Drydock Documentation

Release 0.1.0

Drydock Authors

Jul 11, 2019

Contents

Drydock is a python REST orchestrator to translate a YAML host topology to a provisioned set of hosts and provide a set of cloud-init post-provisioning instructions.

CHAPTER 1

User's Guide

1.1 Drydock Configuration Guide

1.1.1 Installing Drydock in a Dev Environment

Bootstrap Kubernetes

You can bootstrap your Helm-enabled Kubernetes cluster via the Openstack-Helm AIO or the Promenade tools.

Deploy Drydock and Dependencies

Drydock is most easily deployed using Armada to deploy the Drydock container into a Kubernetes cluster via Helm charts. The Drydock chart is in the `charts/drydock` directory. It depends on the deployments of the MaaS chart and the Keystone chart.

A integrated deployment of these charts can be accomplished using the Armada tool. An example integration chart can be found in the Airship in a Bottle repo in the `./manifests/dev_single_node` directory.

Load Site

To use Drydock for site configuration, you must craft and load a site topology YAML. An example of this is in `./test/yaml_samples/deckhand_fullsite.yaml`.

Documentation on building your topology document is at *Authoring Site Topology*.

Drydock requires that the YAML topology be hosted somewhere, either the preferred method of using Deckhand or through a simple HTTP server like Nginx or Apache.

Use the CLI to create tasks to deploy your site

```
# drydock task create -d <design_url> -a verify_site
# drydock task create -d <design_url> -a prepare_site
# drydock task create -d <design_url> -a prepare_nodes
# drydock task create -d <design_url> -a deploy_nodes
```

A demo of this process is available at https://asciinema.org/a/133906

1.1.2 Configuring Drydock

Drydock uses an INI-like standard oslo_config file. A sample file can be generated via tox:

```
$ tox -e genconfig
```

Customize your configuration based on the information below

Keystone Integration

Drydock requires a service account to use for validating client tokens:

```
$ openstack domain create 'ucp'
$ openstack project create --domain 'ucp' 'service'
$ openstack user create --domain ucp --project service --project-domain 'ucp' --
↪password drydock drydock
$ openstack role add --project-domain ucp --user-domain ucp --user drydock --project␣
↪service admin
```

The service account must then be included in the drydock.conf:

```
[keystone_authtoken]
auth_uri = http://<keystone_ip>:5000/v3
auth_version = 3
delay_auth_decision = true
auth_type = password
auth_section = keystone_authtoken_password
auth_url = http://<keystone_ip>:5000
project_name = service
project_domain_name = ucp
user_name = drydock
user_domain_name = ucp
password = drydock
```

MaaS Integration

Drydock uses Canonical MaaS to provision new nodes. This requires a running MaaS instance and providing Drydock with the address and credentials. The MaaS API enforces authentication via a API key generated by MaaS and used to sign API calls. Configure Drydock with the MaaS API URL and a valid API key.:

```
[maasdriver]
maas_api_url = http://<maas_ip>:<maas_port>/MAAS
maas_api_key = <valid API key>
```

1.1.3 Sample Configuration File

The following is a sample Drydock configuration for adaptation and use. It is auto-generated from Drydock when this documentation is built, so if you are having issues with an option, please compare your version of Drydock with the version of this documentation.

The sample configuration can also be viewed in file form.

```
[DEFAULT]

#
# From drydock_provisioner
#

# Polling interval in seconds for checking subtask or downstream status (integer
# value)
# Minimum value: 1
#poll_interval = 10

# How long a leader has to check-in before leadership can be usurped, in seconds
# (integer value)
#leader_grace_period = 300

# How often will an instance attempt to claim leadership, in seconds (integer
# value)
#leadership_claim_interval = 30


[database]

#
# From drydock_provisioner
#

# The URI database connect string. (string value)
#database_connect_string = <None>

# The SQLalchemy database connection pool size. (integer value)
#pool_size = 15

# Should DB connections be validated prior to use. (boolean value)
#pool_pre_ping = true

# How long a request for a connection should wait before one becomes available.
# (integer value)
#pool_timeout = 30

# How many connections above pool_size are allowed to be open during high usage.
# (integer value)
#pool_overflow = 10

# Time, in seconds, when a connection should be closed and re-established. -1
# for no recycling. (integer value)
#connection_recycle = -1


[keystone_authtoken]
```

(continues on next page)

(continued from previous page)

```
#
# From drydock_provisioner
#

# Authentication URL (string value)
#auth_url = <None>

# Domain ID to scope to (string value)
#domain_id = <None>

# Domain name to scope to (string value)
#domain_name = <None>

# Project ID to scope to (string value)
# Deprecated group/name - [keystone_authtoken]/tenant_id
#project_id = <None>

# Project name to scope to (string value)
# Deprecated group/name - [keystone_authtoken]/tenant_name
#project_name = <None>

# Domain ID containing project (string value)
#project_domain_id = <None>

# Domain name containing project (string value)
#project_domain_name = <None>

# Trust ID (string value)
#trust_id = <None>

# Optional domain ID to use with v3 and v2 parameters. It will be used for both
# the user and project domain in v3 and ignored in v2 authentication. (string
# value)
#default_domain_id = <None>

# Optional domain name to use with v3 API and v2 parameters. It will be used for
# both the user and project domain in v3 and ignored in v2 authentication.
# (string value)
#default_domain_name = <None>

# User id (string value)
#user_id = <None>

# Username (string value)
# Deprecated group/name - [keystone_authtoken]/user_name
#username = <None>

# User's domain id (string value)
#user_domain_id = <None>

# User's domain name (string value)
#user_domain_name = <None>

# User's password (string value)
#password = <None>

#
```

(continues on next page)

(continued from previous page)

```
# From keystonemiddleware.auth_token
#

# Complete "public" Identity API endpoint. This endpoint should not be an
# "admin" endpoint, as it should be accessible by all end users. Unauthenticated
# clients are redirected to this endpoint to authenticate. Although this
# endpoint should  ideally be unversioned, client support in the wild varies.
# If you're using a versioned v2 endpoint here, then this  should *not* be the
# same endpoint the service user utilizes  for validating tokens, because normal
# end users may not be  able to reach that endpoint. (string value)
#auth_uri = <None>

# API version of the admin Identity API endpoint. (string value)
#auth_version = <None>

# Do not handle authorization requests within the middleware, but delegate the
# authorization decision to downstream WSGI components. (boolean value)
#delay_auth_decision = false

# Request timeout value for communicating with Identity API server. (integer
# value)
#http_connect_timeout = <None>

# How many times are we trying to reconnect when communicating with Identity API
# Server. (integer value)
#http_request_max_retries = 3

# Request environment key where the Swift cache object is stored. When
# auth_token middleware is deployed with a Swift cache, use this option to have
# the middleware share a caching backend with swift. Otherwise, use the
# ``memcached_servers`` option instead. (string value)
#cache = <None>

# Required if identity server requires client certificate (string value)
#certfile = <None>

# Required if identity server requires client certificate (string value)
#keyfile = <None>

# A PEM encoded Certificate Authority to use when verifying HTTPs connections.
# Defaults to system CAs. (string value)
#cafile = <None>

# Verify HTTPS connections. (boolean value)
#insecure = false

# The region in which the identity server can be found. (string value)
#region_name = <None>

# Directory used to cache files related to PKI tokens. (string value)
#signing_dir = <None>

# Optionally specify a list of memcached server(s) to use for caching. If left
# undefined, tokens will instead be cached in-process. (list value)
# Deprecated group/name - [keystone_authtoken]/memcache_servers
#memcached_servers = <None>
```

(continues on next page)

