LinchPin Documentation

Release 2.0.0

Samvaran Kashyap Rallabandi

May 02, 2020

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 (2.0.0). For recent features, see the updated 2.0.0.

CHAPTER

ONE

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*. Test

1.1.1 Installation

LinchPin can be run either as a container or as a bare-metal application

Docker Installation

The LinchPin container is built using the latest Fedora image. The image exists in the docker hub as contrainfra/linchpin and is updated with each release. The image can also be build manually.

From within the config/Dockerfiles/linchpin directory:

```
$ sudo buildah bud -t linchpin .
```

Finally, to run the linchpin container:

```
$ sudo buildah run linchpin -v /path/to/workspace:/workdir -- linchpin -w /wordir up
$ sudo buildah run linchpin -v /path/to/workspace:/workdir -- linchpin -w /workdir -
↪vv destroy
```

Note: Setting the CREDS_PATH environment variable pointing the /workdir is recommended. AWS credentials can also be passed as evironment variables when the container is run, named AWS_SECRET_ACCESS_KEY and AWS_ACCESS_KEY_ID

Note: Beaker uses kinit, which is installed in the container but must be run within the container after it starts The default /etc/krb5.conf for kerberos requires privilege escalation. The linchpin Dockerfile replaces it with a version that eliminates this need

Bare Metal Installation

Currently, LinchPin can be run from any machine with Python 2.6+ (Python 3.x is currently experimental), and requires Ansible 2.7.1 or newer.

Note: Some providers have additional dependencies. Additional software requirements can be found in the *Examples for all 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 2.9.0 or newer, this is a core requirement. Beyond installing Ansible, there are several packages that need to be installed:

```
* libffi-devel
* libyaml-devel
* python3-libselinux
* make
* gcc
* redhat-rpm-config
* libxml2-python
* libxslt-python
```

For CentOS or RHEL the following packages should be installed:

```
$ sudo yum install python3-pip python3-virtualenv libffi-devel \
openssl-devel libyaml-devel gmp-devel libselinux-python make \
gcc redhat-rpm-config git
```

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

For Fedora 30+ the following packages should be installed:

```
$ sudo dnf install python3-virtualenv libffi-devel \
openssl-devel libyaml-devel gmp-devel python3-libselinux 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) $ pip3 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) $ pip3 install linchpin[docs]
(linchpin) $ pip3 install linchpin[tests]
```

Virtual Environments and SELinux

When using a virtualenv with SELinux enabled, LinchPin may fail due to an error related to the python3-libselinux libraries. This is because the python3-libselinux 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 python3-libselinux
.. snip ..
(linchpin) $ echo ${VIRTUAL_ENV}
/path/to/virtualenvs/linchpin
(linchpin) $ export VENV_LIB_PATH=lib/python3.x/site-packages
(linchpin) $ export LIBSELINUX_PATH=/usr/lib64/python3.x/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 **:code1.5:`scripts/install_selinux_venv.sh`**

Installing on Fedora 30+

Install RPM pre-reqs

```
$ sudo dnf -y install python3-virtualenv libffi-devel openssl-devel libyaml-devel␣
↪python3-libselinux 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) $ pip3 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 python3-
↪libselinux make gcc redhat-rpm-config libxml2-devel libxslt-devel libxslt-python␣
↪libxslt-python
```

Create a working-directory

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

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

```
$ mkvirtualenv linchpin
..snip..
(linchpin) $ pip3 install linchpin
```

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

```
(linchpin) $ pip3 install -U setuptools
```

Install linchpin

```
(linchpin) $ pip3 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) $ pip3 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.

Install all the dependencies:

```
$ linchpin setup
```

To install only a subset of dependencies, pass as arguments list:

```
$ linchpin setup beaker docs
```

It also supports ask-sudo-pass parameter when installing dnf related dependencies:

```
$ linchpin setup libvirt --ask-sudo-pass
```

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 ..

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

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 *Examples for all Providers*.

1.1.7 Linchpin Hooks

Description:

Every resource provisioned by linchpin goes through multiple states. Each state has its own context. Depending upon the state Linchpin provides a feature to trigger single or multiple events. In Linchpin terminology, each event can initiate execution of a script/scripts or Ansible playbooks called hooks. Hooks are used to configure or interact with resources provisioned or about to be provisioned. The trigger to the hooks is determined by the state in which it is defined.

Different states linchpin provisioning undertakes are as follows:

- preup: State before provisioning the topology resources
- postup: State after provisioning the topology resources, and generating the optional inventory

- predestroy: State before teardown of the topology resources
- postdestroy: State after teardown of the topology resources

Depending upon the state section in which it is defined the hooks are triggered.

In linchpin, there are a set of python interfaces called ActionManagers which are responsible for the execution of a hook. Based on the runtime they use to execute hook there are multiple types of Action managers exists. Here's a list of built-in Action Managers:

- shell: Allows either inline shell commands or an executable shell script
- python: Executes a Python script
- ansible: Executes an Ansible playbook, allowing passing of a vars_file and extra_vars represented as a Python dict
- nodejs: Executes a Node.js script
- ruby: Executes a Ruby script

In addition to the above action managers, User can define their custom action manager. Refer Action managers documentation for more details.

A hook is bound to a specific target and must be re-stated for each target used.

Based on how they are packaged linchpin hooks are classified into two types:

- User defined hooks: These hooks are written following specific syntax and folder structure within the workspace. These are triggered based upon the section in which it is declared.

  User-defined hooks are to be declared within a linchpin workspace folder named "hooks" by default. However, this path can be configured by variable hooks_folder in [evars] section of linchpin.conf.

```
[evars]
...
hooks_folder = /path/to/hooks_folder
```

- Built-in hooks (in development): These hooks are pre-packaged with linchpin and they do not need any file structure to be declared in workspaces to work. They can be directly referenced within the Pinfile.

User defined hook example:

Let us consider a user-defined hook for example.

Each hook follows a strict folder structure. If not followed the hooks execution will result in failure. The following is an example workspace which has a user-defined ansible hook named example_hook. The following would be the directory tree structure of the workspace.

```
.
├── credentials
├── hooks
│   └── ansible
│       ├── example_hook
│       │   ├── test_hook1.yaml
│       │   └── test_hook2.yaml
│       └── example_hook2
│           └── test_ex.yaml
├── inventories
├── layouts
│   └── dummy-layout.yml
```

(continues on next page)

(continued from previous page)

```
├── linchpin.conf
├── linchpin.log
├── PinFile
├── resources
└── topologies
  └── dummy-topology.yml
```

Every hook with respect to their type is declared in their respective folder ie., ansible hooks go inside ansible folder, python hooks are declared in python folder etc., The current example illustrates the folder structure of ansible based hooks. For more examples folder structures of other hooks refer *Hooks examples*. Further, the name of the folder should be the name of the hook that will be referred to within a PinFile. Since Ansible relies on the playbooks. All the playbooks are to be defined within the folder.

The following is how a user-defined hook looks like when referenced in a Pinfile dummy provider.

```
---
dummy_target:
  topology:
    topology_name: "dummy"
    resource_groups:
    - resource_group_name: "dummy"
      resource_group_type: "dummy"
      resource_definitions:
      - role: "dummy_node"
        name: "web"
        count: 1
  layout:
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        example-node:
          count: 1
          host_groups:
          - example
  hooks:
    postup:
    - name: example_hook     # name of the hook
      type: ansible       # type of the hook ie., the type of action manager being␣
↪used.
      context: True       # whether to pass the linchpin context variables or not.
      actions:
        - playbook: test_hook1.yaml  # file name of the playbook to be run
        - playbook: test_hook2.yaml
    - name: example_hook2     # name of the hook
      type: ansible       # type of the hook ie., the type of action manager being␣
↪used.
      context: True       # whether to pass the linchpin context variables or not.
      actions:
        - playbook: test_ex.yaml  # file name of the playbook to be run
```

As mentioned previously, depending upon the state where the user would like to execute hooks can be triggered at preup, postup, predestroy, postdestroy states. Within Pinfile these states are defined as separate sections. Every hook declared within a section is executed in a top-down approach. Thus, according to the above example, example_hook would be executed first after that execution is successful, example hook2 would be executed.

Parameters of user-defined hooks:

- name: Name of the hook that is defined. Further, it should match the name of the folder inside the hooks_folder configured
- type: Type of the action manager that is to be used can be any one of ansible, shell, python, ruby, and nodejs.
- Context: while declaring hooks provide an option called as context. When the context variable is set to True some of the linchpin context variables are passed as runtime parameters to the playbooks/scripts executed. This is feature is very helpful when end-user would like to run addition configuration playbooks on provisioned instances.
- actions: Actions are the list of commands, scripts or playbooks which will be run. There can be multiple actions with the same hook file referenced. If it is an ansible type hook, The elements in action should have a playbook, extra_vars(Optional) parameters instead of directly referencing the file path. For more examples refer Linchpin Hooks examples section.

Action manager specific parameters:

The following are examples for different types of hooks using multiple action_managers

- Ansible:

```
- name: example_hook2    # name of the hook
  type: ansible      # type of the hook ie., the type of action manager being␣
↪used.
  context: True      # whether to pass the linchpin context variables or not.
    path: /path/to/scripts # optional , by default path would be configured hooks_
↪folder
    actions:
    - playbook: test_ex.yaml  # file name of the playbook to be run
      extra_vars:
        testvar: testval # extravars are optional
```

- Python:

```
- name: example_hook2    # name of the hook
  type: python      # type of the hook ie., the type of action manager being used.
  context: True      # whether to pass the linchpin context variables or not.
  path: /path/to/scripts # optional , by default path would be configured hooks_
↪folder
  actions:
    - script.py  #file name of the playbook to be run
```

- shell:

```
- name: example_hook3    # name of the hook
  type: shell      # type of the hook ie., the type of action manager being used.
  context: True      # whether to pass the linchpin context variables or not.
  path: /path/to/scripts # optional , by default path would be configured hooks_
↪folder
  actions:
  # make sure the script file has execute permissions and shebang header included.
    - script.sh  #file name of the playbook to be run
```

- Ruby:

```
- name: example_ruby    # name of the hook
  type: ruby       # type of the hook ie., the type of action manager being used.
  context: True       # whether to pass the linchpin context variables or not.
  path: /path/to/scripts # optional , by default path would be configured hooks_
↪folder
  actions:
    - script.rb  #file name of the playbook to be run
```

- Nodejs:

```
- name: example_nodejs    # name of the hook
  type: nodejs       # type of the hook ie., the type of action manager being used.
  context: True       # whether to pass the linchpin context variables or not.
  path: /path/to/scripts # optional , by default path would be configured hooks_
↪folder
  actions:
    - script.js  #file name of the playbook to be run
```

Note: For both ruby and nodejs the runtime interpreters should be pre-installed in the host machine.

- linchpin global hooks or builtins:

Linchpin also provides a prepackaged set of built-in hooks which can be referenced within Pinfile without creating a hooks folder structure. These built-ins are ansible based hooks each having different parameters. Currently, There are three builtin linchpin hooks available to end user. They are:

- ping: Simple ICMP ping to check the host provisioned in inventory is up or not
- check_ssh: linchpin tries to check the ssh server is up and running by logging into the machines provisioned using a ssh key
- port_up: Checks whether the list of network ports are up or down.

All the builtin hooks are context-aware, Thus, every built-in hook is run against the inventory file generated during the linchpin provisioning process.

Builtin hooks Example:

```
---
os-server-target:
  topology:
    topology_name: os-server-inst
    resource_groups:
      - resource_group_name: os-server-addl-vols
        resource_group_type: openstack
        resource_definitions:
        - name: "database"
          role: os_server
          flavor: m1.small
          image: CentOS-7-x86_64-GenericCloud-1612
          count: 1
          keypair: test_keypairsk2
          fip_pool: 10.8.240.0
          networks:
            - e2e-openstack
        credentials:
          filename: clouds.yaml
          profile: ci-rhos
  layout:
```

(continues on next page)

(continued from previous page)

