestroy = postdestroy ~~

Define the name of the so called *postdestroy* state. This state is set and executed after to the 'destroy' action is completed.

```
postdestroy = postdestroy
```

postinv

Define the name of the so called *postinv* state. This state is set and executed after to the 'up' action is completed, and before the *postup* state is executed. This is eventually going to be the inventory generation hook.

```
postinv = inventory
```

The following settings are in the `[hookstates]` section of the linchpin.conf.

These settings define the 'STATES' each action uses in hooks.

up

For the 'up' action, types of hook states are available for execution

```
up = pre,post,inv
```

destroy

For the 'destroy' action, types of hook states are available for execution

```
destroy = pre,post
```

inv

New in version 1.2.0

For the inventory generation, which only happens on an 'up' state.

Note: The *postinv* state currently doesn't do anything. In the future, it will provide a way to generate inventories besides the standard Ansible static inventory.

```
inv = post
```

File Extensions

The following settings are in the `[extensions]` section of the linchpin.conf.

These settings define the file extensions certain files have..

resource

Deprecated in version 1.2.0

Removed in version 1.5.0

When generating resource output files, append this extension

```
resource = .output
```

inventory

When generating Ansible static inventory files, append this extension

```
inventory = .inventory
```

playbooks

New in version 1.5.0

Since playbooks fundamentially changed between v1.2.0 and v1.5.0, this option was added for looking up a provider playbook. It's unlikely this value will change.

```
playbooks = .yml
```

Playbook Settings

The following settings are in the `[playbooks]` section of the linchpin.conf.

Note: The entirety of this section is removed in version 1.5.0+. The redesign of the LinchPin API now calls individual playbooks based upon the *resource_group_type* value.

up

Removed in version 1.5.0

Name of the playbook associated with the 'up' (provision) action

```
up = site.yml
```

destroy

Removed in version 1.5.0

Name of the playbook associated with the 'destroy' (teardown) action

```
destroy = site.yml
```

down

Removed in version 1.5.0

Name of the playbook associated with the 'down' (halt) action

Note: This action has not been implemented.

```
down = site.yml
```

schema_check

Removed in version 1.5.0

Name of the playbook associated with the 'schema_check' action.

Note: This action was never implemented.

```
schema_check = schemacheck.yml
```

inv_gen

Removed in version 1.5.0

Name of the playbook associated with the 'inv_gen' action.

Note: This action was never implemented.

```
inv_gen = invgen.yml
```

test

Removed in version 1.5.0

Name of the playbook associated with the 'test' action.

Note: This action was never implemented.

```
test = test.yml
```

See also:

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

irc.freenode.net #linchpin IRC chat channel

CHAPTER 2

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*.

2.1 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
```

2.2 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
```

2.3 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

2.4 linchpin journal

Upon completion of any provision (up) or teardown (destroy) task, there's a record that is created ans stored in the *RunDB*. The `linchpin journal` command provides data from the tasks, based upon the target. The data displayed to the screen shows the last three (3) tasks, along with some useful information.

```
$ linchpin journal dummy-new

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

The `linchpin journal` 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
```

2.5 linchpin fetch

CHAPTER 3

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.

3.1 PinFiles

These PinFiles represent many combinations of complexity and providers.

PinFiles are processed top to bottom.

3.1.1 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

3.1.2 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

3.1.3 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
```

3.1.4 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
```

3.1.5 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"
                }
            ]
        }
    }
}
```

3.2 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.

3.2.1 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
```

3.2.2 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.

3.2.3 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

3.2.4 JSON

New in version 1.5.0

Topologies can be written using JSON format.

- dummy.json

3.2.5 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
```

3.3 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.

3.3.1 YAML

Layouts written using YAML format:

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

3.3.2 JSON

New in version 1.5.0

Layouts can be written using JSON format.

- gcloud.json

3.3.3 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 *Inventory Layouts*.

See also:

Providers

CHAPTER 4

Providers

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

4.1 openstack

The openstack provider manages multiple types of resources.

4.1.1 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	

4.1.2 os_obj

Openstack Object Storage can be provisioned using this resource.

- Topology Example
- Ansible module

4.1.3 os_vol

Openstack Cinder Volumes can be provisioned using this resource.

- Topology Example
- Ansible module

4.1.4 os_sg

Openstack Security Groups can be provisioned using this resource.

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

4.1.5 Additional Dependencies

No additional dependencies are required for the Openstack Provider.

4.1.6 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:

      .. 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
```

4.2 libvirt

The libvirt provider manages two types of resources.