(continued from previous page)

```
# In order to prevent excessive effort spent validating tokens, the middleware
# caches previously-seen tokens for a configurable duration (in seconds). Set to
# -1 to disable caching completely. (integer value)
#token_cache_time = 300

# Determines the frequency at which the list of revoked tokens is retrieved from
# the Identity service (in seconds). A high number of revocation events combined
# with a low cache duration may significantly reduce performance. Only valid for
# PKI tokens. (integer value)
#revocation_cache_time = 10

# (Optional) If defined, indicate whether token data should be authenticated or
# authenticated and encrypted. If MAC, token data is authenticated (with HMAC)
# in the cache. If ENCRYPT, token data is encrypted and authenticated in the
# cache. If the value is not one of these options or empty, auth_token will
# raise an exception on initialization. (string value)
# Possible values:
# None - <No description provided>
# MAC - <No description provided>
# ENCRYPT - <No description provided>
#memcache_security_strategy = None

# (Optional, mandatory if memcache_security_strategy is defined) This string is
# used for key derivation. (string value)
#memcache_secret_key = <None>

# (Optional) Number of seconds memcached server is considered dead before it is
# tried again. (integer value)
#memcache_pool_dead_retry = 300

# (Optional) Maximum total number of open connections to every memcached server.
# (integer value)
#memcache_pool_maxsize = 10

# (Optional) Socket timeout in seconds for communicating with a memcached
# server. (integer value)
#memcache_pool_socket_timeout = 3

# (Optional) Number of seconds a connection to memcached is held unused in the
# pool before it is closed. (integer value)
#memcache_pool_unused_timeout = 60

# (Optional) Number of seconds that an operation will wait to get a memcached
# client connection from the pool. (integer value)
#memcache_pool_conn_get_timeout = 10

# (Optional) Use the advanced (eventlet safe) memcached client pool. The
# advanced pool will only work under python 2.x. (boolean value)
#memcache_use_advanced_pool = false

# (Optional) Indicate whether to set the X-Service-Catalog header. If False,
# middleware will not ask for service catalog on token validation and will not
# set the X-Service-Catalog header. (boolean value)
#include_service_catalog = true

# Used to control the use and type of token binding. Can be set to: "disabled"
# to not check token binding. "permissive" (default) to validate binding
```

(continues on next page)

(continued from previous page)

```
# information if the bind type is of a form known to the server and ignore it if
# not. "strict" like "permissive" but if the bind type is unknown the token will
# be rejected. "required" any form of token binding is needed to be allowed.
# Finally the name of a binding method that must be present in tokens. (string
# value)
#enforce_token_bind = permissive

# If true, the revocation list will be checked for cached tokens. This requires
# that PKI tokens are configured on the identity server. (boolean value)
#check_revocations_for_cached = false

# Hash algorithms to use for hashing PKI tokens. This may be a single algorithm
# or multiple. The algorithms are those supported by Python standard
# hashlib.new(). The hashes will be tried in the order given, so put the
# preferred one first for performance. The result of the first hash will be
# stored in the cache. This will typically be set to multiple values only while
# migrating from a less secure algorithm to a more secure one. Once all the old
# tokens are expired this option should be set to a single value for better
# performance. (list value)
#hash_algorithms = md5

# Authentication type to load (string value)
# Deprecated group/name - [keystone_authtoken]/auth_plugin
#auth_type = <None>

# Config Section from which to load plugin specific options (string value)
#auth_section = <None>


[libvirt_driver]

#
# From drydock_provisioner
#

# Polling interval in seconds for querying libvirt status (integer value)
#poll_interval = 10


[logging]

#
# From drydock_provisioner
#

# Global log level for Drydock (string value)
#log_level = INFO

# Logger name for the top-level logger (string value)
#global_logger_name = drydock_provisioner

# Logger name for OOB driver logging (string value)
#oobdriver_logger_name = ${global_logger_name}.oobdriver

# Logger name for Node driver logging (string value)
#nodedriver_logger_name = ${global_logger_name}.nodedriver
```

(continues on next page)

(continued from previous page)

```
# Logger name for Kubernetes driver logging (string value)
#kubernetesdriver_logger_name = ${global_logger_name}.kubernetesdriver

# Logger name for API server logging (string value)
#control_logger_name = ${global_logger_name}.control


[maasdriver]

#
# From drydock_provisioner
#

# The API key for accessing MaaS (string value)
#maas_api_key = <None>

# The URL for accessing MaaS API (string value)
#maas_api_url = <None>

# Polling interval for querying MaaS status in seconds (integer value)
#poll_interval = 10


[network]

#
# From drydock_provisioner
#

# Timeout for initial read of outgoing HTTP calls from Drydock in seconds.
# (integer value)
#http_client_connect_timeout = 16

# Timeout for initial read of outgoing HTTP calls from Drydock in seconds.
# (integer value)
#http_client_read_timeout = 300

# Number of retries for transient errors of outgoing HTTP calls from Drydock.
# (integer value)
#http_client_retries = 3


[oslo_policy]

#
# From oslo.policy
#

# The file that defines policies. (string value)
#policy_file = policy.json

# Default rule. Enforced when a requested rule is not found. (string value)
#policy_default_rule = default

# Directories where policy configuration files are stored. They can be relative
# to any directory in the search path defined by the config_dir option, or
# absolute paths. The file defined by policy_file must exist for these
```

(continues on next page)

(continued from previous page)

```
# directories to be searched.  Missing or empty directories are ignored. (multi
# valued)
#policy_dirs = policy.d


[plugins]

#
# From drydock_provisioner
#

# Module path string of a input ingester to enable (string value)
#ingester = drydock_provisioner.ingester.plugins.yaml.YamlIngester

# List of module path strings of OOB drivers to enable (list value)
#oob_driver = drydock_provisioner.drivers.oob.pyghmi_driver.PyghmiDriver

# Module path string of the Node driver to enable (string value)
#node_driver = drydock_provisioner.drivers.node.maasdriver.driver.MaasNodeDriver

# Module path string of the Kubernetes driver to enable (string value)
#kubernetes_driver = drydock_provisioner.drivers.kubernetes.promenade_driver.driver.
↪PromenadeDriver

# Module path string of the Network driver enable (string value)
#network_driver = <None>


[pyghmi_driver]

#
# From drydock_provisioner
#

# Polling interval in seconds for querying IPMI status (integer value)
#poll_interval = 10


[timeouts]

#
# From drydock_provisioner
#

# Fallback timeout when a specific one is not configured (integer value)
#drydock_timeout = 5

# Timeout in minutes for creating site network templates (integer value)
#create_network_template = 2

# Timeout in minutes for creating user credentials (integer value)
#configure_user_credentials = 2

# Timeout in minutes for initial node identification (integer value)
#identify_node = 10

# Timeout in minutes for node commissioning and hardware configuration (integer
```

(continues on next page)

(continued from previous page)

```
# value)
#configure_hardware = 30

# Timeout in minutes for configuring node networking (integer value)
#apply_node_networking = 5

# Timeout in minutes for configuring node storage (integer value)
#apply_node_storage = 5

# Timeout in minutes for configuring node platform (integer value)
#apply_node_platform = 5

# Timeout in minutes for deploying a node (integer value)
#deploy_node = 45

# Timeout in minutes between deployment completion and the all boot actions
# reporting status (integer value)
#bootaction_final_status = 15

# Timeout in minutes for releasing a node (integer value)
#destroy_node = 30

# Timeout in minutes for relabeling a node (integer value)
#relabel_node = 5
```

1.1.4 Sample Policy File

The following is a sample Drydock policy file for adaptation and use. It is auto-generated from Drydock when this documentation is built, so if you are having issues with an option, please compare your version of Drydock with the version of this documentation.

