airship-specs Documentation

Release 0.1.0

Airship Authors

Nov 14, 2019

Contents

genindex

CHAPTER 1

About Specs

1.1 Instructions

- Use the template.rst as the basis of your specification.
- Attempt to detail each applicable section.
- If a section does not apply, use N/A, and optionally provide a short explanation.
- New specs for review should be placed in the `approved` subfolder of `1.x` or `2.x` directories (depending on Airship version they primarily belong to), where they will undergo review and approval in Gerrit.
- Test if the spec file renders correctly in a web-browser by running `make docs` command and opening `doc/build/html/index.html` in a web-browser. Ubuntu needs the following packages to be installed:

```
apt-get install -y make tox gcc python3-dev
```

- Specs that have finished implementation should be moved to the `implemented` subfolder of respective `1.x` or `2.x` directories.

1.1.1 Indexing and Categorization

Use of the index directive in reStructuredText for each document provides the ability to generate indexes to more easily find items later. Authors are encouraged to use index entries for their documents to help with discovery.

1.1.2 Naming

Document naming standards help readers find specs. For the Airship repository, the following document naming is recommended. The categories listed here are likely incomplete, and may need expansion to cover new cases. It is preferrable to deviate (and hopefully amend the list) than force document names into nonsense categories. Prefer using categories that have previously been used or that are listed here over new categories, but don't force the category into something that doesn't make sense.

Document names should follow a pattern as follows:

```
[category]_title.rst
```

Use the following guidelines to determine the category to use for a document:

1. For new functionality and features, the best choice for a category is to match a functional duty of Airship.

   site-definition Parts of the platform that support the definition of a site, including management of the yaml definitions, document authoring and translation, and the collation of source documents.

   genesis Used for the steps related to preparation and deployment of the genesis node of an Airship deployment.

   baremetal Those changes to Airflow that provide for the lifecycle of bare metal components of the system - provisioning, maintenance, and teardown. This includes booting, hardware and network configuration, operating system, and other host-level management

   k8s For functionality that is about interfacing with Kubernetes directly, other than the initial setup that is done during genesis.

   software Functionality that is related to the deployment or redeployment of workload onto the Kubernetes cluster.

   workflow Changes to existing workflows to provide new functionality and creation of new workflows that span multiple other areas (e.g. baremetal, k8s, software), or those changes that are new arrangements of existing functionality in one or more of those other areas.

   administration Security, logging, auditing, monitoring, and those things related to site administrative functions of the Airship platform.

2. For specs that are not feature focused, the component of the system may be the best choice for a category, e.g. `shipyard`, `armada` etc… When there are multiple components involved, or the concern is cross cutting, use of `airship` is an acceptable category.

3. If the spec is related to the ecosystem Airship is maintained within, an appropriate category would be related to the aspect it is impacting, e.g.: `git`, `docker`, `zuul`, etc…

Note: Blueprints are written using ReSTructured text.

Add *index* directives to help others find your spec by keywords. E.g.:

```
.. index::
   single: template
   single: creating specs
```

1.2 Template: The title of your blueprint

Introduction paragraph – What is this blueprint about?

1.2.1 Links

Include pertinent links to where the work is being tracked (e.g. Storyboard ID and Gerrit topics), as well as any other foundational information that may lend clarity to this blueprint

1.2.2 Problem description

A detailed description of the problem being addressed or solved by this blueprint

1.2.3 Impacted components

List the Airship components that are impacted by this blueprint

1.2.4 Proposed change

Provide a detailed description of the change being proposed. Include how the problem will be addressed or solved.

If this is an incremental part of a larger solution or effort, provide the specific scope of this blueprint, and how it fits into the overarching solution.

Details of changes to specific Airship components should be specified in this section, as well as interaction between those components.

Special attention should be given to interfaces between components. New interfaces shuld attempt to follow established patterns within Airship, or should be evaluated for suitability as new precedent.

If this blueprint changes testing needs or approaches, that information should be disclosed here, and should be regarded as part of the deliverable related to this design.

If this blueprint introduces new functionality that requires new kinds of documentation, or a change to the documentation processes, that information should be included in this section.

Security impact

Details of any security-related concerns that this proposed change introduces or addresses.

Performance impact

Analysis of performance changes that are introduced or addressed with this proposed design.