```
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        addl-vols-node:
          count: 1
          host_groups:
          - hello
hooks:
  postup:
    # check_ssh, ping and port_up are builtin hooks
    # note builtin hooks follow different structure when compared to localhooks
    - name: check_ssh
      extra_vars:
        # since checking ssh depends on logging into machine pem file, ssh_user are␣
↪must
        ansible_ssh_private_key_file:  test_keypairsk2.key
        ansible_ssh_user: centos
        ansible_ssh_common_args: "'-o StrictHostKeyChecking=no'"
        ansible_python_interpreter: "/usr/bin/python"
    - name: ping
    - name: port_up
      ports:
        - 22
        - 8080
```

Hook Communication:

Hooks can read data from other hooks run in the same target. Hook data is not shared between a provisioning and corresponding teardown task, but is shared between pre- and post- provisioning as well as between action managers.

Experimental With the exception of the Ansible action manager, hook data is passed via the command line. Each hook will receive two arguments on the command line. The first is a json array containing data from previous hook runs. If the hooks are associated with a teardown, this will include hook data for both the hooks in the current run and the hooks in the corresponding provisioning step. Each item in the array is an object with three fields: return_code. data, and state (e.g. preup). The second argument is a path to a temporary file. In order for a hook to share data, it should write any data it wants to share as json to this file. If the data in the file is not valid json, it will be ignored.

The ansible action manager handles data somewhat differently. The results array is passed as a variable called hook_results to Ansible's extra vars. Data from Ansible will be sent back to LinchPin using the PlaybookCallback class.

Note: For more examples please refer hooks examples section.

Linchpin Hooks CLI (Options)

By default, hooks run as a part of the provisioning process. Hooks are executed in the following order: 1. preup 2. postup 3. predestroy 4. postdestroy

Since each state can have multiple hooks defined hooks linchpin provisioning process can be affected by success and failure of hook. By default, whenever there is any failure in execution of hook the provisioning process aborts. However, this behaviour can be defined changed by two command line options –ignore-failed-hooks and –no-hooks

–ignore-failed-hooks on enabling this option the failure of hooks does not affect the provisioning process. If provisioning is successful linchpin exits with 0

–no-hooks Allows user to skip the execution of hooks

Usage:

```
linchpin -vvvv --creds-path ./credentials/ up --no-hooks

linchpin -vvvv --creds-path ./credentials/ destroy --no-hooks

linchpin -vvvv --creds-path ./credentials/ up --ignore-failed-hooks

linchpin -vvvv --creds-path ./credentials/ destroy --ignore-failed-hooks
```

Further, the above mentioned options can be configured permanently in hookflags section of linchpin.conf

```
[hookflags]
no_hooks = False
ignore_failed_hooks = False
```

Linchpin Hooks: Examples

Following document has most common examples of linchpin hooks

Example1: Running ansible based hooks on Openstack based instances

- Refer: *Workspace <https://github.com/samvarankashyap/linchpin_hooks_openstack_ws>*
- Pinfile:

```
---
os-server-addl-vols:
  topology:
    topology_name: os-server-inst
    resource_groups:
      - resource_group_name: os-server-addl-vols
        resource_group_type: openstack
        resource_definitions:
        - name: "database"
          role: os_server
          flavor: m1.small
          image: CentOS-7-x86_64-GenericCloud-1612
          count: 1
          keypair: testkeypair_sk
          fip_pool: 10.8.240.0
          networks:
          - e2e-openstack
        credentials:
          filename: clouds.yaml
          profile: ci-rhos
  layout:
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        addl-vols-node:
          count: 1
          host_groups:
            - hello
  hooks:
```

(continues on next page)

(continued from previous page)

```
    postup:
        actions:
        - name: osoos
          type: ansible
          context: True
          actions:
        - playbook: install_packages.yaml
          extra_vars:
            ansible_ssh_private_key_file: "testkeypair_sk.key"
            ansible_ssh_user: centos
        - playbook: git_clone.yaml
          extra_vars:
            ansible_ssh_private_key_file: "testkeypair_sk.key"
            ansible_ssh_user: centos
```

Example: Running Global hook ping, check_ssh, port_up on Openstack based resources

```
---
os-server-addl-vols:
  topology:
    topology_name: os-server-inst
    resource_groups:
    - resource_group_name: os-server-addl-vols
      resource_group_type: openstack
      resource_definitions:
        - name: "database"
          role: os_server
          flavor: m1.small
          image: CentOS-7-x86_64-GenericCloud-1612
          count: 1
          keypair: test_keypairsk2
          fip_pool: 10.8.240.0
          networks:
            - e2e-openstack
      credentials:
        filename: clouds.yaml
        profile: ci-rhos
  layout:
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        addl-vols-node:
          count: 1
          host_groups:
          - hello
  hooks:
    postup:
      # check_ssh, ping and port_up are builtin hooks
      # note builtin hooks follow different structure when compared to localhooks
      - name: check_ssh
        extra_vars:
          # since checking ssh depends on logging into machine pem file, ssh_user are␣
↪must
          ansible_ssh_private_key_file:  /home/srallaba/.ssh/test_keypairsk2.key
          ansible_ssh_user: centos
          ansible_ssh_common_args: "'-o StrictHostKeyChecking=no'"
```

(continues on next page)

(continued from previous page)

```
        ansible_python_interpreter: "/usr/bin/python"
    - name: ping
```

Example3: Running python based hook on dummy workspace

- Workspace tree:

```
.
```

├── credentials ├── hooks │ └── python │ └── test_python │ └── test.py ├── inventories ├── layouts │ └── dummy-layout.yml ├── linchpin.conf ├── PinFile ├── resources └── topologies

- Pinfile:

```
---
```

dummy_target:

topology: topology_name: "dummy" resource_groups: - resource_group_name: "dummy"
resource_group_type: "dummy" resource_definitions: - role: "dummy_node"
name: "web" count: 1

layout:

inventory_layout:

vars: hostname: __IP__

hosts:

example-node: count: 1 host_groups:

– example

hooks:

preup:

– name: test_python type: python context: False actions: - test.py hello hi # hello hi will be command line parameters parameters passed to script test.py

Linchpin Custom Action Managers

Linchpin custom action managers:

In linchpin, ActionManagers are set of python interfaces responsible for execution of linchpin hook based on their type. There are two types of ActionManagers builtins and custom.

Here's a list of built-in Action Managers:

- shell: Allows either inline shell commands or an executable shell script
- python: Executes a Python script
- ansible: Executes an Ansible playbook, allowing passing of a vars_file and extra_vars represented as a Python dict
- nodejs: Executes a Node.js script
- ruby: Executes a Ruby script

In addition to the above action managers, User can define their custom action manager. custom/userdefined action managers are helpful when there is a specific runtime end user would like to make use of for executing a hook.

For example, if linchpin end user would like to use a "xyz" language based runtime or a custom command to be run when certain paramters are passed to a hook. They can do it with help of hook based on custom_action_manager.

Consider the following dummy workspace example for custom_action_manager:

```
.
├── credentials
├── hooks
│   ├── custom
│   │   └── somecustomhook
│   │       ├── custom_action_manager.py
│   │       ├── custom_action_manager.pyc
│   │       └── test_custom.py
├── inventories
├── layouts
├── linchpin.conf
├── linchpin.log
├── localhost
├── PinFile
├── resources
└── topologies
    └── dummy-topology.yml
```

```
---
dummy_target:
  topology:
    topology_name: "dummy"
    resource_groups:
    - resource_group_name: "dummy"
      resource_group_type: "dummy"
      resource_definitions:
      - role: "dummy_node"
        name: "web"
        count: 1
  layout:
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        example-node:
          count: 1
          host_groups:
            - example
  hooks:
    postup:
      - name: somecustomhook
        type: custom
        action_manager: custom_action_manager.py
        # action_manager: /path/to/manager
        # if not absolute path
        # linchpin searches in hooks folder configured
        context: True
        actions:
          - script: some_script.go
```

As you can see in the above structure the custom hook follows the same structure of a userdefined hook. However, we

also need to add python interface custom_action_manager.py (which can be named any) within thehooks folder or the absolute path to the python file is to be mentioned in the Pinfile

In order to write a custom_action_manager one has to implement builtin linchpin ActionManager class overriding the following functions:

- validate: (optional): validate schema for hook designed
- load: How to load the context parameters
- execute: Responsible for executing the files based on the parameters

Once the above functions are implemented the class file can be included in Pinfile.

Following is an example for the python interface implemented:

```
import os
import yaml
import json

from cerberus import Validator

from linchpin.exceptions import HookError
from linchpin.hooks.action_managers.action_manager import ActionManager


class CustomActionManager(ActionManager):

    def __init__(self, name, action_data, target_data, **kwargs):

        """
        The following is an example for CustomActionManager
        AnsibleActionManager constructor
        :param name: Name of Action Manager , ( ie., ansible)
        :param action_data: dictionary of action_block
        consists of set of actions
        example:
        - name: nameofthehook
          type: custom
          actions:
            - script: test_playbook.yaml
        :param target_data: Target specific data defined in PinFile
        :param kwargs: anyother keyword args passed as metadata
        """

        self.name = name
        self.action_data = action_data
        self.target_data = target_data
        self.context = kwargs.get("context", True)
        self.kwargs = kwargs


    def validate(self):

        """
        Validates the action_block based on the cerberus schema
        example:: ansible_action_block::::
        - name: nameofthehook
          type: customhook
          actions:
```

(continues on next page)

(continued from previous page)

```
            - script: test_playbook.yaml
        """
        """
        schema = {
            'name': {'type': 'string', 'required': True},
            'type': {'type': 'string', 'allowed': ['custom']},
            'path': {'type': 'string', 'required': False},
            'context': {'type': 'boolean', 'required': False},
            'actions': {
                'type': 'list',
                'schema': {
                    'type': 'dict',
                    'schema': {
                        'script': {'type': 'string', 'required': True}
                    }
                },
                'required': True
            }
        }

        v = Validator(schema)
        status = v.validate(self.action_data)

        if not status:
            raise HookError("Invalid syntax: {0}".format((v.errors)))
        else:
            return status


    def load(self):

        """
        Loads the ansible specific managers and loaders
        """
        return True

    def get_ctx_params(self):

        """
        Reformats the ansible specific context variables
        """

        ctx_params = {}
        ctx_params["resource_file"] = (
            self.target_data.get("resource_file", None))
        ctx_params["layout_file"] = self.target_data.get("layout_file", None)
        ctx_params["inventory_file"] = (
            self.target_data.get("inventory_file", None))

        return ctx_params


    def execute(self):

        """
        Executes the action_block in the PinFile
        The following logic just prints out path of the script being used
```

(continues on next page)

(continued from previous page)

```
        """

        self.load()
        extra_vars = {}
        runners = []

        print("This is the custom hook that runs custom logic")

        for action in self.action_data["actions"]:
            path = self.action_data["path"]
            script = action.get("script")
            print(script)
            print(path)
```

Tutorials

There are several tutorials available to help you learn how to use LinchPin.

Provisioning AWS EC2 with LinchPin

LinchPin can be used to provision compute instances on Amazon Web Services. If you need to familiarize yourself with EC2, read this. Now let's step through the process of creating a new workspace for provisioning EC2

Fetch

It is possible that you want to access a workspace that already exists. If that workspace exists online, `linchpin fetch` can be used to clone the repository. For example, the OpenShift on OpenStack example from release 1.7.2 in the linchpin repository can be cloned as follows:

```
$ linchpin fetch --root docs/source/examples/workspaces openshift-on-openstack --
↪branch 1.7.2 --dest ./fetch-example https://github.com/CentOS-PaaS-SIG/linchpin
```

You can even choose to fetch only a certain component of the workspace. For example, if you only wish to fetch the topologies you can add `--type topologies`. If you were able to fetch a complete workspace, you can skip to *Up*

Initialization

Assuming you are creating a workspace from scratch, you can run `linchpin init` to initialize a workspace. The following line of code will create a linchpin.conf, dummy PinFile, and README.rst in a directory called "simple"

```
$ linchpin init simple
```

The PinFile contains a single target, called simple, which contains a topology but no layout. A group of related provisioning tasks is called a target. Each target has a topology, which can contain many resource groups, and an optional layout. We'll explain what each of those means later on in further detail

Creating a Topology

Now that we have a PinFile, its time to add the code for an AWS EC2 instance. Edit your PinFile so it looks like the one below.