The sample policy file can also be viewed in file form.

```
# Actions requiring admin authority
#"admin_required": "role:admin or is_admin:1"

# Get task status
# GET  /api/v1.0/tasks
# GET  /api/v1.0/tasks/{task_id}
#"physical_provisioner:read_task": "role:admin"

# Create a task
# POST  /api/v1.0/tasks
#"physical_provisioner:create_task": "role:admin"

# Create validate_design task
# POST  /api/v1.0/tasks
#"physical_provisioner:validate_design": "role:admin"

# Create verify_site task
# POST  /api/v1.0/tasks
#"physical_provisioner:verify_site": "role:admin"

# Create prepare_site task
# POST  /api/v1.0/tasks
```

(continues on next page)

(continued from previous page)

```
#"physical_provisioner:prepare_site": "role:admin"

# Create verify_nodes task
# POST  /api/v1.0/tasks
#"physical_provisioner:verify_nodes": "role:admin"

# Create prepare_nodes task
# POST  /api/v1.0/tasks
#"physical_provisioner:prepare_nodes": "role:admin"

# Create deploy_nodes task
# POST  /api/v1.0/tasks
#"physical_provisioner:deploy_nodes": "role:admin"

# Create destroy_nodes task
# POST  /api/v1.0/tasks
#"physical_provisioner:destroy_nodes": "role:admin"

# Create relabel_nodes task
# POST  /api/v1.0/tasks
#"physical_provisioner:relabel_nodes": "role:admin"

# Read build data for a node
# GET  /api/v1.0/nodes/{nodename}/builddata
#"physical_provisioner:read_build_data": "role:admin"

# Read loaded design data
# GET  /api/v1.0/designs
# GET  /api/v1.0/designs/{design_id}
#"physical_provisioner:read_data": "role:admin"

# Load design data
# POST  /api/v1.0/designs
# POST  /api/v1.0/designs/{design_id}/parts
#"physical_provisioner:ingest_data": "role:admin"

# et health status
# GET  /api/v1.0/health/extended
#"physical_provisioner:health_data": "role:admin"

# Validate site design
# POST  /api/v1.0/validatedesign
#"physical_provisioner:validate_site_design": "role:admin"
```

1.1.5 Exceptions Guide

Drydock Exceptions

API Errors

Bootaction Errors

Client Errors

Design Errors

Driver Errors

Orchestrator Errors

BuildData Errors

1.2 API Documentation

1.2.1 Drydock API

The Drydock API is a RESTful interface used for accessing the services provided by Drydock. All endpoints are located under `/api/<version>/`.

Secured endpoints require Keystone authentication and proper role assignment for authorization

v1.0

tasks API

The Tasks API is used for creating and listing asynchronous tasks to be executed by the Drydock orchestrator. See *Tasks* for details on creating tasks and field information.

nodes API

GET nodes

The Nodes API will provide a report of current nodes as known by the node provisioner and their status with a few hardware details.

GET nodes/hostname/builddata

Get all the build data record for node `hostname`. The response will be a list of objects in the below form.:

```
{
  "node_name": "hostname",
  "generator": "description of how data was generated",
  "collected_date": ios8601 UTC datestamp,
  "task_id": "UUID of task initiating collection",
  "data_format": "MIME-type of data_element",
  "data_element": "Collected data"
}
```

If the query parameter `latest` is passed with a value of `true`, then only the most recently collected data for each `generator` will be included in the response.

nodefilter API

POST nodefilter

The Nodes API will provide a list of node names based on design_ref. This API requires design_ref in the POST body with an optional node_filter to return the node names.

bootdata

The boot data API is used by deploying nodes to load the appropriate boot actions to be instantiated on the node. It uses alternative authentication and is not accessible with Keystone.

GET bootdata/hostname/files

Returns a gzipped tar file containing all the file-type boot action data assets for the node `hostname` with appropriate permissions set in the tar-file.

GET bootdata/hostname/units

Returns a gzipped tar file containing all the unit-type boot action data assets for the node `hostname` with appropriate permissions set in the tar-file.

bootaction API

The boot action API is used by deploying nodes to report status and results of running boot actions. It expects a JSON-formatted body with the top-level entity of an object. The status of the boot action and any detail status messages for it will be added to the DeployNode task that prompted the node deployment the boot action is associated with.

POST bootaction/bootaction-id

Example:

```
{
    "status": "Failure"|"Success",
    "details": [
        {
            "message": "Boot action status message",
            "error": true|false,
            ...
        },
        ...
    ]
}
```

POSTs to this endpoint can be made repeatedly omitting the `status` field and simply adding one or more detail status messages. The `message` and `error` fields are required and the `context`, `context_type` and `ts` fields are reserved. Otherwise the message object in details can be extended with additional fields as needed.

Once a POST containing the `status` field is made to a bootaction-id, that bootaction-id can no longer be updated with status changes nor additional detailed status messages.

Each request made must contain the `X-Bootaction-Key` header with the correct hex key for `bootaction-id`.

validatedesign API

The Validatedesign API is used for validating documents before they will be used by Drydock. See *Validate Design* for more details on validating documents.

1.2.2 Tasks

Tasks are requests for Drydock to perform an action asynchronously. Depending on the action being requested, tasks could take seconds to hours to complete. When a task is created, a identifier is generated and returned. That identifier can be used to poll the task API for task status and results.

Task Document Schema

This document can be posted to the Drydock *tasks API* to create a new task.:

```
{
  "action": "validate_design|verify_site|prepare_site|verify_node|prepare_node|deploy_
↪node|destroy_node|relabel_nodes",
  "design_ref": "http_uri|deckhand_uri|file_uri",
  "node_filter": {
    "filter_set_type": "intersection|union",
    "filter_set": [
      {
        "filter_type": "intersection|union",
        "node_names": [],
        "node_tags": [],
        "node_labels": {},
        "rack_names": [],
        "rack_labels": {},
      }
    ]
  }
}
```

The filter is computed by taking the set of all defined nodes. Each filter in the filter set is applied by either finding the union or intersection of filtering the full set of nodes by the attribute values specified. The result set of each filter is then combined as either an intersection or union with that result being the final set the task is executed against.

Assuming you have a node inventory of:

```
[
  {
    "name": "a",
    "labels": {
      "type": "physical",
      "color": "blue"
    }
  },
  {
    "name": "b",
    "labels": {
      "type": "virtual",
```

(continues on next page)

(continued from previous page)

```
      "color": "yellow"
    }
  },
  {
    "name": "c",
    "labels": {
      "type": "physical",
      "color": "yellow"
    }
  }
```

Example:

```
"filter_set": [
  {
    "filter_type": "intersection",
    "node_labels": {
      "color": "yellow",
      "type": "physical"
    }
  },
  {
    "filter_type": "intersection",
    "node_names": ["a"]
  }
],
"filter_set_type": "union"
```

The above filter set results in a set `a` and `c`.

Task Status Schema

When querying the state of an existing task, the below document will be returned:

```
{
  "Kind": "Task",
  "apiVersion": "v1.0",
  "task_id": "uuid",
  "action": "validate_design|verify_site|prepare_site|verify_node|prepare_node|deploy_
↪node|destroy_node|relabel_nodes",
  "design_ref": "http_uri|deckhand_uri|file_uri",
  "parent_task_id": "uuid",
  "subtask_id_list": ["uuid","uuid",...],
  "status": "requested|queued|running|terminating|complete|terminated",
  "node_filter": {
    "filter_set_type": "intersection|union",
    "filter_set": [
      {
        "filter_type": "intersection|union",
        "node_names": [],
        "node_tags": [],
        "node_labels": {},
        "rack_names": [],
        "rack_labels": {},
      }
```

(continues on next page)

(continued from previous page)

```
    ]
  },
  "created": iso8601 UTC timestamp,
  "created_by": "user",
  "updated": iso8601 UTC timestamp,
  "terminated": iso8601 UTC timestamp,
  "terminated_by": "user",
  "result": Status object
}
```

The Status object is based on the Airship standardized response format:

```
{
  "Kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "message": "Drydock Task ...",
  "reason": "Failure reason",
  "status": "failure|success|partial_success|incomplete",
  "details": {
    "errorCount": 0,
    "messageList": [
       StatusMessage
    ]
  }
}
```

The StatusMessage object will change based on the context of the message, but will at a minimum consist of the below:

```
{
  "message": "Textual description",
  "error": true|false,
  "context_type": "site|network|node",
  "context": "site_name|network_name|node_name",
  "ts": iso8601 UTC timestamp,
}
```

Task Build Data

When querying the detail state of an existing task, adding the parameter `builddata=true` in the query string will add one additional field with a list of build data elements collected by this task.:

```
{
  "Kind": "Task",
  "apiVersion": "v1",
  ....
  "build_data": [
    {
      "node_name": "foo",
      "task_id": "uuid",
      "collected_data": iso8601 UTC timestamp,
      "generator": "lshw",
      "data_format": "application/json",
      "data_element": "{ \"id\": \"foo\", \"class\": \"system\" ...}"
```

(continues on next page)

(continued from previous page)

```
    }
  ]
```

Adding the parameter `subtaskerrors=true` in the query string will add one additional field with an object of subtask errors keyed by task_id.