Alternatives

If other approaches were considered, include a summary of those here, and a short discussion of why the proposed approach is preferred.

1.2.5 Implementation

If known, include any information detailing assigned individuals, proposed milestones, intermediate deliverable products, and work items.

If there are Assignee(s) or Work Items, use a sub-heading for that information.

1.2.6 Dependencies

If there are any dependencies on other work, blueprints, or other things that impact the ability to deliver this solution, include that information here.

1.2.7 References

Any external references (other than the direct links above)

CHAPTER 2

Airship 2.x

2.1 Approved Specs

2.1.1 Airshipctl Bootstrap Image Generator

This spec defines the new `isogen` sub-command for `airshipctl bootstrap` and describes the interface for image builder. Airship CLI tool will be extended with an ability to generate an ISO image or image for USB stick. This image can be used to boot up an ephemeral node with Kubernetes cluster installed.

Links

Jira tasks:

- airshipctl bootstrap isogen
- LiveCD PoC
- ISO builder contract spec
- isogen subcommand spec
- Sub command implementation
- Cloud-init generator

Problem Description

Common approach for spinning new Kubernetes cluster is Cluster API deployed on top of a small single node cluster based on `kind` or `minikube`. In order to create Kubernetes cluster on hardware nodes in Data Center user must deploy this single node cluster on a virtual machine attached to PXE network or to deploy operating system and Kubernetes cluster to one of the hardware servers.

In scope of Airship 2.0 user needs to be able to bootstrap ephemeral Kubernetes cluster with minimal required services (e.g. Cluster API, Metal3, etc). Ephemeral Cluster should be deployed remotely (if possible) and deployment process needs to be fully automated.

Impacted Components

- airshipctl

Proposed Change

Airship 2.0 command line tool (i.e. `airshipctl`) will be able to perform full cycle of bootstrapping ephemeral Kubernetes node.

First bootstrap step is to generate ISO or flash drive image. Image generator is executed inside of a container and returns LiveCD or LiveUSB image.

Image generator must implement interface defined below (see *Image Generator Container Interface* section) since `isogen` command treats image generator container as a black box.

Airshipctl Subcommand

`airshipctl bootstrap` is extended with `isogen` subcommand. Subcommand is extendable by adding Container Runtime Interface drivers.

Command flags:

- `-c` or `--conf` Configuration file (YAML-formatted) path for ISO builder container. If option is omitted `airshipctl config` is used to determine isogen configuration file path. This configuration file is used to identify container execution options (e.g. CRI, volume binds etc) and as a source of ISO builder parameters (e.g. cloud-init configuration file name). File format described in *Command and ISO Builder Configuration File Format*

Command arguments:

- `-` can be used when rendered document model has been passed to STDIN.

Subcommand should implement following steps:

- Utilize the `airshipctl config` to identify the location of YAML documents which contains site information.
- Extract information for ephemeral node from the appropriate documents, such as IP, Name, MAC, etc.
- Generate the appropriate user-data and network-config for Cloud-Init.
- Execute container with ISO builder and put YAML-formatted builder config, user-data and network-config to a container volume.

YAML manipulations which are required for operations described above rely on functions and methods that have been implemented as a part of `airshipctl document` command.

Image Generator Container Interface

Image generator container input.

- Volume (host directory) mounted to certain directory in container. Example: `docker run -v /source/path/on/host:/dst/path/in/container ...`

- YAML-formatted configuration file saved on the mounted volume. Described in *Command and ISO Builder Configuration File Format*
- Shell environment variable `BUILDER_CONFIG` which contains ISO builder configuration file path (e.g. if volume is bound to `/data` in the container then `BUILDER_CONFIG=/data/isogen.yaml`).
- Cloud-init configuration file named according to `userDataFileName` parameter of `builder` section specified in ISO builder configuration file. User data file must be placed to the root of the volume which is bound to the container.
- Network configuration for cloud init (i.e. network-config) named according to `networkConfigFileName` parameter of `builder` section specified in ISO builder configuration file. Network configuration file must be placed in the root of the volume which is bound to the container.

Image generator output.