```
---
simple:
  topology:
    topology_name: simple
    resource_groups:
      - resource_group_name: aws_simple
        resource_group_type: aws
        resource_definitions:
          - name: simple_ec2
            role: aws_ec2
            flavor: m1.small
            count: 1
```

There are a number of other fields available for these two roles. Information about those fields as well as the other AWS roles can be found on the AWS provider page.

A resource group is a group of resources related to a single provider. In this example we have an AWS resource group that defines one type of AWS resources. We could also define an OpenStack resource group below it that provisions a handful of OpenStack Server nodes. A single resource group can contain many resource definitions. A resource definition details the requirements for a specific resource. We could add another resource definition to this topology to create a security group for our EC2 nodes. Multiple resources can be provisioned from a single resource definition by editing the `count` field, but all non-unique properties of the resources will be identical. So the flavor will be the same, but each node will have a unique name. The name will be {{ name }}_0, {{ name }}_1, etc. from 0 to count.

Credentials

Finally, we need to add credentials to the resource group. 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
        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 **:dirs1.5:`default_config_path <workspace/linchpin.conf#L22>`** value, and can be overridden in the **:docs1.5:`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
```

Creating a Layout

LinchPin can use layouts to describe what an Ansible inventory might look like after provisioning. Layouts can include information such as IP addresses, zones, and FQDNs. Under the simple key, put the following data:

```
---
layout:
  inventory_layout:
    vars:
      hostname: __IP__
    hosts:
      server:
        count: 1
        host_groups:
          - frontent
    host_groups:
      all:
        vars:
          ansible_user: root
        frontend:
          vars:
            ansible_ssh_common_args: -o StrictHostKeyChecking=no
```

After provisioning the hosts, LinchPin will through each host type in the inventory_layout, pop `count` hosts off of the list, and add them to the relevant host groups. The `host_groups` section of the layout is used to set environment variables for each of the hosts in a given host group

Up

Once the resources have been defined, LinchPin can be run as follows:

```
$ linchpin --workspace . -vv up simple
```

The `--workspace` flag references the relevant workspace. By default, the workspace is the current working directory. If the PinFile has a name (or path) other than {{workspace}}/PinFile, the `--pinfile` flag can override that. Finally, `-vv` sets a verbosity level of 2. As with Ansible, the verbosity can be set between 0 and 4.

If the provisioning was successful, you should see some output at the bottom that looks something like this:

```
ID: 122
Action: up

Target                    Run ID   uHash    Exit Code
-------------------------------------------------
simple                       1     3a0c59           0
```

You can use that uhash value to get the inventory generated according to the layout we discussed above. The file will be titled `inventories/${target}-${uhash}` but you can change this naming schema by editing the `inventory_file` field in the `inventory_layout` section of the layout. When `linchpin up` is run, each target will generate its own inventory layout. The inventories folder and inventory_path can also be set in the *evars* section of linchpin.conf

Destroy

At some point you'll no longer need the machines you provisioned. You can destroy the provisioned machines with `linchpin destroy`. However, you may not want to remove every single target from your last provision. For example, lets say you ran the simple provision above, then ran a few others. You could use the transaction ID, labeled "ID" above, to do so.

```
$ linchpin -vv destroy -t 122
```

You may also have provisioned multiple targets at once. If you only want to destroy one of them, you can do so with the name of the target and the run ID.

```
$ linchpin -vv destroy -r 1 simple
```

Journal

Each time you provision or destroy resources with LinchPin, information about the run is stored in the Run Database, or RunDB. Data from the RunDB can be printed using `linchpin journal`. This allows you to keep track of which resources you have provisioned but haven't destroyed and gather the transaction and run IDs for those resources. To list each resource by target, simply run:

```
$ linchpin journal

Target: simple
run_id      action           uhash              rc
--------------------------------------------------
2         destroy          bb8064               0
1              up          bb8064               0

Target: beaker-openstack
run_id      action           uhash              rc
--------------------------------------------------
2         destroy          b1e364               2
1              up          b1e364               2

Target: os-subnet
run_id      action           uhash              rc
--------------------------------------------------
3         destroy          c619ac               0
2              up          c619ac               0
1         destroy          ab9d81               0
```

As you can see, linchpin printed out the run data for the `simple` target that we provisioned and destroyed above, but also printed out information for a number of other targets which had been provisioned recently. You can provide a target as an argument to only print out the given target. You can also group by transaction id with the flag `--view tx`. Click here to read more about linchpin journal

Provisioning Beaker Server with LinchPin

LinchPin can be used to provision compute instances on Beaker. If you need to familiarize yourself with Beaker Server, read this. Now let's step through the process of creating a new workspace for provisioning Beaker

Fetch

It is possible that you want to access a workspace that already exists. If that workspace exists online, `linchpin fetch` can be used to clone the repository. For example, the OpenShift on OpenStack example from release 1.7.2 in the linchpin repository can be cloned as follows:

```
$ linchpin fetch --root docs/source/examples/workspaces openshift-on-openstack --
↪branch 1.7.2 --dest ./fetch-example https://github.com/CentOS-PaaS-SIG/linchpin
```

You can even choose to fetch only a certain component of the workspace. For example, if you only wish to fetch the topologies you can add `--type topologies`. If you were able to fetch a complete workspace, you can skip to *Up*

Initialization

Assuming you are creating a workspace from scratch, you can run `linchpin init` to initialize a workspace. The following line of code will create a linchpin.conf, dummy PinFile, and README.rst in a directory called "simple"

```
$ linchpin init simple
```

The PinFile contains a single target, called simple, which contains a topology but no layout. A group of related provisioning tasks is called a target. Each target has a topology, which can contain many resource groups, and an optional layout. We'll explain what each of those means later on in further detail

Creating a Topology

Now that we have a PinFile, its time to add the code for a Beaker server. Edit your PinFile so it looks like the one below.

```
---
simple:
  topology:
    topology_name: simple
    resource_groups:
      - resource_group_name: bkr_simple
        resource_group_type: beaker
        resource_definitions:
          - role: bkr_server
            recipesets:
              - distro: RHEL-7.5
                name: rhelsimple
                arch: x86_64
                variant: Server
                count: 1
                hostrequires:
                  - rawxml: '<key_value key="model" op="=" value="KVM"/>'
```

There are a number of other fields available for these two roles. Information about those fields as well as the other Beaker roles can be found on the Beaker provider page.

A resource group is a group of resources related to a single provider. In this example we have a Beaker resource group that defines two different types of Beaker resources. We could also define an AWS resource group below it that provisions a handful of EC2 nodes. A single resource group can contain many resource definitions. A resource definition details the requirements for a specific resource. Multiple resources can be provisioned from a single resource definition by editing the count field, but all non-unique properties of the resources will be identical. So the distro will be the same, but each node will have a unique name. The name will be {{ name }}_0, {{ name }}_1, etc. from 0 to count.

Credentials

Finally, we need to add credentials to the resource group.

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.

Creating a Layout

LinchPin can use layouts to describe what an Ansible inventory might look like after provisioning. Layouts can include information such as IP addresses, zones, and FQDNs. Under the simple key, put the following data:

```
---
layout:
  inventory_layout:
    vars:
      hostname: __IP__
    hosts:
      server:
        count: 1
        host_groups:
          - frontent
    host_groups:
      all:
        vars:
          ansible_user: root
        frontend:
          vars:
            ansible_ssh_common_args: -o StrictHostKeyChecking=no
```

After provisioning the hosts, LinchPin will through each host type in the inventory_layout, pop `count` hosts off of the list, and add them to the relevant host groups. The `host_groups` section of the layout is used to set environment variables for each of the hosts in a given host group

Up

Once the resources have been defined, LinchPin can be run as follows:

```
$ linchpin --workspace . -vv up simple
```

The `--workspace` flag references the relevant workspace. By default, the workspace is the current working directory. If the PinFile has a name (or path) other than {{workspace}}/PinFile, the `--pinfile` flag can override that. Finally, `-vv` sets a verbosity level of 2. As with Ansible, the verbosity can be set between 0 and 4.

If the provisioning was successful, you should see some output at the bottom that looks something like this:

```
ID: 122
Action: up

Target                   Run ID  uHash   Exit Code
-------------------------------------------------
simple                        1    3a0c59          0
```

You can use that uhash value to get the inventory generated according to the layout we discussed above. The file will be titled `inventories/${target}-${uhash}` but you can change this naming schema by editing the `inventory_file` field in the `inventory_layout` section of the layout. When `linchpin up` is run, each target will generate its own inventory layout. The inventories folder and inventory_path can also be set in the *evars* section of linchpin.conf

Destroy

At some point you'll no longer need the machines you provisioned. You can destroy the provisioned machines with `linchpin destroy`. However, you may not want to remove every single target from your last provision. For example, lets say you ran the simple provision above, then ran a few others. You could use the transaction ID, labeled "ID" above, to do so.

```
$ linchpin -vv destroy -t 122
```

You may also have provisioned multiple targets at once. If you only want to destroy one of them, you can do so with the name of the target and the run ID.

```
$ linchpin -vv destroy -r 1 simple
```

Journal

Each time you provision or destroy resources with LinchPin, information about the run is stored in the Run Database, or RunDB. Data from the RunDB can be printed using `linchpin journal`. This allows you to keep track of which resources you have provisioned but haven't destroyed and gather the transaction and run IDs for those resources. To list each resource by target, simply run:

```
$ linchpin journal

Target: simple
run_id      action           uhash              rc
-------------------------------------------------
2         destroy          bb8064               0
1              up          bb8064               0
```

(continues on next page)

(continued from previous page)

```
Target: beaker-openstack
run_id      action           uhash               rc
-------------------------------------------------
2         destroy          b1e364               2
1              up          b1e364               2

Target: os-subnet
run_id      action           uhash               rc
-------------------------------------------------
3         destroy          c619ac               0
2              up          c619ac               0
1         destroy          ab9d81               0
```

As you can see, linchpin printed out the run data for the `simple` target that we provisioned and destroyed above, but also printed out information for a number of other targets which had been provisioned recently. You can provide a target as an argument to only print out the given target. You can also group by transaction id with the flag `--view tx`. Click here to read more about linchpin journal

Provisioning OpenStack Server with LinchPin

LinchPin can be used to provision compute instances on OpenStack. If you need to familiarize yourself with OpenStack Server, read this. Now let's step through the process of creating a new workspace for provisioning OpenStack

Fetch

It is possible that you want to access a workspace that already exists. If that workspace exists online, `linchpin fetch` can be used to clone the repository. For example, the OpenShift on OpenStack example from release 1.7.2 in the linchpin repository can be cloned as follows:

```
$ linchpin fetch --root docs/source/examples/workspaces openshift-on-openstack --
↪branch 1.7.2 --dest ./fetch-example https://github.com/CentOS-PaaS-SIG/linchpin
```

You can even choose to fetch only a certain component of the workspace. For example, if you only wish to fetch the topologies you can add `--type topologies`. If you were able to fetch a complete workspace, you can skip to *Up*

Initialization

Assuming you are creating a workspace from scratch, you can run `linchpin init` to initialize a workspace. The following line of code will create a linchpin.conf, dummy PinFile, and README.rst in a directory called "simple"

```
$ linchpin init simple
```

The PinFile contains a single target, called simple, which contains a topology but no layout. A group of related provisioning tasks is called a target. Each target has a topology, which can contain many resource groups, and an optional layout. We'll explain what each of those means later on in further detail

Creating a Topology

Now that we have a PinFile, its time to add the code for an OpenStack server. Edit your PinFile so it looks like the one below.

```
simple:
  topology:
    topology_name: simple
    resource_groups:
      - resource_group_name: os_simple
        resource_group_type: openstack
        resource_definitions:
          - name: simple_keypair
            role: os_keypair
          - name: simple_server
            role: os_server
            flavor: m1.small
            keypair: simple_keypair
            count: 1
```

There are a number of other fields available for these two roles. Information about those fields as well as the other OpenStack roles can be found on the OpenStack provider page.