Adding the parameter `layers=x` where x is -1 for all or a positive number to limit the number of layers. Will convert the response into an object of tasks and all subtasks keyed by task_id. It will also include the field init_task_id with the top task_id.

1.2.3 Boot Actions

Boot actions can be more accurately described as post-deployment file placement. This file placement can be leveraged to install actions for servers to take after the permanent OS is installed and the server is rebooted. Including custom or vendor scripts and a SystemD service to run the scripts on first boot or on all boots allows almost any action to be configured.

Boot Action Schema

Boot actions are configured via YAML documents included in the site topology definition. The schema for these YAML documents is described below.

```
data:
  signaling: true
  assets:
    - path: /save/file/here
      location: http://get.data.here/data
      type: unit|file|pkg_list
      data: |
        inline data here
      location_pipeline:
        - template
      data_pipeline
        - base64_decode
        - template
        - base64_encode
      permissions: 555
  node_filter:
    ...
```

`signaling` is a boolean noting whether Drydock should expect a signal at the completion of this boot action. If set to `true` for a boot action that does not send a signal, it will elongate the deployment step and consider the boot action failed.

`assets` is a list of data assets. More details below on how each data asset is rendered.

`node_filter` is an optional filter for selecting to which nodes this boot action will apply. If no node filter is included, all nodes will receive the boot action. Otherwise it will be only the nodes that match the logic of the filter set. See *Tasks* for a definition of the node filter.

Rendering Data Assets

The boot action framework supports assets of several types. `type` can be `unit` or `file` or `pkg_list`.

- `unit` is a SystemD unit, such as a service, that will be saved to `path` and enabled via `systemctl enable [filename]`.
- `file` is simply saved to the filesystem at `path` and set with `permissions`.
- `pkg_list` is a list of packages

Data assets of type `unit` or `file` will be rendered and saved as files on disk and assigned the `permissions` as specified. The rendering process can follow a few different paths.

Referenced vs Inline Data

The asset contents can be sourced from either the in-document `data` field of the asset mapping or dynamically generated by requesting them from a URL provided in `location`. Currently Drydock supports the schemes of `http`, `deckhand+http` and `promenade+http` for referenced data.

Package List

For the `pkg_list` type, the data section is expected to be a YAML mapping with key: value pairs of `package_name`: `version` where `package_name` is a Debian package available in one of the configured repositories and `version` is a valid apt version specifier or a empty/null value. Null indicates no version requirement.

If using a referenced data source for the package list, Drydock expects a YAML or JSON document returned in the above format.

Pipelines

The boot action framework supports pipelines to allow for some dynamic rendering. There are separate pipelines for the `location` field to build the URL that referenced assets should be sourced from and the `data` field (or the data sourced from resolving the `location` field).

The `location` string will be passed through the `location_pipeline` before it is queried. This response or the `data` field will then be passed through the `data_pipeline`. The data entity will start the pipeline as a bytestring meaning if it is defined in the `data` field, it will first be encoded into a bytestring. Below are pipeline segments available for use.

base64_decode Decode the data element from base64

base64_encode Encode the data element in base64

utf8_decode Decode the data element from bytes to UTF-8 string

utf8_encode Encode the data element from a UTF-8 string to bytes

template Treat the data element as a Jinja2 template and apply a node context to it. The defined context available to the template is below.

- node.network.[network_name].ip - IP address of this node on network [network_name]
- node.network.[network_name].cidr - CIDR of [network_name]
- node.network.[network_name].dns_suffix - DNS suffix of [network_name]
- node.hostname - Hostname of the node
- node.domain - DNS Domain of the primary network on the node
- node.tags - Sequence of tags assigned to this node

- node.labels - Key, value pairs of both explicit and dynamic labels for this node
- action.action_id - A ULID that uniquely identifies this boot action on this node. Can be used for signaling boot action result.
- action.action_key - A random key in hex that authenticates API calls for signaling boot action result.
- action.report_url - The URL that can be POSTed to for reporting boot action result.
- action.design_ref - The design reference for the deployment that initiated the bootaction

Also available in the Jinja2 template is the `urlencode` filter to encode a string for inclusion in a URL.

Reporting Results

The assets put in place on a server can report the results of applying the boot action using the Drydock *bootaction API*. The report API URL and boot action key are both available via the `template` pipeline segment context. It is up to the boot action assets to implement the call back to the API for reporting whatever data the boot action desires.

1.2.4 Validate Design

The Drydock Validation API is a set of logic checks that must be passed before any information from the YAMLs will be processed by Drydock. These checks are performed synchronously and will return a message list with a success or failures for each check.

Formatting

This document can be POSTed to the Drydock validatedesign to validate a set of documents that have been processed by Deckhand:

```
{
  rel : "design",
  href: "deckhand+https://{{deckhand_url}}/revisions/{{revision_id}}/rendered-
↪documents",
  type: "application/x-yaml"
}
```

v1.0

Validation Checks

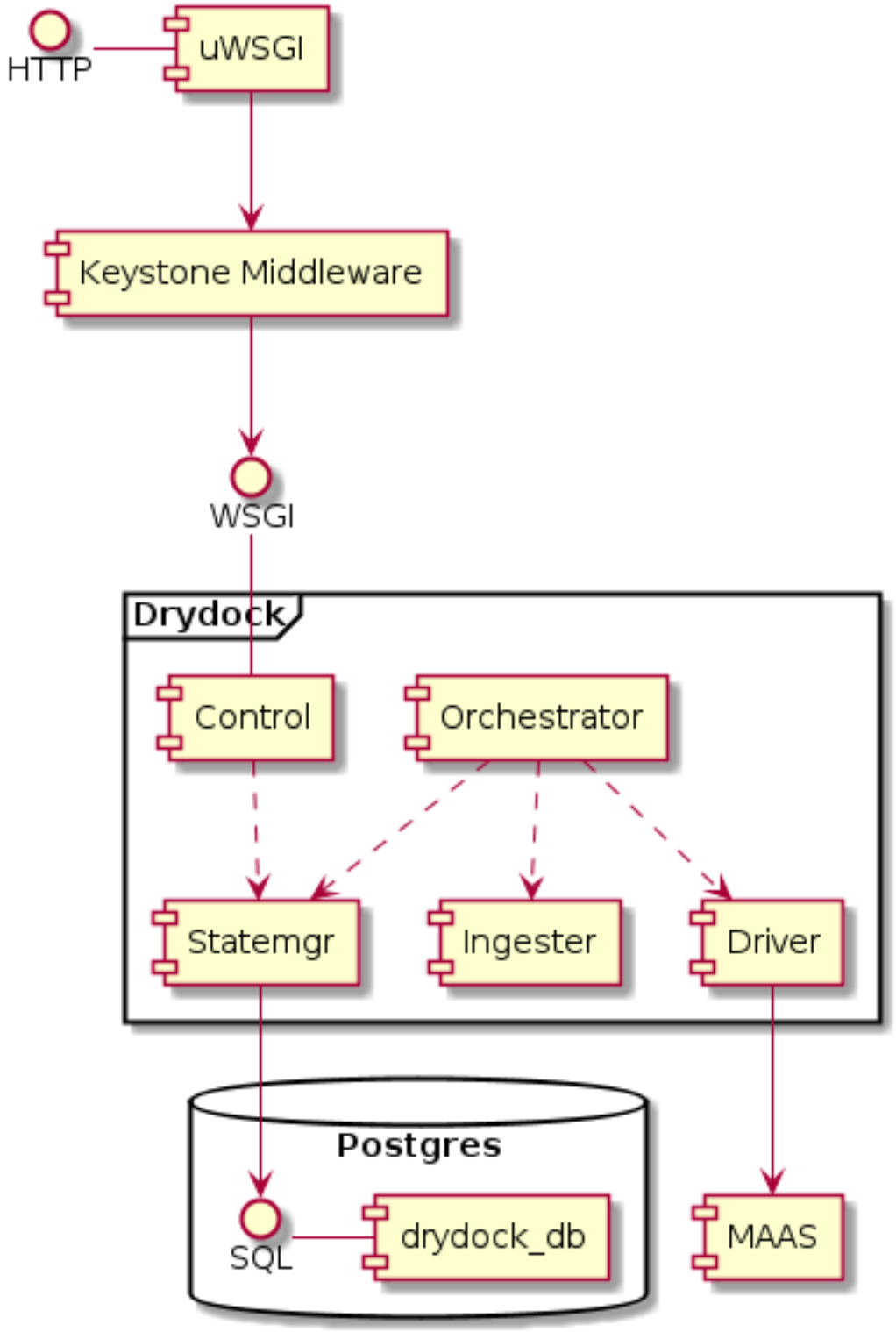
These checks are meant to check the business logic of documents sent to the validatedesign API.