4.2.1 libvirt_node

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

- Topology Example
- Ansible module

4.2.2 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

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

4.2.3 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
```

4.2.4 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.

4.2.5 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.

4.3 aws

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

4.3.1 aws_ec2

AWS Instances can be provisioned using this resource.

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

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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 Additional Dependencies

No additional dependencies are required for the AWS Provider.

4.3.6 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.

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
```

4.4 gcloud

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

4.4.1 gcloud_gce

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

- Topology Example
- Ansible module

4.4.2 Additional Dependencies

No additional dependencies are required for the Openstack Provider.

4.4.3 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.

4.5 beaker

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

4.5.1 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

4.5.2 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]
```

4.5.3 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.

4.6 duffy

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

4.6.1 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.

4.6.2 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.

4.6.3 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.

4.6.4 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
```

4.7 ovirt

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

4.7.1 ovirt_vms

oVirt Domains/VMs can be provisioned using this resource.

- Topology Example

- Ansible module

4.7.2 Additional Dependencies

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

4.7.3 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
```

4.8 openshift

The openshift provider manages two resources, `openshift_inline`, and `openshift_external`.

4.8.1 openshift_inline

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

- Topology Example

The ansible module for openshift is written and bundled as part of LinchPin.

- openshift.py

Note: The **'oc <https://docs.ansible.com/ansible/2.4/oc_module.html'_** module was included into ansible after the above openshift module was created and included with LinchPin. The future may see the oc module used.

4.8.2 openshift_external

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

4.8.3 Additional Dependencies

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

4.8.4 Credentials Management

An openshift topology can have a `credentials` section for each resource_group, which requires the *api_endpoint*, and the *api_token* values.

```
---
topology_name: topo
resource_groups:
  - resource_group_name: openshift
    resource_group_type: openshift
    resource_definitions:
      - name: openshift
        role: openshift_inline
        data:

      .. snip ..

    credentials:
      api_endpoint: example.com:8443/
      api_token: mytokentextrighthere
```

CHAPTER 5

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*.

5.1 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.

5.2 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.

5.3 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.

CHAPTER 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.

6.1 Inventory Layouts

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

6.1.1 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 ..
```

6.1.2 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 %}"
```

6.1.3 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 %}"
```

CHAPTER 7

Developer Information

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

7.1 Python API Reference

This page contains the list of project's modules

7.1.1 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*)
This function takes provision_data, and executes the given action for each target within the provision_data disctionary.

Parameters

- **provision_data** – PinFile as a dictionary, with target information
- **action** – Action taken (up, destroy, etc). (Default: up)
- **run_id** – Provided run_id to duplicate/destroy (Default: None)

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)

hook_state
getter function for hook_state property of the API object

lp_journal(*targets=[]*, *fields=None*, *count=1*)

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

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*)
Playbook callback

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

v2_runner_on_ok(*result*)
Save ok result

7.1.2 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*)
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

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

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=None*, *fetch_type='workspace'*)

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_up(*targets=()*, *run_id=None*)
This function takes a list of targets, and provisions them according to their topology.

Parameters

- **pinfile** – Provided PinFile, with available targets
- **targets** – A tuple of targets to provision
- **run_id** – An optional run_id if the task is idempotent

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

7.1.3 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*)

format_commands(*ctx*, *formatter*)

get_command(*ctx*, *cmd_name*)

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

parse_args(*ctx*, *args*)

resolve_command(*ctx*, *args*)

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

7.1.4 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

7.1.5 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*)
parses yaml file into json object

process(*file_w_path*, *data_w_path=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
- **targets** – A tuple of targets to provision
- **run_id** – An optional run_id if the task is idempotent or a destroy action

render(*template*, *context*)
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*)

linchpin.utils.dataparser.**dict_constructor**(*loader*, *node*)

linchpin.utils.dataparser.**dict_representer**(*dumper*, *data*)

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.fetch.**FetchLocal**(*ctx*, *fetch_type*, *src*, *dest*, *cache_dir*, *root*)

fetch_files()

class linchpin.fetch.**FetchHttp**(*ctx*, *fetch_type*, *src*, *dest*, *cache_dir*, *root*)

call_wget(*src*, *fetch_dir=None*)

fetch_files()

class linchpin.fetch.**FetchGit**(*ctx*, *fetch_type*, *src*, *dest*, *cache_dir*, *root*)

call_clone(*fetch_dir=None*)

fetch_files()

See also:

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

irc.freenode.net #linchpin IRC chat channel

CHAPTER 8

FAQs

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

CHAPTER 9

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

irc.freenode.net #linchpin IRC chat channel

LinchPin on Github Code Contributions and Latest Software

CHAPTER 10

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

—

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

See also:

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

irc.freenode.net #linchpin IRC chat channel

LinchPin on Github Code Contributions and Latest Software

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

H

I

L

O

P

R

S

T

U

V

W