A resource group is a group of resources related to a single provider. In this example we have an openstack resource group that defines two different types of openstack resources. We could also define an AWS resource group below it that provisions a handful of EC2 nodes. A single resource group can contain many resource definitions. A resource definition details the requirements for a specific resource. Multiple resources can be provisioned from a single resource definition by editing the `count` field, but all non-unique properties of the resources will be identical.

Credentials

Finally, we need to add credentials to the resource group. OpenStack 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 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.

Using OpenStack Credentials

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.

It's worth noting that we can't give you credentials to use, so you'll have to provide your own filename and profile here. By default, LinchPin searches for the filename in {{ workspace}}/credentials but can be made to search other places by setting the `evars.default_credentials_path` variable in your linchpin.conf. The credentials path can also be overridden by using the `--creds-path` flag.

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

      .. snip ..

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

Creating a Layout

LinchPin can use layouts to describe what an Ansible inventory might look like after provisioning. Layouts can include information such as IP addresses, zones, and FQDNs. Under the simple key, put the following data:

```
---
layout:
  inventory_layout:
    vars:
      hostname: __IP__
    hosts:
      server:
        count: 1
```

(continues on next page)

(continued from previous page)

```
      host_groups:
        - frontent
  host_groups:
    all:
      vars:
        ansible_user: root
      frontend:
        vars:
          ansible_ssh_common_args: -o StrictHostKeyChecking=no
```

After provisioning the hosts, LinchPin will through each host type in the inventory_layout, pop `count` hosts off of the list, and add them to the relevant host groups. The `host_groups` section of the layout is used to set environment variables for each of the hosts in a given host group

Up

Once the resources have been defined, LinchPin can be run as follows:

```
$ linchpin --workspace . -vv up simple
```

The `--workspace` flag references the relevant workspace. By default, the workspace is the current working directory. If the PinFile has a name (or path) other than {{workspace}}/PinFile, the `--pinfile` flag can override that. Finally, `-vv` sets a verbosity level of 2. As with Ansible, the verbosity can be set between 0 and 4.

If the provisioning was successful, you should see some output at the bottom that looks something like this:

```
ID: 122
Action: up

Target                   Run ID  uHash   Exit Code
-------------------------------------------------
simple                      1    3a0c59          0
```

You can use that uhash value to get the inventory generated according to the layout we discussed above. The file will be titled `inventories/${target}-${uhash}` but you can change this naming schema by editing the `inventory_file` field in the `inventory_layout` section of the layout. When `linchpin up` is run, each target will generate its own inventory layout. The inventories folder and inventory_path can also be set in the *evars* section of linchpin.conf

Destroy

At some point you'll no longer need the machines you provisioned. You can destroy the provisioned machines with `linchpin destroy`. However, you may not want to remove every single target from your last provision. For example, lets say you ran the simple provision above, then ran a few others. You could use the transaction ID, labeled "ID" above, to do so.

```
$ linchpin -vv destroy -t 122
```

You may also have provisioned multiple targets at once. If you only want to destroy one of them, you can do so with the name of the target and the run ID.

```
$ linchpin -vv destroy -r 1 simple
```

Journal

Each time you provision or destroy resources with LinchPin, information about the run is stored in the Run Database, or RunDB. Data from the RunDB can be printed using `linchpin journal`. This allows you to keep track of which resources you have provisioned but haven't destroyed and gather the transaction and run IDs for those resources. To list each resource by target, simply run:

```
$ linchpin journal

Target: simple
run_id      action           uhash              rc
--------------------------------------------------
2         destroy          bb8064               0
1              up          bb8064               0

Target: beaker-openstack
run_id      action           uhash              rc
--------------------------------------------------
2         destroy          b1e364               2
1              up          b1e364               2

Target: os-subnet
run_id      action           uhash              rc
--------------------------------------------------
3         destroy          c619ac               0
2              up          c619ac               0
1         destroy          ab9d81               0
```

As you can see, linchpin printed out the run data for the `simple` target that we provisioned and destroyed above, but also printed out information for a number of other targets which had been provisioned recently. You can provide a target as an argument to only print out the given target. You can also group by transaction id with the flag `--view tx`. Click here to read more about linchpin journal

See also:

Commands (CLI) Linchpin Command-Line Interface

workflow Common LinchPin Workflows

Managing Resources Managing Resources

Examples for all Providers Providers in Detail

1.1.8 Monitor and Progress Bar

Linchpin execution of Ansible is mostly a black box, where Ansible receives input from Linchpin and returns expected output. The output is received in a form of files and database changes. However, in version 1.9.1 there was another channel of communication was created, a message bus. Before version 1.9.1, Linchpin was calling Ansible in a synchronize mode, that is once Ansible was called, Linchpin was waiting for it to finish the execution. To support progress bar, ZMQ message bus and multiprocessing was added. From version 1.9.1, Linchpin by default runs Ansible in multiprocess with a "monitoring" process. The ZMQ message bus was added to Ansible using plugins, and to the monitoring process. That means that Ansible, on different events or steps will able to communicate with Linchpin. For progress bar it meant that Ansible could update Linchpin with its progress in details, which allows better user experience and understanding of deployment or tear down progress. The new functionality is limited to provisioning process ('up' and 'destroy') and can be disabled or limited with options –no-monitor or –no-progress:

–no-monitor will disable multiprocessing entirely and thus also disables the progress bar.

–no-progress will cancel the progress bar which could be helpful in shell scripts or in CI, but the monitoring/multiprocessing remains.

Examples:

```
# Linchpin runs with multiprocessing and progress bar enabled
linchpin up

# Linchpin runs in verbose mode, progress bar disabled
linchpin -vvvv up

# Linchpin runs with disabled multiprocessing and without progress bar
linchpin --no-monitor up

# Linchpin runs without progress bar but with multiprocessing
linchpin up --no-progress
```

The progress bar and multiprocessing can be disabled via linchpin.conf settings file:

```
[progress_bar]
no_progress = True

[monitor]
no_monitor = True
```

1.1.9 Linchpin API (until 1.7.5)

LinchPin can be used to provision resources by invoking linchpin python API.

Provisioning example using a Pinfile

While provisioning with a Pinfile as a dictionary we have to set various config parameters and workspaces as follows.

```
from linchpin import LinchpinAPI
from linchpin.context import LinchpinContext

context = LinchpinContext()
context.setup_logging()
context.load_config()
context.load_global_evars()
context.set_cfg('lp', 'workspace', '.')
context.set_evar('workspace', '.')
context.set_evar('debug_mode', True)
linchpin_api = LinchpinAPI(context)
pindict = {
  "simple": {
    "layout": {
      "inventory_layout": {
        "hosts": {
          "example-node": {
            "count": 1,
            "host_groups": [
              "example"
            ]
          }
```

(continues on next page)

(continued from previous page)

```
        },
        "vars": {
          "hostname": "__IP__",
          "ansible_ssh_private_key_file": "~/.ssh/id_rsa"
        }
       }
    },
    "topology": {
       "topology_name": "simple",
      "resource_groups": [
        {
          "resource_group_name": "os-server-new",
          "resource_definitions": [
            {
              "count": 1,
              "name": "database",
              "image": "CentOS-7-x86_64-GenericCloud-1612",
              "keypair": "ci-factory",
              "role": "os_server",
              "fip_pool": "10.8.240.0",
              "flavor": "m1.small",
              "networks": [
                "QE-test"
              ]
            }
          ],
          "resource_group_type": "openstack",
          "credentials": {
            "filename": "clouds.yaml",
            "profile": "default"
          }
        }
      ]
    }
  }
}

# credentials alternatives: file vs environment variables
linchpin_api.do_action(pindict, action='up')
# inorder to destroy the pinfile we need to pass action parameter as destroy
# linchpin_api.do_action(pindict, action='destroy')
```

1.1.10 Linchpin revised API (Preview in 1.7.6)

In linchpin new api restructure linchpin provides two classes Pinfile, Workspace to provision resources

This feature is currently in Preview state for 1.7.6 will be available from version 2.0

Examples for provisioning using linchpin api Pinfile and workspace are as follows

```
import json
import linchpin
from linchpin.api import Pinfile
from linchpin.api import Workspace

# workspace requires workspace path
```

(continues on next page)

(continued from previous page)

```
  wksp = Workspace(path="/tmp/tmp3BAAhC/")
  wksp.up()
  #prints the inventory generated after provisioning
  wksp.get_inventory(inv_format="json")
  wksp.destroy()

  # Provisioning with Pinfile structure

  pinfile="""
  dummy-test:
    topology:
      topology_name: "dummy_cluster" # topology name
      resource_groups:
      - resource_group_name: "dummy"
        resource_group_type: "dummy"
        resource_definitions:
        - name: "web"
          role: "dummy_node"
          count: 3
        - name: "test"
          role: "dummy_node"
          count: 1
    layout:
      inventory_layout:
        vars:
          hostname: __IP__
        hosts:
          example-node:
            count: 3
            host_groups:
            - example
          test-node:
            count: 1
            host_groups:
            - test
        host_groups:
          all:
            vars:
              ansible_user: root
  """
  import yaml
  pinfile = yaml.load(pinfile)
  pf = Pinfile(pinfile=pinfile)
  print(pf.validate())
  #pf.up()
  #pf.destroy()

# workspace with external credential path
  wsp = Workspace(path="/home/srallaba/workspace/lp_ws_backup/lp_ws/ex_hooks/testw/
↪dummy-creds-vault")
  print(wsp.validate())
  wsp.set_creds_path("/home/srallaba/workspace/lp_ws_backup/lp_ws/ex_hooks/testw/
↪dummy-creds-vault/credentials/")
  wsp.set_evar("vault_password","testval")
  wsp.up()
  wsp.get_inventory()
  wsp.destroy()
```

Note: The both examples provided are backward compatible in nature. Introduction of new API does not change functionality the existing API

Refer the API reference section here *Linchpin API and Context Modules* for more documentation on specific functions

See also:

Commands (CLI) Linchpin Command-Line Interface

workflow Common LinchPin Workflows

Managing Resources Managing Resources

Examples for all 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.

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).

Preview Feature:

linchpin up and destroy includes –use-shell parameter which makes linchpin run as a subprocess rather than ansible api call usefull when we would like to overwrite environment varibles

```
$ linchpin -vvvv up --use-shell --env-vars TESTENV testenv value
```

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 folder with All in one PinFile which includes topology, and layout structure.

```
$ pwd
/tmp/workspace
$ linchpin init
Created destination workspace <path>
$ tree

├── dummy
```

(continues on next page)

(continued from previous page)

```
    ├── PinFile
    ├── PinFile.json
    └── README.rst
└── linchpin.log
```

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 *Linchpin 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 **:dirs1.5:`the source code <workspace/topologies>`**.

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 **:dirs1.5:`the source code <workspace/layouts>`**.