1.3 Developer Overview

1.3.1 Developer Overview of Drydock

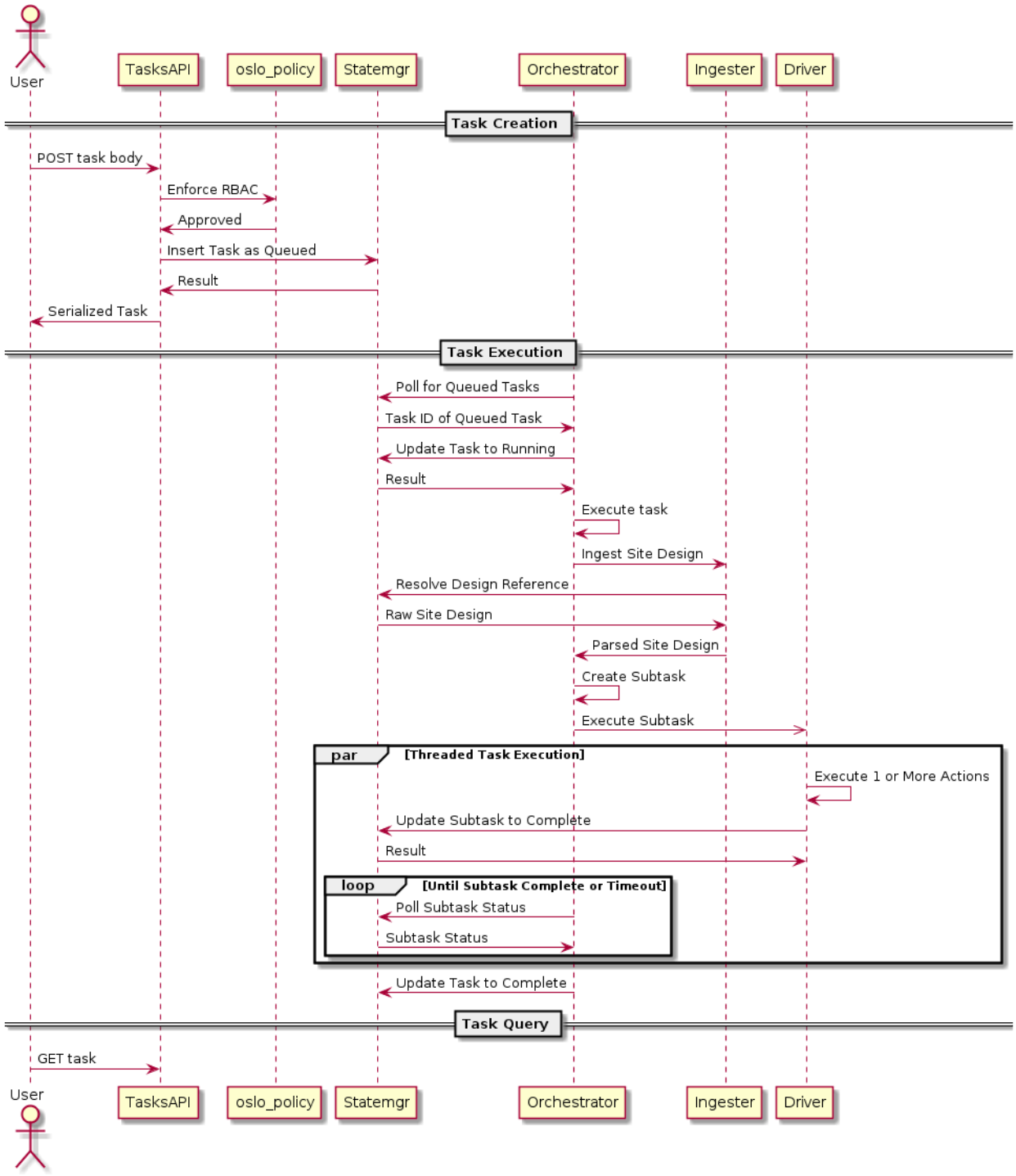
The core objective of Drydock is to fully deploy physical servers based on a declarative YAML topology. The actual provisioning work is completed by a downstream 3rd party tool managed by a pluggable driver. The initial use-case is Canonical MAAS.

Architecture



At a very high level Drydock is a very simple workflow engine fronted by a RESTful API and maintains state in a Postgres relational database. Clients create a task via the API that defines two main attributes of an action and a reference to a site design or topology. The Drydock orchestrator will asynchronously execute the task while the client

polls the API for task status. Once execution is complete, the task status is updated with results and the orchestrator will move to the next queued task.



Components

Control

The `control` module is simply the RESTful API. It is based on the Falcon Framework and utilizes oslo_policy for RBAC enforcement of the API endpoints. The normal deployment of Drydock uses uWSGI and PasteDeploy to build a pipeline that includes Keystone Middleware for authentication and role decoration of the request.

Statemgr

The `statemgr` module is the interface into all backing stores for Drydock. This is mainly a Postgres, but Drydock also uses the state manager for accessing external URLs to ingest site designs. Interactions with Postgres use the core libraries of SQLAlchemy (not the ORM).

Ingester

The `ingester` module is basically a pluggable translator between external site definitions (currently supports YAML formats) and the internal object model. Most of the internal object model utilizes oslo_versionedobjects, much to my regret.

Orchestrator

The `orchestrator` module is the brain of the task execution. It requests queued tasks from the state manager and when one is available, it executes it. The orchestrator is single-threaded in that only a single user-created task is executed at once. However, that task can spawn many subtasks that may be executed concurrently depending on their synchronization requirements. For some actions, the orchestrator creates subtasks that are handed off to the driver for execution. A common question about this module is why Drydock doesn't use Celery as a task management engine. The simple answer is that it wasn't considered due to unfamiliarity at the time.

Driver

The `driver` module is a framework that supports pluggable drivers to execute task actions. The subtle difference between the `driver` and `orchestrator` modules is the orchestrator manages a wide scope of task execution that may cross the boundaries of a single driver plugin. Each driver plugin is more focused on using a single downstream tool to accomplish the actions.

Developer Workflow / Test Cycle

Because Airship is a container-centric platform, the developer workflow heavily utilizes containers for testing and publishing. It also requires Drydock to produce multiple artifacts that are related, but separate: the Python package, the Docker image and the Helm chart. The code is published via the Docker image artifact.

Drydock strives to conform to the Airship coding conventions.

Python

The Drydock Python codebase is under `/drydock_provisioner` and the testing is under `/tests`. The developer tools expect to run on Ubuntu 16.04 and you'll need GNU `make` available. With that you should be able to use make targets for testing code changes:

- `make pep8` - Lint the Python code against the PEP8 coding standard

- `make unit_tests` - Run the local unit tests
- `make security` - Scan the code with Bandit
- `make coverage_test` - Run unit tests and Postgres integration tests

Docker

The Drydock dockerfile is located in `/images/drydock` along with any artifacts built specifically to enable the container image. Again make targets are used for generating and testing the artifacts.

- `make images` - Build the Drydock Docker image. See *Makefile Options* below.
- `make run_images` - Build the image and then run a rudimentary local test

Helm

The Drydock helm chart is located in `/charts/drydock`. Local testing currently only supports linting and previewing the rendered artifacts. Richer functional chart testing is a TODO.

- `make helm_lint` - Lint the Helm chart
- `make dry-run` - Render the chart and output the Kubernetes manifest YAML documents

Makefile Options

The Makefile supports a few options that override default values to allow use behind a proxy or for geneting the Docker image with custom tags.

- `DOCKER_REGISTRY` - Defaults to `quay.io`, used as the Docker registry for tagging images
- `IMAGE_NAME` - Defaults to `drydock`, the image name.
- `IMAGE_PREFIX` - Defaults to `airshipit`, the registry organization to push images into
- `IMAGE_TAG` - Defaults to `dev`, a tag to apply to the image
- **`PUSH_IMAGE` - Defaults to `false`, set to `true` if you want the build process to also** push the image. Likely will require you have previously run `docker login`.
- **`PROXY` - A HTTP/HTTPS proxy server to add to the image build environment. Required if you** are building the image behind a proxy.
- **`USE_PROXY` - Defaults to `false`, set to `true` to include the `PROXY` configuration** above in the build.

1.4 Client Documentation

1.4.1 drydock_client - client for drydock_provisioner RESTful API

The drydock_client module can be used to access a remote (or local) Drydock REST API server. It supports tokenized authentication and marking API calls with an external context marker for log aggregation.

It is composed of two parts - a DrydockSession which denotes the call context for the API and a DrydockClient which gives access to actual API calls.

Simple Usage

The usage pattern for drydock_client is to build a DrydockSession with your credentials and the target host. Then use this session to build a DrydockClient to make one or more API calls. The DrydockSession will care for TCP connection pooling and header management:

```
import drydock_provisioner.drydock_client.client as client
import drydock_provisioner.drydock_client.session as session

dd_session = session.DrydockSession('host.com', port=9000, token='abc123')
dd_client = client.DrydockClient(dd_session)

drydock_task = dd_client.get_task('ba44e582-6b26-11e7-81cc-080027ef795a')
```

Drydock Client Method API

drydock_client.client.DrydockClient supports the following methods for accessing the Drydock RESTful API

get_design_ids

Return a list of UUID-formatted design IDs

get_design

Provide a UUID-formatted design ID, receive back a dictionary representing an objects.site.SiteDesign instance. You can provide the kwarg 'source' with the value of 'compiled' to see the site design after inheritance is applied.

create_design

Create a new design. Optionally provide a new base design (by UUID-formatted design_id) that the new design uses as the starting state. Receive back a UUID-formatted string of design_id

get_part

Get the attributes of a particular design part. Provide the design_id the part is loaded in, the kind (one of `Region`, `NetworkLink`, `Network`, `HardwareProfile`, `HostProfile` or `BaremetalNode` and the part key (i.e. name). You can provide the kwarg 'source' with the value of 'compiled' to see the site design after inheritance is applied.

load_parts

Parse a provided YAML string and load the parts into the provided design context

get_tasks

Get a list of all task ids

get_task

Get the attributes of the task identified by the provided task_id

create_task

Create a task to execute the provided action on the provided design context

1.5 Topology Documentation

1.5.1 Authoring Site Topology

Drydock uses a YAML-formatted site topology definition to configure downstream drivers to provision baremetal nodes. This topology describes the networking configuration of a site as well as the set of node configurations that will be deployed. A node configuration consists of network attachment, network addressing, local storage, kernel selection and configuration and metadata.

The best source for a sample of the YAML schema for a topology is the unit test input source in `./tests/yaml_samples/fullsite.yaml`.

Defining Networking

Network definitions in the topology are described by two document types: NetworkLink and Network. NetworkLink describes a physical or logical link between a node and switch. It is concerned with attributes that must be agreed upon by both endpoints: bonding, media speed, trunking, etc. A Network describes the layer 2 and layer 3 networks accessible over a link.

Network Links

The NetworkLink document defines layer 1 and layer 2 attributes that should be in-sync between the node and the switch. Each link can support a single untagged VLAN and 0 or more tagged VLANs.

Example YAML schema of the NetworkLink spec:

```
spec:
  bonding:
    mode: 802.3ad
    hash: layer3+4
    peer_rate: slow
  mtu: 9000
  linkspeed: auto
  trunking:
    mode: 802.1q
  allowed_networks:
    - public
    - mgmt
```

`bonding` describes combining multiple physical links into a single logical link (aka LAG or link aggregation group).

- `mode`: What bonding mode to configure
  - `disabled`: Do not configure a bond

  - `802.3ad`: Use 802.3ad dynamic aggregation (aka LACP)
  - `active-backup`: Use static active/standby bonding
  - `balanced-rr`: Use static round-robin bonding

For a `mode` of `802.3ad` the optional attributes below are available:

- `hash`: The link selection hash. Supported values are `layer3+4`, `layer2+3`, `layer2`. Default is `layer3+4`
- `peer_rate`: How frequently to send LACP control frames. Supported values are `fast` and `slow`. Default is `fast`
- `mon_rate`: Interval between checking link state in milliseconds. Default is `100`
- `up_delay`: Delay in milliseconds between a link coming up and being marked up in the bond. Must be greater than `mon_rate`. Default is `200`
- `down_delay`: Delay in milliseconds between a link going down and being marked down in the bond. Must be greater than `mon_rate`. Default is `200`

`mtu` is the maximum transmission unit for the link. It must be equal or greater than the MTU of any VLAN interfaces using the link. Default is `1500`.

`linkspeed` is the physical layer speed and duplex. Recommended to always be `auto`

`trunking` describes how multiple layer 2 networks will be multiplexed on the link.

- `mode`: Can be `disabled` for no trunking or `802.1q` for standard VLAN tagging
- `default_network`: For `mode: disabled`, this is the single network on the link. For `mode: 802.1q` this is optionally the network accessed by untagged frames.

`allowed_networks` is a sequence of network names listing all networks allowed on this link. Each Network can be listed on one and only one NetworkLink.

Network

The Network document defines the layer 2 and layer 3 networks nodes will access. Each Network is accessible over exactly one NetworkLink. However that NetworkLink can be attached to different interfaces on different nodes to support changing hardware configurations.

Example YAML schema of the Network spec:

```
spec:
  vlan: '102'
  mtu: 1500
  cidr: 172.16.3.0/24
  routedomain: storage
  ranges:
    - type: static
      start: 172.16.3.15
      end: 172.16.3.200
    - type: dhcp
      start: 172.16.3.201
      end: 172.16.3.254
  routes:
    - subnet: 0.0.0.0/0
      gateway: 172.16.3.1
      metric: 10
```

(continues on next page)

(continued from previous page)

```
    - gateawy: 172.16.3.2
      metric: 10
      routedomain: storage
  dns:
    domain: sitename.example.com
    servers: 8.8.8.8
```

If a Network is accessible over a NetworkLink using 802.1q VLAN tagging, the `vlan` attribute specified the VLAN tag for this Network. It should be omitted for non-tagged Networks.

`mtu` is the maximum transmission unit for this Network. Must be equal or less than the `mtu` defined for the hosting NetworkLink. Can be omitted to default to the NetworkLink `mtu`.

`cidr` is the classless inter-domain routing address for the network.

`routedomain` is a logical grouping of L3 networks such that a network that describes a static route for accessing the route domain will yield a list of static routes for all the networks in the routedomain. See the description of `routes` below for more information.

`ranges` defines a sequence of IP addresses within the defined `cidr`. Ranges cannot overlap.