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.

```
# Example 1
dummy_cluster:
  topology: dummy-topology.yml
  layout: dummy-layout.yml

# Example 2
dummy-topo:
  topology:
    topology_name: "dummy_cluster" # topology name
    resource_groups:
    - resource_group_name: "dummy"
      resource_group_type: "dummy"
      resource_definitions:
      - name: "{{ distro | default('') }}web"
        role: "dummy_node"
        count: 3
      - name: "{{ distro | default('') }}test"
        role: "dummy_node"
        count: 1
  layout:
    inventory_layout:
      vars:
        hostname: __IP__
      hosts:
        example-node:
          count: 3
          host_groups:
          - example
        test-node:
          count: 1
```

(continues on next page)

(continued from previous page)

```
          host_groups:
          - test
```

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

To use a PinFile with an Ansible Galaxy role, simply provide the role name as the *resource_group_type*. An example is shown below.

```
dummy-new:
  topology:
    topology_name: "dummy_cluster" # topology name
    resource_groups:
      - resource_group_name: "dummy"
        resource_group_type: "14rcole.ansible_role_lp_dummy"
        resource_definitions:
          - name: "{{ distro | default('') }}web"
            role: "dummy_node"
            count: 3
          - name: "{{ distro | default('') }}test"
            role: "dummy_node"
            count: 1
```

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

Additional PinFile examples can be found in **:dirs1.5:`the source code <workspace>`**

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
```

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 *Examples for all 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.

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
```

(continues on next page)

(continued from previous page)

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

See also:

Examples for all 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 **:dirs1.5:`default_config_path <workspace/linchpin.conf#L22>`** value, and can be overridden in the **:docs1.5:`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

Examples for all Providers Providers in Detail

1.2.2 General Configuration

Managing LinchPin requires a few configuration files. Most configurations are stored in the **:code1.5:`linchpin configuration <linchpin/linchpin.constants>`** 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": [],
```

(continues on next page)

(continued from previous page)

```
                "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 *Examples for all 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 folder with All in one PinFile which includes topology, and layout structure.

```
$ pwd
/tmp/workspace
$ linchpin init
Created destination workspace <path>
$ tree

├── dummy
│   ├── PinFile
```

(continues on next page)

(continued from previous page)

```
│   ├── PinFile.json
│   └── README.rst
└── linchpin.log
```

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-new            83      a18e9a              0
dummy-topo           70      044695              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
```

A subset of the hosts in a PinFile can be provisioned by listing each of them at the end of the command

```
$ linchin -vv up dummy-new

Target               Run ID  uHash       Exit Code
-------------------------------------------------
dummy-new            83      a18e9a              0
```

Preview Feature:

linchpin up and destroy includes –use-shell parameter which makes linchpin run as a subprocess rather than ansible api call usefull when we would like to overwrite environment varibles

```
$ linchpin -vvvv up dummy-new --use-shell --env-vars TESTENV testenv value
```

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:

Examples for all 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.
```

(continues on next page)

(continued from previous page)

```
  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)
  -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
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------
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
```

(continues on next page)

(continued from previous page)

```
                              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
```

(continues on next page)

(continued from previous page)

```
        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
```

(continues on next page)

(continued from previous page)

```
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
```

linchpin setup

Some providers require additional dependencies installed on the system running linchpin. Use `linchpin setup` to setup the given provider(s) properly.

If a list of providers is ommitted, then it will install dependencies for all providers that need so.

In case you execute `linchpin setup` with a user not allowed to install packages, then pass the *–ask-sudo-pass* option to prompt for the sudo password.

linchpin ssh

The `linchpin ssh` command provides a simple way to connect to provisioned systems. Instead of looking for the system in the inventory file and writing an ssh command, it is easy as writing `linchpin ssh`, hitting <TAB><TAB> and selecting the system. The double tab works with linchpin auto-complete that can be enabled by running: `eval "$(_LINCHPIN_COMPLETE=source linchpin)"`

The SSH command will look for the latest inventory generated by Linchpin for connection information.

```
$ linchpin ssh --help
Usage: linchpin ssh [OPTIONS] TARGET

Options:
  -h, --help  Show this message and exit.
```

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 topologies, 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:

- **:docs1.5:`PinFile.dummy.yml <workspace/PinFile.dummy.yml>`**
- **:docs1.5:`PinFile.openstack.yml <workspace/PinFile.openstack.yml>`**
- **:docs1.5:`PinFile.complex.yml <workspace/PinFile.complex.yml>`**

The combined format is only available in v1.5.0+

- **:docs1.5:`PinFile.combined.yml <workspace/PinFile.combined.yml>`**

JSON

New in version 1.5.0

PinFiles written using JSON format.

- **:docs1.5:`PinFile.dummy.json <workspace/PinFile.dummy.json>`**
- **:docs1.5:`PinFile.aws.json <workspace/PinFile.aws.json>`**
- **:docs1.5:`PinFile.duffy.json <workspace/PinFile.duffy.json>`**
- **:docs1.5:`PinFile.combined.json <workspace/PinFile.combined.json>`**
- **:docs1.5:`PinFile.complex.json <workspace/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*.

- **:docs1.5:`PinFile.beaker.template <workspace/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.

- **:docs1.5:`PinFile.libvirt-mi.template <workspace/PinFile.libvirt-mi.template>`**
- **:docs1.5:`Data.libvirt-mi.yml <workspace/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.

- **:docs1.5:`generate_dummy.sh <workspace/scripts/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:

- **:docs1.5:`os-server-new.yml <workspace/topologies/os-server-new.yml>`**
- **:docs1.5:`libvirt-new.yml <workspace/topologies/libvirt-new.yml>`**
- **:docs1.5:`bkr-new.yml <workspace/topologies/bkr-new.yml>`**

Older topologies, supported in v1.5.0+

- **:docs1.5:`os-server.yml <workspace/topologies/os-server.yml>`**
- **:docs1.5:`libvirt.yml <workspace/topologies/libvirt.yml>`**
- **:docs1.5:`bkr.yml <workspace/topologies/bkr.yml>`**

JSON

New in version 1.5.0

Topologies can be written using JSON format.

- **:docs1.5:`dummy.json <workspace/topologies/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*.

- **:docs1.5:`jenkins-slave.j2 <workspace/topologies/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:

:docs1.5:`Pinfile.jenkins.j2 <workspace/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:

- **:docs1.5:`aws-ec2.yml <workspace/layouts/aws-ec2.yml>`**
- **:docs1.5:`dummy-new.yml <workspace/layouts/dummy-new.yml>`**

JSON

New in version 1.5.0

Layouts can be written using JSON format.

- **:docs1.5:`gcloud.json <workspace/layouts/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*.

- **:docs1.5:`dummy.json <workspace/layouts/dummy.json>`**

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

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

See also:

:docs1.5:`PinFile.dummy.json <workspace/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:

Examples for all Providers

1.2.5 Examples for all 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.

- **:docs1.5:`Topology Example <workspace/topologies/os-server-new.yml>`**
- 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 is 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	Name of the instance
flavor	true	string	flavor	Defines the compute, memory, and storage capacity of the node
image	true	string	image	The disk image used to provision the server instances
region	false	string	region	
count	false	integer	count	
keypair	false	string	key_name	Public key of an OpenSSH keypair to be used for access to created servers
security_groups	false	string	security_groups	
fip_pool	false	string	floating_ip_pools	
networks	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.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-obj-new.yml>`**
- Ansible module

os_vol

OpenStack Cinder Volumes can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-vol-new.yml>`**
- Ansible module

os_sg

OpenStack Security Groups can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-sg-new.yml>`**
- Ansible Security Group module
- Ansible Security Group Rule module

os_network

OpenStack networks can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-network.yml>`**
- Ansible os_network module

os_router

OpenStack routers can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-router.yml>`**
- Ansible os_router module

os_subnet

OpenStack subnets can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-subnet.yml>`**
- Ansible os_subnet module

os_keypair

OpenStack keypairs can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/openstack/topologies/os-keypair.yml>`**
- Ansible os_keypair 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 provisioning 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.

- **:docs1.5:`Topology Example <workspace/topologies/libvirt-new.yml>`**
- 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.

libvirt_network

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

- **:docs1.5:`Topology Example <workspace/topologies/libvirt-el7net.yml>`**
- 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		
delete_on_destroy	false	boolean	N/A	False	If true, libvirt destroy will destroy and undefine the network

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 **:docs1.5:`linchpin.conf <workspace/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 and qemu groups

   ```
   $ usermod -aG libvirt,qemu <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.

- **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-new.yml>`**
- **:docs1.5:`Topology Example w/ VPC <workspace/topologies/aws-ec2-vpc.yml>`**
- 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	boolean	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.

- **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-key-new.yml>`**
- 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.

- **:docs1.5:`Topology Example <workspace/topologies/aws-s3-new.yml>`**
- 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.

- **:docs1.5:`Topology Example <workspace/topologies/aws-sg-new.yml>`**
- *ec2_group module <http://docs.ansible.com/ansible/latest/ec2_group_module.html>*

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_ec2_eip

AWS EC2 elastic ips can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-eip.yml>`**
- *ec2_eip module <http://docs.ansible.com/ansible/latest/ec2_eip_module.html>*

aws_ec2_vpc_net

AWS VPC networks can be provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/aws-ec2-vpc-net.yml>`**
- `ec2_vpc_net module <https://docs.ansible.com/ansible/latest/modules/ec2_vpc_net_module.html

>`_

aws_ec2_vpc_internet_gateway

Manage AWS VPC INTERNET Gateways. * **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-vpc-internet-gateway.yml>`** * *ec2_vpc_net module <https://docs.ansible.com/ansible/latest/modules/ec2_vpc_igw_module.html>*

aws_ec2_vpc_nat_gateway

Manage AWS VPC NAT Gateways.

- **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-vpc-nat-gateway.yml>`**
- ec2_vpc_net module

aws_ec2_vpc_subnet

AWS VPC subnets can be provisioned using this resource. * **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-vpc-subnet.yml>`** * ec2_vpc_subnet module

aws_ec2_vpc_routetable

AWS VPC routetable can be provisioned using this resource. * **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-vpc-routetable.yml>`** * ec2_vpc_route_table module

aws_ec2_vpc_endpoint

AWS VPC endpoint can be provisioned using this resource. * **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-vpc-endpoint.yml>`** * ec2_vpc_endpoint module

aws_ec2_elb_lb

AWS EC2 elb lb load balancer can be provisioned using this resource. * **:docs1.5:`Topology Example <workspace/topologies/aws-ec2-elb-lb.yml>`** * ec2_vpc_endpoint module

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:

Examples for all 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 **:dirs1.5:`default_config_path <workspace/linchpin.conf#L22>`** value, and can be overridden in the **:docs1.5:`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
```

Azure

The Azure provider manages multiple types of resources.

Note: The dependencies is perfectly working for the latest version of Ansible, if you are not using the latest version, may not work.

---

azure_vm

Azure VM Instances can be provisioned using this resource.

- Example
- azure_vm module

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
vm_name	true	string	name	It can't include '_' and other special char
private_image	false	string	image	This takes private images
virtual_network_name	false	string	virtual_network_name	
vm_username	false	string	image	
vm_password	false	string	image	
count	false	int		
resource_group	true	string	resource_group	
vm_size	false	string	vm_size	
public_image	false	dict	image	This para takes public images
vm_username	false	string	admin_username	
vm_password	false	string	admin_password	
public_key	false	string		Copy you key here
delete_all_attached	false	string	remove_on_absent	
availability_set	false	string	availability_set	

azure_nsg

Azure Network Security Group can be provisioned using this resource.