- `type`: The type of address range.
  - `static`: A range used for static, explicit address assignments for nodes.
  - `dhcp`: A range used for assigning DHCP addresses. Note that a network being used for PXE booting must have a DHCP range defined.
  - `reserved`: A range of addresses that will not be used by MaaS.
- `start`: The starting IP of the range, inclusive.
- `end`: The last IP of the range, inclusive

`routes` defines a list of static routes to be configured on nodes attached to this network. The routes can defined in one of two ways: an explicit destination `subnet` where the route will be configured exactly as described or a destination `routedomain` where Drydock will calculate all the destination L3 subnets for the routedomain and add routes for each of them using the `gateway` and `metric` defined.

- `subnet`: Destination CIDR for the route
- `gateway`: The gateway IP on this Network to use for accessing the destination
- `metric`: The metric or weight for this route
- **`routedomain`: Use this route's gateway and metric for accessing networks in the** defined routedomain.

`dns` is used for specifying the list of DNS servers to use if this network is the primary network for the node.

- `servers`: A comma-separated list of IP addresses to use for DNS resolution
- `domain`: A domain that can be used for automated registration of IP addresses assigned from this Network

DHCP Relay

DHCP relaying is used when a DHCP server is not attached to the same layer 2 broadcast domain as nodes that are being PXE booted. The DHCP requests from the node are consumed by the relay (generally configured on a top-of-rack switch) which then encapsulates the request in layer 3 routing and sends it to an upstream DHCP server. The Network spec supports a `dhcp_relay` key for Networks that should relay DHCP requests.

- The Network must have a configured DHCP relay, this is *not* configured by Drydock or MaaS.

- The `upstream_target` IP address must be a host IP address for a MaaS rack controller
- The Network must have a defined DHCP address range.
- The upstream target network must have a defined DHCP address range.

The `dhcp_relay` stanza:

```
dhcp_relay:
  upstream_target: 172.16.4.100
```

Defining Node Configuration

Node configuration is defined in three documents: `HostProfile`, `HardwareProfile` and `BaremetalNode`. `HardwareProfile` defines attributes directly related to hardware configuration such as card-slot layout and firmware levels. `HostProfile` is a generic definition for how a node should be configured such that many nodes can reference a single `HostProfile` and each will be configured identically. A `BaremetalNode` is a concrete reference to the particular physical node. The `BaremetalNode` definition will reference a `HostProfile` and can then extend or override any of the configuration values.

NOTE: Drydock does not support hostnames containing '__' (double underscore)

Hardware Profile

The hardware profile is used to convert some abstractions in the HostProfile documents into concrete configurations based a particular hardware build. A host profile will designate how the bootdisk should be configured, but the hardware profile will designate which exact device is used for the bootdisk. This allows a heterogeneous mix of hardware in a site without duplicating definitions of how that hardware should be configured.

An example HardwareProfile document:

```
---
schema: 'drydock/HardwareProfile/v1'
metadata:
  schema: 'metadata/Document/v1'
  name: AcmeServer
  storagePolicy: 'cleartext'
  labels:
    application: 'drydock'
data:
  vendor: HP
  generation: '8'
  hw_version: '3'
  bios_version: '2.2.3'
  boot_mode: bios
  bootstrap_protocol: pxe
  pxe_interface: 0
  device_aliases:
    prim_nic01:
      address: '0000:00:03.0'
      dev_type: '82540EM Gigabit Ethernet Controller'
      bus_type: 'pci'
    prim_nic02:
      address: '0000:00:04.0'
      dev_type: '82540EM Gigabit Ethernet Controller'
      bus_type: 'pci'
```

(continues on next page)

(continued from previous page)

```
    primary_boot:
      address: '2:0.0.0'
      dev_type: 'VBOX HARDDISK'
      bus_type: 'scsi'
  cpu_sets:
    sriov: '2,4'
  hugepages:
    sriov:
      size: '1G'
      count: 300
    dpdk:
      size: '2M'
      count: 530000
```

Device Aliases

Device aliases are a way of mapping a particular device bus address to an alias. In the example above we map the PCI address `0000:00:03.0` to the alias `prim_nic01`. A host profile or baremetal node definition can then provide a configuration using `prim_nic01` and Drydock will translate that to the correct operating system device name for the NIC device at PCI address `0000.00.03.0`. Currently device aliases are supported for network interface slave devices and storage physical devices.

Kernel Parameter References

Some kernel parameters specified in a host profile rely on particular hardware builds, such as `isolcpus`. To support the greatest flexibility in building host profiles, you can specify a few values in a hardware profile that will then be sourced when needed by a host profile or baremetal node definition.

- `cpu_sets`: Each key should have a value of a comma-separated list of CPUs/cores/hyperthreads that would be appropriate for the `isolcpus` kernel parameters. A host profile can then select any one of these CPU sets for a host.
- `hugepages`: Each key should have a value of a mapping containing two keys: `size` and `count`. Again, a host profile can then select these values when defining kernel parameters for a host. Note the `size` field is a string and will be used as-is, so the format must be usable by the kernel.

Host Profiles and Baremetal Nodes

Example `HostProfile` and `BaremetalNode` configuration:

```
---
apiVersion: 'drydock/v1'
kind: HostProfile
metadata:
  name: defaults
  region: sitename
  date: 17-FEB-2017
  author: sh8121@att.com
spec:
  # configuration values
---
apiVersion: 'drydock/v1'
```

(continues on next page)

(continued from previous page)

```
kind: HostProfile
metadata:
  name: compute_node
  region: sitename
  date: 17-FEB-2017
  author: sh8121@att.com
spec:
  host_profile: defaults
  # compute_node customizations to defaults
---
apiVersion: 'drydock/v1'
kind: BaremetalNode
metadata:
  name: compute01
  region: sitename
  date: 17-FEB-2017
  author: sh8121@att.com
spec:
  host_profile: compute_node
  # configuration customization specific to single node compute01
```

In the above example, the *compute_node* `HostProfile` adopts all values from the *defaults* `HostProfile` and can then override defined values or append additional values. `BaremetalNode` *compute01* then adopts all values from the *compute_node* `HostProfile` (which includes all the configuration items it adopted from *defaults*) and can then again override or append any configuration that is specific to that node.

Defining Node Out-Of-Band Management

Drydock supports plugin-based OOB management. At a minimum a OOB driver supports configuring a node to PXE boot during the next boot cycle and power cycling the node to initiate the provisioning process. Richer features might also be supported such as BIOS configuration or BMC log analysis. The value of `oob.type` in the host profile or baremetal node definition will define what additional parameters are required for that type and what capabilities are available via OOB driver tasks.

IPMI

The `ipmi` OOB type requires additional configuration to allow OOB management:

1. The `oob` parameters `account` and `credential` must be populated with a valid account and password that can access the BMC via IPMI over LAN.
2. The `oob` parameter `network` must reference which node network is used for OOB access.
3. The `addressing` section of the node definition must contain an IP address assignment for the network referenced in `oob.network`.

Currently the IPMI driver supports only basic management by setting nodes to PXE boot and power-cycling the node.

Libvirt

The `libvirt` OOB type requires additional configuration within the site definition as well as particular configuration in the deployment of Drydock (and likely the node provisioning driver.):

1. A SSH public/private key-pair should be generated with the public key being added to the authorized_keys file on all hypervisors hosting libvirt-based VMs being deployed. The account for this must be in the `libvirt` group.
2. The private key should be provided in the Drydock and MAAS charts as an override to `conf.ssh.private_key`
3. The Drydock and MAAS chart should override `manifests.secret_ssh_key: true`.
4. In the site definition, each libvirt-based node must define `oob` parameter `libvirt_uri` of the form `qemu+ssh://account@hostname/system` where `account` is an account in the libvirt group on the hypervisor with an authorized_key and `hostname` is an IP address or FQDN for the hypervisor hosting the VM.

Currently the Libvirt driver supports only basic management by setting nodes to PXE boot and power-cycling the node.

Defining Node Interfaces and Network Addressing

Node network attachment can be described in a `HostProfile` or a `BaremetalNode` document. Node addressing is allowed only in a `BaremetalNode` document. If a `HostProfile` or `BaremetalNode` needs to remove a defined interface from an inherited configuration, it can set the mapping value for the interface name to `null`.