- Example <workspaces/azure/Pinfile>`
- azure_nsg module <https://docs.ansible.com/ansible/latest/modules/azure_rm_securitygroup_module.html?highlight=azure%20security#examples>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
name	true	string	name	
purge_rules	false	string	purge_rules	
rules	false	list(dict)	rules	

- If you declare both public and private image, only the private will be taken

azure_api

Any Azure resources can be provisioned using this role, it supported by the Azure Api

- Example
- azure_api module
- Azure API

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	true	String	resource_group	
resource_type	true	String	resource_type	
resource_name	true	string	resource_name	
api_version	true	string	api_version	
body_path	true	string		Path to request body
url	true	string	url	

azure_loadbalancer

With this role you can provision and configure the Azure Load Balancer

- Example <workspaces/azure/Pinfile>`
- azure_loadbalancer module <https://docs.ansible.com/ansible/latest/modules/azure_rm_loadbalancer_module.html?highlight=azure%20load%20balance>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
frontend_ip_configuration	false	string	frontend_ip_configuration	
backend_address_pools	false	string	backend_address_pools	
probes	false	string	probes	
inbound_nat_pools	false	string	inbound_nat_pools	
inbound_nat_rules	false	string	inbound_nat_rules	
load_balacing_rules	false	string	load_balacing_rules	

azure_publicipaddress

With this role, you can provision and manage Azure public ip address

- Example <workspaces/azure/Pinfile>`
- azure_publicipaddress module <https://docs.ansible.com/ansible/latest/modules/azure_rm_publicipaddress_module.html?highlight=azure%20public%20address>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
allocation_method	true	string	allocation_method	
domain_name	false	string	domain_name	
sku	false	string	sku	

azure_availabilityset

Any Azure resources can be provisioned using this role, it supported by the Azure Api

- Example <workspaces/azure/Pinfile>`
- azure_availabilityset module <https://docs.ansible.com/ansible/latest/modules/azure_rm_availabilityset_module.html?highlight=azure%20avail>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
location	false	string	name	
platform_update_domain_count	false	string	platform_update_domain_count	
platform_fault_domain_count	false	string	platform_fault_domain_count	
sku	false	string	sku	

azure_network_interface

Azure network interface can be provisioned using this role

- Example <workspaces/azure/Pinfile>`
- azure_rm_networkinterface module <https://docs.ansible.com/ansible/latest/modules/azure_rm_networkinterface_module.html?highlight=azure%20network%20interface>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
virtual_network_name	false	string	virtual_network	
subnet_name	false	string	platform_update_domain_count	

azure_resource_group

Azure network interface can be provisioned using this role

- Example <workspaces/azure/Pinfile>`
- azure_rm_resourcegroup module <https://docs.ansible.com/ansible/latest/modules/azure_rm_resourcegroup_module.html?highlight=azure%20resource%20group>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
location	false	string	location	

azure_virtual_network

Azure virtual network can be provisioned using this role

- Example <workspaces/azure/Pinfile>`
- azure_rm_virtualnetwork module <https://docs.ansible.com/ansible/latest/modules/azure_rm_virtualnetwork_module.html?highlight=azure%20virtual%20network>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
address_prefixes	false	string	address_prefixes	

azure_virtual_subnet

Azure network interface can be provisioned using this role

- Example <workspaces/azure/Pinfile>`
- azure_rm_subnet module <https://docs.ansible.com/ansible/latest/modules/azure_rm_subnet_module.html?highlight=azure%20subnet>`_

Topology Schema

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

Parameter	required	type	ansible value	comments
role	true	string	N/A	
resource_group	false	string	resource_group	
name	true	string	name	
virtual_network_name	false	string	virtual_network_name	
address_prefix	false	string	address_prefix	

Credentials Management

Linchpin supports Ansible authentication options:

- Active Directory
- Service Principal

Active Directory

Active Directory authentication works only with organization users (not guests). You can create a new user in the organization but do not invite users. The following keys are required in the credentials file for AD authentication:

user The user name, you can verify it manually in Azure portal.

password The password, you can verify it manually in Azure portal and change it.

subscription_id The subscription id to use, you can check what subscriptions available and what permission you have in Azure portal.

tenant Is the Active Directory ID, and it is required if the user is member of multiple directories. You can find tenant ID in Azure portal at Azure Active Directory

Example of credentials file with Azure Active directory:

```
[default]
user: linchpin@redhat.com
password: MySecretPassword
subscription_id: 2q3d2d-ad3adw-adwa3d-dwade-awedawee
tenant: 3rfawca-awd3daw-d3cc33-ASCEA-CAEESA-caceace
```

Service Principal

The following keys are required in the credentials file for SP authentication:

client_id The client ID is the application ID.

secret The application secret token, can be generated in Azure portal

subscription_id The subscription id to use, you can check what subscriptions available and what permission you have in Azure portal.

tenant Is the Active Directory ID, and it is required if the user is member of multiple directories. You can find tenant ID in Azure portal at Azure Active Directory

Example of credentials file with Azure Service Principal:

```
[default]
client_id: 2q3d2d-ad3adw-adwa3d-dwade-awedawee
secret: 2q3d2d-ad3adw-adwa3d-dwade-awedawee
subscription_id: 2q3d2d-ad3adw-adwa3d-dwade-awedawee
tenant: 3rfawca-awd3daw-d3cc33-ASCEA-CAEESA-caceace
```

How to create new Service Principal in Azure portal

1. Go to Azure Active Directory in Azure portal
2. Go to *App registration* on the left bar
3. Create a new app
4. The Application ID is *client_id*
5. The Directory ID is *tenant*
6. Go to *Certificates and secrets* on left bar
7. Upload or create a new key, that is the *secret*
8. Go to the *Access Control* of you resource group or subscription
9. Click on *Add* button to add new role assignment
10. Assign the role of *Contributor* to the application you just created
11. Go to *Subscription* to find out its ID for subscription id

How to create new Service Principal using Azure command line client

```
accountname@Azure:~$ az ad sp create-for-rbac --name ServicePrincipalName
Changing "ServicePrincipalName" to a valid URI of "http://ServicePrincipalName",␣
↪which is the required format used for service principal names
Creating a role assignment under the scope of "/subscriptions/dcc74c29-4db6-4c49-9a0f-
↪ac0ee03fa17e"
  Retrying role assignment creation: 1/36
  Retrying role assignment creation: 2/36
  Retrying role assignment creation: 3/36
  Retrying role assignment creation: 4/36
{
  "appId": "xxxxxxxxxxxxxxxxxxxxxxxx",
  "displayName": "ServicePrincipalName",
  "name": "http://ServicePrincipalName",
  "password": "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxx",
  "tenant": "xxxxx-xxxxx-xxxx-xxxx-xxxxxxxxxxx"
}
```

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.

- **:docs1.5:`Topology Example <workspaces/topologies/gce-new.yml>`**
- Ansible module

gcloud_gce_eip

Google Compute enginer external IP (gce_eip) are provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/gce-eip.yml>`**
- *Ansible module <http://docs.ansible.com/ansible/latest/gce_eip_module.html>*

gcloud_gce_net

Google compute engine network (gce_net) are provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/gce-net.yml>`**
- *Ansible module <http://docs.ansible.com/ansible/latest/gce_net_module.html>*

gcloud_gcdns_zone

Google DNS zone (gcdns_zone) are provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/gcdns-zone.yml>`**
- *Ansible module <https://docs.ansible.com/ansible/latest/modules/gcdns_zone_module.html>*

gcloud_gcdns_record

Google DNS zone records (gcdns_record) are provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/gcdns-record.yml>`**
- *Ansible module <https://docs.ansible.com/ansible/latest/modules/gcdns_record_module.html>*

gcloud_gcp_compute_network

Google cloud compute networks are provisioned using this resource.

- **:docs1.5:`Topology Example <workspaces/topologies/gcp-compute-network.yml>`**
- *Ansible module <https://docs.ansible.com/ansible/latest/modules/gcp_compute_network_module.html>*

gcloud_gcp_compute_router

Google cloud compute routers are provisioned using this resource.

- **:docs1.5:`Topology Example <workspace/topologies/gcp-compute-router.yml>`**
- *Ansible module <https://docs.ansible.com/ansible/latest/modules/gcp_compute_router_module.html>*

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.

Google Cloud Key File

GCloud allows for the creation of keyfiles for authentication. A keyfile will look something like this:

```
{
  "type": "service_account",
  "project_id": "[PROJECT-ID]",
  "private_key_id": "[KEY-ID]",
  "private_key": "-----BEGIN PRIVATE KEY-----\n[PRIVATE-KEY]\n-----END PRIVATE KEY----
↪-\n",
  "client_email": "[SERVICE-ACCOUNT-EMAIL]",
  "client_id": "[CLIENT-ID]",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://accounts.google.com/o/oauth2/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/[SERVICE-
↪ACCOUNT-EMAIL]"
}
```

To learn how to generate key files, *see the google cloud documentation <https://cloud.google.com/iam/docs/creating-managing-service-account-keys>*.

This mechanism requires that credentials data be passed into LinchPin. A GCloud topology can have a `credentials` section for each resource_group, which requires the filename and the profile name. By default, LinchPin searches for the filename in {{ workspace }}/credentials but can be made to search other places by setting the `evars.default_credentials_path` variable in your linchpin.conf. The credentials path can also be overridden by using the `--creds-path` flag.

```
---
topology_name: mytopo
resource_groups:
  - resource_group_name: gce
  - resource_group_type: gcloud
    resource_definitions:

      .. snip ..
```

(continues on next page)

(continued from previous page)

```
    credentials:
      filename: gcloud.key
```

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.

VMware

The VMware provider manages a single resource, `vmware_guest`.

vmware_guest

VMware VMs can be provisioned using this resource

- **docs1.7** *Topology Example <workspace/topologies/vmware.yml>*
- *Ansible module https://docs.ansible.com/ansible/latest/modules/vmware_guest_module.html*

Topology Schema

Within Linchpin, the vmware_guest supports all the Ansible module options with the same schema structure. All the limitation of the module apply too.

Additional Dependencies

The vmware resources group requires additional dependency, the following must be installed:

- PyVmomi

```
$ pip install linchpin[vmware]
```

Credentials Management

Environment Variables

Linchpin honors the following environment variables:

Environment variable	Credentials variable	Description
VMWARE_PASSWORD	password	The password of the vSphere vCenter or ESXi server
VMWARE_USER	username	The username of the vSphere vCenter or ESXi server.
VMWARE_HOST	hostname	The hostname or IP address of the vSphere vCenter or ESXi server.
VMWARE_PORT	port	The port number of the vSphere vCenter or ESXi server.
VMWARE_VALIDATE_CERTS	validate_certs	Allows connection when SSL certificates are not valid.

Credentials File

An example credentials file may look like this for vmware.

```
$ cat vmware.key
  [default]
  username=root
  password=VMware1!
  hostname=192.168.122.125
  validate_certs=false
```

See also:

Examples for all 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 'vmware.key'. The user could pick either profile in that file.

```
---
topology_name: vmware-new
resource_groups:
  - resource_group_name: vmware-new
    resource_group_type: vmware
    resource_definitions:
      - role: vmware_guest
        name: vmware-node
        cdrom:
          type: iso
          iso_path: "[ha-datacenter] tc_vmware4.iso"
        folder: /
        datastore: ha-datacenter
        disk:
          - size_mb: 10
            type: thin
        hardware:
          num_cpus: 1
          memory_mb: 256
        networks:
          - name: VM Network
        wait_for_ip_address: yes
```

(continues on next page)

(continued from previous page)

```
    credentials:
      filename: vmware.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 **:dirs1.5:`default_config_path <workspace/linchpin.conf#L22>`** value, and can be overridden in the **:docs1.5:`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 vmware-new
```

Note: The `vmware.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 vmware-new
```

Beaker

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

bkr_server

Beaker instances are provisioned using this resource.

- **:docs1.5:`Topology Example <workspace/topologies/bkr-new.yml>`**

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

- **:code1.5:`bkr_server.py <linchpin/provision/library/bkr_server.py>`**
- **:code1.5:`bkr_info.py <linchpin/provision/library/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	
ssh_keys_path	false	string	ssh_keys_path	Credentials directory
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		
ks_meta	false	string	N/A		
kernel_options	false	string	N/A		
kernel_options_post	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		
ks_append	false	list	list of strings		
ssh_key	false	list	list of strings		
ssh_key_file	false	list	list of file names		
kickstart	false	string	absolute path to a kickstart template		

continues on next page

Table 1 – continued from previous page

Parameter	required	type	sub-field layout options
partitions	false	list	**param** | **required** | **type**
			name | true | string size | true | integer fs | false | string type | false | string

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.

- **:docs1.5:`Topology Example <workspace/topologies/duffy-new.yml>`**

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.

- **:docs1.5:`Topology Example <workspace/topologies/ovirt-new.yml>`**
- 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
```

Docker

The docker provider manages `docker_container` and `docker_image` resources.

- **:docs1.5:`Topology Example <workspaces/docker/topologies/docker-new.yml>`**

docker_container

The `docker_container` resource provides the ability to provision a Docker container. It is implemented as a wrapper around the Ansible's *docker_container <https://docs.ansible.com/ansible/latest/modules/docker_container_module.html>* module so that same requirements, parameters, and behavior are expected.

Topology Schema

Within Linchpin, the docker_container resource_definition has more options than what are shown in the examples above. For each docker_container definition, the same options of the Ansible *docker_container* module are available. The :term: name :term: option is required.

See the *docker_container parameters <https://docs.ansible.com/ansible/latest/modules/docker_container_module.html#parameters>* for the complete list and defaults.

docker_image

The `docker_image` resource provides the ability to manage a Docker image. It is implemented as a wrapper around the Ansible's *docker_image <https://docs.ansible.com/ansible/latest/modules/docker_image_module.html>* module so that same requirements, parameters, and behavior are expected.

Topology Schema

Within Linchpin, the docker_image resource_definition has more options than what are shown in the examples above. For each docker_image definition, the same options of the Ansible *docker_image* module are available. The :term: name :term: option is required.

See the *docker_image parameters <https://docs.ansible.com/ansible/latest/modules/docker_image_module.html#parameters>* for the complete list and defaults.

Note: The provider assume that the `cacert_path`, `cert_path`, `path`, and `load_path` parameter value are relative to the workspace path, unless its value is absolute (e.g. /path/to/cert) or relative (e.g. ./path/to/cert) to the OS filesystem.

Additional Dependencies

The docker resource group requires the same dependencies of the Ansible docker_container module. See the *docker_container requirements <https://docs.ansible.com/ansible/latest/modules/docker_container_module.html#requirements>* documentation for the complete list of dependencies and any further detail.

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. * **:code1.5:`openshift.py <linchpin/provision/library/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. * **:docs1.5:`Topology Example <workspace/topologies/openshift-new.yml>`**

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
        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

```
oc login
```

After login run following command to get the API_ENDPOINT:

```
oc version | grep Server | awk '{print $2}'
```

Run the following command to get API_TOKEN

```
oc whomai -t
```

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 the name of the target and the uhash. However, the value can be overridden simply by adding the `inventory_file` option. The uhash can be disabled for all targets by setting the `enable_uhash` flag to False in `linchpin.conf` or for a subset of targets by using the `--disable-uhash` flag when running `linchpin up` and providing a comma-separated list of targets

```
---
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"
                }
```

(continues on next page)

(continued from previous page)

```
            },
            "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",
```

(continues on next page)

(continued from previous page)

```
                                "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": [
                    {
                        "actions": [
                            "echo hello"
                        ],
                        "name": "hello",
                        "type": "shell"
                    }
                ]
            }
        }
    ],
    "outputs": [
        {
            "resources": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "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
* 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"
                        ]
                    }
                },
                "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"
                }
            ]
        }
    }
],
```

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"
                ]
            }
        ]
    }
],
```

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,
```

(continues on next page)

(continued from previous page)


```
        "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
↪",
                "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,
```


(continues on next page)

(continued from previous page)


```
                    "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.

RunDB Drivers

Custom database drivers can be added to LinchPin. LinchPin requires that these drivers contain certain functions in order to interface with the existing LinchPin code.

Existing Drivers

Currently, LinchPin supports two drivers.

TinyDB

This is the default database driver for LinchPin. It has no exernal dependencies but cannot support reading and writing from multiple linchpin processes at the same time. If you need this functionality, you should use another driver.

MongoDB

This driver has the advantage of concurrency, but also requires a daemon in order to run.

Adding Custom Drivers

All database drivers for LinchPin must extend the *linchpin.rundb.BaseDB* class and contain the following functions:

```
@schema.setter
def schema(self, schema)
```

Sets the *schema* property for the class. If your database requires a schema (such as MySQL), this is where you should set it.

```
def init_table(self, table)
```

Sets up the table for the current run of LinchPin. Returns a run_id for the next run. *run_id* is a variable used to identify a document in the RunDB. It begins at 1 and increments from there.

```
def update_record(self, table, run_id, key, value)
```

updates a single record in the database. Note that the "outputs" record is a list containing two items: a dict in the format of *{ "resources": [] }* and another one in the format of *{ "inventory_path": [] }*. If a resources dict is passed to *update_record()*, the array needs to be appended to the existing resources array. If the driver supports concurrent transactions, care must be taken to avoid race conditions.

```
def get_tx_record(self, tx_id)
```

Retrieves a single transaction record for the rundb. A transaction record contains a list of the targets provisioned, their uhashes and their return codes. It does not contain the topology, layouts, or outputs from the cloud.

```
def get_tx_records(self, tx_ids)
```

Gets multiple records corresponding with a list of transaction ids.

```
def get_run_id(self, table, action='up')
```

Returns the id corresponding with the most recent instance of the given action.

```
def get_record(self, table, action=None, run_id=None)
```

Returns a single record. If a run_id is supplied, the record corresponding with the given run_id will be returned. Else if an action is supplied, the most recent record corresponding with that action is supplied.

```
def get_records(self, table, count=10)
```

Returns the *count* most recent records.

```
def get_tables(self)
```

Returns a list of tables.

```
def remove_record(self, table, key)
```

Removes a record from the rundb

```
def purge(self, table)
```

Deletes a single database

In addition, the functions that use the database use the @usedb decorator, which opens the database, performs the operation, and closes it again

```
def usedb(func):
    def func_wrapper(*args, **kwargs):
        args[0]._opendb()
        x = func(*args, **kwargs)
        args[0]._closedb()
        return x
    return func_wrapper
```

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 **:dirs1.5:`linchpin.conf <workspace/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 **:code1.5:`linchpin.constants <linchpin/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": [
        {
```

(continues on next page)

(continued from previous page)


```
            "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"
        }
    ]
}
```


PinFile Configs

You can use the cfgs section of the PinFile to define variables for use in inventories. These variables map to values in the json returned by the relevant provider, and are dot-separated. For example, the variable __IP__ in the cfgs below would map to the address 55.234.16.11 in the following json:


```
{
   'addresses': [
      {
         'public_v4': '55.234.16.11'
      },
      {
         'public_v4': '219.16.122.93'
      }
   ]
}
```


```
cfgs:
  aws:
    __IP__: addresses.0.public_v4
```

Information on the json returned by different providers can be found below:

AWS Sample Output


```
{
   "kernel": null,
   "root_device_type": "ebs",
   "private_dns_name": "",
   "public_ip": "",
   "private_ip": "",
   "id": "i-01cc0455abe8465b8",
   "ebs_optimized": false,
   "state": "running",
   "virtualization_type": "hvm",
```


(continues on next page)

(continued from previous page)


```
    "root_device_name": "/dev/sda1",
    "ramdisk": null,
    "block_device_mapping": {
      "/dev/sdb": {
        "status": "attached",
        "delete_on_termination": true,
        "volume_id": "vol-0f3311851115c8241"
      },
      "/dev/sda1": {
        "status": "attached",
        "delete_on_termination": true,
        "volume_id": "vol-00f6f149c57ac152c"
      }
    },
    "key_name": null,
    "image_id": "ami-984189e2",
    "tenancy": "default",
    "groups": {
      "sg-eae64983": "default",
      "sg-8a1d78e3": "public"
    },
    "public_dns_name": "",
    "state_code": 16,
    "tags": {
      "color": "blue",
      "resource_group_name": "aws",
      "shape": "oval",
      "name": "demo-day"
    },
    "placement": "us-east-1c",
    "ami_launch_index": "0",
    "dns_name": "",
    "region": "us-east-1",
    "launch_time": "2018-10-01T17:19:23.000Z",
    "instance_type": "m1.small",
    "architecture": "x86_64",
    "hypervisor": "xen"
}
```


Dummy Sample Output


```
{
    "hypervisor": "xen"
    "failed": false,
    "changed": true,
    "hosts": [
       dummy-8c8b6b-0,
       dummy-8c8b6b-1,
       dummy-8c8b6b-2,
  ],
    "resource_type": "dummy_res",
    "dummy_file": "/tmp/dummy.hosts"
}
```

Libvirt Sample Output


```
{
    "ip": "192.168.122.119",
    "name": "centos71-872d6a_0"
}
```


openstack sample output


```
{
    "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": "2018-09-19T14:53:12.000000",
    "OS-SRV-USG:terminated_at": null,
    "accessIPv4": "",
    "accessIPv6": "",
    "addresses": {
        "e2e-openstack": [
            {
                "OS-EXT-IPS-MAC:mac_addr": "fa:16:3e:a1:c0:6b",
                "OS-EXT-IPS:type": "fixed",
                "addr": "",
                "version": 4
            }
        ]
    },
    "adminPass": "",
    "az": "nova",
    "cloud": "defaults",
    "config_drive": "",
    "created": "2018-09-19T14:46:51Z",
    "created_at": "2018-09-19T14:46:51Z",
    "disk_config": "MANUAL",
    "flavor": {
        "id": "2",
        "name": "m1.small"
    },
    "has_config_drive": false,
    "hostId": "190ddf5e439d5fa9a5e767485c44e8fdbfa206166eaf5aa6ed100fc0",
    "host_id": "190ddf5e439d5fa9a5e767485c44e8fdbfa206166eaf5aa6ed100fc0",
    "id": "83e2d9d3-7823-45f3-8a58-52452acddaa8",
    "image": {
        "id": "11b72b11-59e8-4919-a918-265c1566bd45",
        "name": "CentOS-7-x86_64-GenericCloud-1612"
    },
    "interface_ip": "",
    "key_name": "ci-factory",
    "launched_at": "2018-09-19T14:53:12.000000",
    "location": {
        "cloud": "defaults",
        "project": {
            "domain_id": null,
```


(continues on next page)

(continued from previous page)


```
            "domain_name": null,
            "id": "f53391f4d50643f283af5d59fc450e09",
            "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
        },
        "region_name": "",
        "zone": "nova"
    },
    "metadata": {},
    "name": "596-master-d7b60a-1",
    "networks": {},
    "os-extended-volumes:volumes_attached": [],
    "power_state": 1,
    "private_v4": "",
    "progress": 0,
    "project_id": "f53391f4d50643f283af5d59fc450e09",
    "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": "2018-09-19T14:53:12.000000",
        "OS-SRV-USG:terminated_at": null,
        "os-extended-volumes:volumes_attached": []
    },
    "public_v4": "",
    "public_v6": "",
    "region": "",
    "security_groups": [
        {
            "description": "Default security group",
            "id": "f48c6b12-497b-4301-97f5-0c8749815089",
            "location": {
                "cloud": "defaults",
                "project": {
                    "domain_id": null,
                    "domain_name": null,
                    "id": "f53391f4d50643f283af5d59fc450e09",
                    "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
                },
                "region_name": "",
                "zone": null
            },
            "name": "default",
            "project_id": "f53391f4d50643f283af5d59fc450e09",
            "properties": {},
            "security_group_rules": [
                {
                    "direction": "ingress",
                    "ethertype": "IPv4",
                    "group": {},
                    "id": "1b315474-5730-483e-a9b7-712530c17b19",
                    "location": {
                        "cloud": "defaults",
                        "project": {
                            "domain_id": null,
                            "domain_name": null,
```


(continues on next page)

(continued from previous page)


```
                        "id": "f53391f4d50643f283af5d59fc450e09",
                        "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
                    },
                    "region_name": "",
                    "zone": null
                },
                "port_range_max": 22,
                "port_range_min": 22,
                "project_id": "",
                "properties": {
                    "group": {}
                },
                "protocol": "tcp",
                "remote_group_id": null,
                "remote_ip_prefix": "0.0.0.0/0",
                "security_group_id": "f48c6b12-497b-4301-97f5-0c8749815089",
                "tenant_id": ""
            },
            {
                "direction": "ingress",
                "ethertype": "IPv4",
                "group": {
                    "name": "default",
                    "tenant_id": "f53391f4d50643f283af5d59fc450e09"
                },
                "id": "2e45cfff-370d-460f-a88f-f3042b4a25d8",
                "location": {
                    "cloud": "defaults",
                    "project": {
                        "domain_id": null,
                        "domain_name": null,
                        "id": "f53391f4d50643f283af5d59fc450e09",
                        "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
                    },
                    "region_name": "",
                    "zone": null
                },
                "port_range_max": null,
                "port_range_min": null,
                "project_id": "",
                "properties": {
                    "group": {
                        "name": "default",
                        "tenant_id": "f53391f4d50643f283af5d59fc450e09"
                    }
                },
                "protocol": null,
                "remote_group_id": null,
                "remote_ip_prefix": null,
                "security_group_id": "f48c6b12-497b-4301-97f5-0c8749815089",
                "tenant_id": ""
            },
            {
                "direction": "ingress",
                "ethertype": "IPv4",
                "group": {},
                "id": "33078914-a857-45c4-8ed2-d4ba9d7b41be",
```