Once the interface attachments to networks is defined, `HostProfile` and `BaremetalNode` specs must define a `primary_network` attribute to denote which network the node should use as the primary route.

Interfaces

Interfaces for a node can be described in either a `HostProfile` or `BaremetalNode` definition. This will attach a defined NetworkLink to a host interface and define which Networks should be configured to use that interface.

Example interface definition YAML schema:

```
interfaces:
  pxe:
    device_link: pxe
    labels:
      pxe: true
    slaves:
      - prim_nic01
    networks:
      - pxe
  bond0:
    device_link: gp
    slaves:
      - prim_nic01
      - prim_nic02
    networks:
      - mgmt
      - private
```

Each key in the interfaces mapping is a defined interface. The key is the name that will be used on the deployed node for the interface. The value must be a mapping defining the interface configuration or `null` to denote removal of that interface for an inherited configuration.

- `device_link`: The name of the defined NetworkLink that will be attached to this interface. The NetworkLink definition includes part of the interface configuration such as bonding.

- `labels`: Metadata for describing this interface.
- `slaves`: The list of hardware interfaces used for creating this interface. This value can be a device alias defined in the HardwareProfile or the kernel name of the hardware interface. For bonded interfaces, this would list all the slaves. For non-bonded interfaces, this should list the single hardware interface used.
- `networks`: This is the list of networks to enable on this interface. If multiple networks are listed, the NetworkLink attached to this interface must have trunking enabled or the design validation will fail.

Addressing

Addressing for a node can only be defined in a `BaremetalNode` definition. The `addressing` stanza simply defines a static IP address or `dhcp` for each network a node should have a configured layer 3 interface on. It is a valid design to omit networks from the `addressing` stanza, in that case the interface attached to the omitted network will be configured as link up with no address.

Example `addressing` YAML schema:

```
addressing:
  - network: pxe
    address: dhcp
  - network: mgmt
    address: 172.16.1.21
  - network: private
    address: 172.16.2.21
  - network: oob
    address: 172.16.100.21
```

Defining Node Storage

Storage can be defined in the `storage` stanza of either a HostProfile or BaremetalNode document. The storage configuration can describe the creation of partitions on physical disks, the assignment of physical disks and/or partitions to volume groups, and the creation of logical volumes. Drydock will make a best effort to parse out system-level storage such as the root filesystem or boot filesystem and take appropriate steps to configure them in the active node provisioning driver. At a minimum, the storage configuration *must* contain a root filesystem partition.

Example YAML schema of the `storage` stanza:

```
storage:
  physical_devices:
    sda:
      labels:
        bootdrive: true
      partitions:
        - name: 'root'
          size: '10g'
          bootable: true
          filesystem:
            mountpoint: '/'
            fstype: 'ext4'
            mount_options: 'defaults'
        - name: 'boot'
          size: '1g'
          filesystem:
            mountpoint: '/boot'
```

(continues on next page)

(continued from previous page)

```
            fstype: 'ext4'
            mount_options: 'defaults'
    sdb:
      volume_group: 'log_vg'
  volume_groups:
    log_vg:
      logical_volumes:
        - name: 'log_lv'
          size: '500m'
          filesystem:
            mountpoint: '/var/log'
            fstype: 'xfs'
            mount_options: 'defaults'
```

Schema

The `storage` stanza can contain two top-level keys: `physical_devices` and `volume_groups`. The latter is optional.

Physical Devices and Partitions

A physical device can either be carved up in partitions (including a single partition consuming the entire device) or added to a volume group as a physical volume. Each key in the `physical_devices` mapping represents a device on a node. The key should either be a device alias defined in the HardwareProfile or the name of the device published by the OS. The value of each key must be a mapping with the following keys

- `labels`: A mapping of key/value strings providing generic labels for the device
- `partitions`: A sequence of mappings listing the partitions to be created on the device. The mapping is described below. Incompatible with the `volume_group` specification.
- `volume_group`: A volume group name to add the device to as a physical volume. Incompatible with the `partitions` specification.

Partition

A partition mapping describes a GPT partition on a physical disk. It can be left as a raw block device or formatted and mounted as a filesystem.

- `name`: Metadata describing the partition in the topology
- `size`: The size of the partition. See the *Size Format* section below
- `bootable`: Boolean whether this partition should be the bootable device
- `part_uuid`: A UUID4 formatted UUID to assign to the partition. If not specified one will be generated
- `filesystem`: An optional mapping describing how the partition should be formatted and mounted
  - `mountpoint`: Where the filesystem should be mounted. If not specified the partition will be left as a raw device
  - `fstype`: The format of the filesystem. Defaults to ext4
  - `mount_options`: fstab style mount options. Default is 'defaults'

  - `fs_uuid`: A UUID4 formatted UUID to assign to the filesystem. If not specified one will be generated
  - `fs_label`: A filesystem label to assign to the filesystem. Optional.

Size Format

The size specification for a partition or logical volume is formed from three parts:

- The first character can optionally be > indicating that the size specified is a minimum and the calculated size should be at least the minimum and should take the rest of the available space on the physical device or volume group.
- The second part is the numeric portion and must be an integer
- The third part is a label
  - m|M|mb|MB: Megabytes or 10^6 * the numeric
  - g|G|gb|GB: Gigabytes or 10^9 * the numeric
  - t|T|tb|TB: Terabytes or 10^12 * the numeric
  - %: The percentage of total device or volume group space

Volume Groups and Logical Volumes

Logical volumes can be used to create RAID-0 volumes spanning multiple physical disks or partitions. Each key in the `volume_groups` mapping is a name assigned to a volume group. This name must be specified as the `volume_group` attribute on one or more physical devices or partitions or the configuration is invalid. Each mapping value is another mapping describing the volume group.

- `vg_uuid`: A UUID4 format uuid applied to the volume group. If not specified, one is generated
- `logical_volumes`: A sequence of mappings listing the logical volumes to be created in the volume group

Logical Volume

A logical volume is a RAID-0 volume. Using logical volumes for `/` and `/boot` is supported

- `name`: Required field. Used as the logical volume name.
- `size`: The logical volume size. See *Size Format* above for details.
- `lv_uuid`: A UUID4 format uuid applied to the logical volume: If not specified, one is generated
- `filesystem`: A mapping specifying how the logical volume should be formatted and mounted. See the *Partition* section above for filesystem details.

Platform Configuration

In the `platform` stanza you can define the operating system `image` and `kernel` to use as well as customize the kernel configuration with `kernel_params`.

The valid `image` and `kernel` values are dependent on what is supported by your node provisioner. In the example of Canonical MaaS using the 16.04 LTS image, the values would be `image: 'xenial'` and `kernel: 'ga-16.04'` for the LTS kernel or `kernel: hwe-16.04` for the hardware-enablement kernel.

The `kernel_params` configuration is a mapping. Each key should either be a string or boolean value. For boolean `true` values, the key will be added to the kernel parameter list as a flag. For string values, the key:value pair will be added to the kernel parameter list as `key=value`.

One special case is supported for values that match a hardware profile reference. When the parameter is rendered for a particular node, the value included in the kernel parameter list will be sourced from the effective HardwareProfile assigned to the node.

- `hardwareprofile:cpuset.<name>`: Sourced from the hardware profile `cpu_sets.<name>` value.
- `hardwareprofile.hugepages.<name>.size`: Source from the hardware profile `hugepages.<name>.size` value.
- `hardwareprofile.hugepages.<name>.count`: Source from the hardware profile `hugepages.<name>.count` value.

1.5.2 Drydock Troubleshooting Guide

This is a guide for troubleshooting issues that can arise when either installing/deploying Drydock or using Drydock to deploy nodes.

Deployment Troubleshooting

Under Construction

Topology Validation

Dryodck Topology Validation

DD1XXX - Storage Validations

To be continued

DD2XXX - Network Validations

To be continued

DD3XXX - Platform Validations

To be continued

DD4XXX - Bootaction Validations

To be continued

Node Deployment

Under Construction