(continues on next page)

(continued from previous page)


```
                    "location": {
                        "cloud": "defaults",
                        "project": {
                            "domain_id": null,
                            "domain_name": null,
                            "id": "f53391f4d50643f283af5d59fc450e09",
                            "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
                        },
                        "region_name": "",
                        "zone": null
                    },
                    "port_range_max": null,
                    "port_range_min": null,
                    "project_id": "",
                    "properties": {
                        "group": {}
                    },
                    "protocol": "icmp",
                    "remote_group_id": null,
                    "remote_ip_prefix": "0.0.0.0/0",
                    "security_group_id": "f48c6b12-497b-4301-97f5-0c8749815089",
                    "tenant_id": ""
                },
                {
                    "direction": "ingress",
                    "ethertype": "IPv4",
                    "group": {
                        "name": "default",
                        "tenant_id": "f53391f4d50643f283af5d59fc450e09"
                    },
                    "id": "b801bf97-f470-476b-9d63-b692de45ec67",
                    "location": {
                        "cloud": "defaults",
                        "project": {
                            "domain_id": null,
                            "domain_name": null,
                            "id": "f53391f4d50643f283af5d59fc450e09",
                            "name": "VALUE_SPECIFIED_IN_NO_LOG_PARAMETER"
                        },
                        "region_name": "",
                        "zone": null
                    },
                    "port_range_max": null,
                    "port_range_min": null,
                    "project_id": "",
                    "properties": {
                        "group": {
                            "name": "default",
                            "tenant_id": "f53391f4d50643f283af5d59fc450e09"
                        }
                    },
                    "protocol": null,
                    "remote_group_id": null,
                    "remote_ip_prefix": null,
                    "security_group_id": "f48c6b12-497b-4301-97f5-0c8749815089",
                    "tenant_id": ""
                }
```


(continues on next page)

(continued from previous page)


```
            ],
            "tenant_id": "f53391f4d50643f283af5d59fc450e09"
        }
    ],
    "status": "ACTIVE",
    "task_state": null,
    "tenant_id": "f53391f4d50643f283af5d59fc450e09",
    "terminated_at": null,
    "updated": "2018-09-19T14:53:12Z",
    "user_id": "e32798f55da74cffa90d629e50939582",
    "vm_state": "active",
    "volumes": []
}
```


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

property hook_state
getter function for hook_state property of the API object

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

prepare_rundb(*target*, *action*, *run_id=None*, *tx_id=None*)

run_hooks(*state*, *action*)

run_target(*target*, *resources*, *action*, *run_id=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_pbar()

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

ssh(*target*)

update_rundb(*rundb_id*, *target*, *provision_data*)

write_results_to_rundb(*results*, *action*)

linchpin.**progress_monitor**(*disable_pbar*, *target*)

linchpin.**tqdm_or_mock**(*disable*, **args*, ***kwargs*)

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_env_vars(*key=None*, *default=None*)
Get the current env_vars

Parameters

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

(default: None)

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(*workspace=None*, *config_path=None*, *search_path=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)

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_env_vars(*key*, *value*)
Set a value into env_vars. Does not persist into a file, only during the current execution.

Parameters

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

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

class linchpin.ansible_runner.**Options**(*connection*, *module_path*, *forks*, *become*, *become_method*, *become_user*, *listhosts*, *listtasks*, *listtags*, *syntax*, *remote_user*, *private_key_file*, *ssh_common_args*, *ssh_extra_args*, *sftp_extra_args*, *scp_extra_args*, *start_at_task*, *verbosity*, *check*, *diff*, *vault_password_files*)

linchpin.ansible_runner.**ansible_runner**(*playbook_path*, *module_path*, *extra_vars*, *vault_password_file*, *inventory_src='localhost'*, *verbosity=2*, *console=True*, *env_vars=()*, *use_shell=False*)
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*, *inventory_src='localhost'*, *console=True*)

linchpin.ansible_runner.**ansible_runner_28x**(*playbook_path*, *extra_vars*, *options*, *inventory_src='localhost'*, *console=True*)

linchpin.ansible_runner.**ansible_runner_shell**(*playbook_path*, *module_path*, *extra_vars*, *vault_password_file=None*, *inventory_src='localhost'*, *verbosity=1*, *console=True*, *env_vars=()*, *check=False*)

linchpin.ansible_runner.**set_environment_vars**(*env_vars*)
Sets environment variables passed : param env_vars: list of tuples

linchpin.ansible_runner.**subprocess_runner**(*cmd*, *shell=False*)
Runs subprocess commands param: cmd in a list param: shell to print stdout, stderr or not

linchpin.ansible_runner.**suppress_stdout**()
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

class linchpin.api.**Pinfile**(*pinfile={}*, *config='linchpin.conf'*, *workspace_path=None*)

destroy()
Destroys pinfile resources constructed through the Pinfile object

returns output dictionary

up()
provsions pinfile resources constructed through the Pinfile object

returns output dictionary

class linchpin.api.**Workspace**(*path=None*)

destroy()
Destroys workspace resources constructed through the workspace object

returns output dictionary

find_pinfile()
find_pinfile function to search pinfiles in workspace path returns pinfile path if found

get_cfg(*section*, *key*)
get_cfg gets current linchpin.conf values based on section, key

returns string

get_credentials_path()
get_credentials_path function gets current credentials path

returns path to credential file

get_evar(*key*)
get_evar function sets extra vars in current run

Parameters **key** – string

returns value for corresponding key

get_flag_ignore_failed_hooks()
get_flag_ignore_failed_hooks get current ignore_failed_hooks flag value

returns boolean

get_flag_no_hooks()
get_flag_no_hooks gets current vault_encryption flag value

returns boolean

get_inventory(*inv_format='json'*)
get_inventory gets inventory of latest run

param: inv_format: string json/ini

returns dict/string

get_latest_run()
get_latest_run get latest resources provisioned

returns dict

get_vault_encryption()
get_vault_encryption gets current vault_encryption flag value

returns boolean

get_vault_pass()
get_valut_pass get current valut_password set

returns boolean

get_workspace()
get_workspace function gets current workspace path

Parameters **path** – path to workspace directory

returns workspace path if set

load_data(*path*)
load_data function to load from workspace path

Parameters **path** – path to workspace directory

set_cfg(*section*, *key*, *value*)
get_flag_ignore_failed_hooks get current ignore_failed_hooks flag value

returns boolean

set_credentials_path(*creds_path*)
set_credentials_path function set credentials path

Parameters **creds_path** – path to credential directory

returns True/False

set_evar(*key*, *value*)
set_evar function sets extra vars in current run

Parameters

- **key** – string
- **value** – string

returns key,value tuple

set_flag_ignore_failed_hooks(*flag*)
set_flag_ignore_failed_hooks sets current ignore_failed_hooks flag value

param: flag: boolean

set_flag_no_hooks(*flag*)
set_flag_no_hooks sets no_hooks flag

param: flag: boolean

returns boolean

set_vault_encryption(*vault_enc*)
set_vault_encryption sets vault_encryption flag if credentials are encrypted in vault current credentials path

param: vault_enc: boolean

returns boolean

set_vault_pass(*vault_pass*)
set_vault_pass set current vault_pass value

param: vault_pass: string returns boolean

set_workspace(*path*)
set_workspace function sets workspace path

Parameters **path** – path to workspace directory

returns workspace path if set

up()
provisions workspace resources constructed through the workspace object

returns output dictionary

validate()
validate function to validate loaded workspace/pinfile

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*, *env_vars=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_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: topologies, layouts, 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'*, *env_vars=()*)
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

property pf_data
getter for pinfile template data

property pinfile
getter function for pinfile name

property 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

property inventory
getter function for inventory

property inventory_folder
getter function for inventory_folder

property inventory_path
getter function for inventory_path

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

property pinfile
getter function for pinfile name

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

property 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 *Linchpin Hooks* functionality within LinchPin.

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

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

execute_hook(*block_obj*, *target*)

fetch_git_src(*block*)

fetch_src(*block*)

get_custom_action_manager(*action_block*)

global_hooks_block(*block*)

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()

resolve_block_path(*block*)

run_action(*state*, *block*, *tgt_data*)

run_actions(*state*, *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*)

run_local_actions(*state*, *action_blocks*, *tgt_data*)

property 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()

1.3.2 Developing LinchPin

This guide will walk you through the basics of contributing to LinchPin.

Topics

Checking out the linchpin code

You can check out the linchpin code by cloning the git repository from github.

```
$ git clone https://github.com/CentOS-PaaS-SIG/linchpin.git
```

But to submit pull requests (PR's) you will need to fork the project on github webui first. Then you can add a remote for that fork. This is where you will push your changes.

```
$ git remote add myfork git@github.com:<YOUR_GITHUB_USERNAME>/linchpin.git
```

Remember to replace <YOUR_GITHUB_USERNAME> with your actual github login.

Working on a feature or bug

All new work happens off the develop branch. It is good practice to make sure you have the latest version before starting work on your changes.

```
$ git checkout develop
$ git pull
```

Now that you are in the develop branch and have the latest version you can create a new branch to use for your changes.

```
$ git checkout -b <DESCRIPTIVE_BRANCH_NAME>
```

Replace <DESCRIPTIVE_BRANCH_NAME> with a branch name that makes sense. This name will show up in your github fork branches.

Creating a Pull Request

After you have committed your changes and tested them locally you can push them to your github fork repo.

```
$ git pull --rebase origin develop
$ git push myfork <DESCRIPTIVE_BRANCH_NAME>:<DESCRIPTIVE_BRANCH_NAME>
Enumerating objects: 1014, done.
Counting objects: 100% (768/768), done.
Delta compression using up to 8 threads
Compressing objects: 100% (290/290), done.
Writing objects: 100% (634/634), 83.86 KiB | 6.99 MiB/s, done.
Total 634 (delta 462), reused 436 (delta 324)
remote: Resolving deltas: 100% (462/462), completed with 84 local objects.
remote:
remote: Create a pull request for 'devel_docs' on GitHub by visiting:
remote:      https://github.com/<YOUR_GITHUB_USERNAME>/linchpin/pull/new/devel_docs
remote:
To github.com:<YOUR_GITHUB_USERNAME>/linchpin.git
 * [new branch]        devel_docs -> devel_docs
```

The remote output explains how you can create a pull request by following the url referenced. Again, <YOUR_GITHUB_USERNAME> will match your github username.

Once a pull request has been created the automated testing will kick off automatically. There is upstream testing which is run on publicly accessable servers and there is downstream testing which is run inside Red Hat. We try to do most testing upstream since this in an open source project, but some of the providers are only available inside Red Hat.

The upstream testing is referenced from the All checks section. Downstream testing is recorded as a comment.

If for some reason you need to kick off the testing again you can add a comment with the keyword [test] in it. It has to be inside the square brackets in order to trigger.

Depending on your contribution status your PR may not kick off automated testing and will require someone from the project to initiate the testing.

You can request reviewers at this point and depending on the files that have been changed github may suggest some reviewers based on who last changed that code.

Updating a Pull Request

If changes are required for your PR then please amend to your commit and force push. If other commits have been merged into develop since you started your PR you may need to rebase your PR on the latest code. One reason for this is if changes to the automated testing infrastructure have been made.

```
$ git add -u
$ git commit --amend
$ git pull --rebase origin develop
$ git push myfork --force <DESCRIPTIVE_BRANCH_NAME>:<DESCRIPTIVE_BRANCH_NAME>
```

Merging a Pull Request

When all the tests are passing and the code has been approved by the reviewers you can merge the PR. Don't use the merge button on github. There is a workflow that does the merge which is triggered by the comment [merge] in it. Again, it has to be inside the square brackets in order to trigger.

The reason for this is we have containers used in the testing process which may need to be updated depending on the code that is changed. Our workflow will promote those containers and do the merge on github.

Depending on your contribution status you may not have permission to do a merge. In that case you can leave a comment saying the PR is ready for merging.

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 **:docs1.5:`linchpin.conf <workspace/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

TWO

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

T

U

V

W