- YAML-formatted metadata file which describes output artifacts. File name for metadata is specified in `builder` section of ISO builder configuration file (see *Command and ISO Builder Configuration File Format* for details). Metadata file name is specified in `aitshipctl` configuration files and handeled by `airshipctl config` command. Metadata must satisfy following schema.

  ```yaml
  $schema: 'http://json-schema.org/schema#'
  type: 'object'
  properties:
    bootImagePath:
      type: 'string'
      description: >
        Image file path on host. Host path of the volume is extracted
        from ISO builder confgiration file passed by isogen command to
        container volume.
  ```

- ISO or flash disk image placed according to `bootImagePath` parameter of output metadata file.

Command and ISO Builder Configuration File Format

YAML formatted configuration file is used for both isogen command and ISO builder container. Configuration file is copied to volume directory on the host. ISO builder uses shell environment variable `BUILDER_CONFIG` to read determine configuration file path inside container.

Configuration file format.

```yaml
$schema: 'http://json-schema.org/schema#'
type: 'object'
properties:
  container:
    type: 'object'
    description: 'Configuration parameters for container'
    properties:
      volume:
        type: 'string'
        description: >
          Container volume directory binding.
          Example: /source/path/on/host:/dst/path/in/container
      image:
        type: 'string'
        description: 'ISO generator container image URL'
      containerRuntime:
        type: 'string'
```

(continues on next page)

(continued from previous page)

```
        description: >
          (Optional) Container Runtime Interface driver (default: docker)
      privileged:
        type: 'bool'
        description: >
          (Optional)Defines if container should be started in privileged mode
          (default: false)
  builder:
    type: 'object'
    description: 'Configuration parameters for ISO builder'
    properties:
      userDataFileName:
        type: 'string'
        description: >
          Cloud Init user-data file name placed to the container volume root
      networkConfigFileName:
        type: 'string'
        description: >
          Cloud Init network-config file name placed to the container
          volume root
      outputMetadataFileName:
        type: 'string'
        description: 'File name for output matadata'
```

Security Impact

- Kubernetes Certificates are saved on the ISO along with other Cloud Init configuration parameters.
- Clound-init contains sensitive information (e.g. could contain ssh keys).

Performance impact

None

Alternatives

- Modify existing LiveCD ISO image using Golang library.
  - Requires implementation of ISO modification module in Golang.
  - Each time user generated new image ISO content has to be copied to temporary build directory since ISO 9660 is read only file system.
  - Support multiple operating systems is challenging since there is no standard for ISO image directory structure and live booting.

Implementation

- Image Generator reference implementation based on Debian container from airship/images Git repository
  - Dockerfile with all packages required to build LiveCD ISO.
  - Builder script.

- `airshipctl bootstrap` extension with new command (i.e. `airshipctl bootstrap isogen`)
  - Define interface for running container execution which enables following methods:
    - Pull image: download container image if it's not presented locally
    - Run container: start container, wait for builder script is finished, output builder log if CLI debug flag is enabled
    - Run container with output: executes run container method and prints its STDOUT
    - Remove container: removes container if command execution successful.
  - Implement interface for docker Container Runtime Environment

Dependencies

- New version of hardware nodes definition format in Treasuremap since Metal3-IO will replace MAAS for Airship 2.0

References

None

2.2 Implemented Specs

2.2.1 Placeholder

Please, remove me once any new spec is added into this directory.

CHAPTER 3

Airship 1.x

3.1 Approved Specs

3.1.1 Airship Copilot

Copilot is an Electron application that can interface with Airship CLIs and REST interfaces. This tool will wrap SSH sessions and HTTP/HTTPS calls to Airship components. The responses will be enhanced with a GUI (links for more commands, color coded, formatting, etc.).

Links

None

Problem description

Airship can be difficult to approach as a user. There are lots of commands to know with lots of data to interpret.

Impacted components

None.

Proposed change

Create an Electron application that simplifies the experience of accessing Airship. The application will be 100% client side, thus no change to the Airship components. The application will default to use HTTP/HTTPS APIs, but will be able to use the CLI commands when needed via an SSH connection. All of the raw commands input and output will be available for the user to see, with the goal of the user not needing to look at the raw input/output.

The application will start as a GUI interface to Shipyard.

- Shipyard - API calls (create, commit, get, logs, etc.) - CLI commands (create, commit, get, logs, etc.) - From a list of actions drill down into logs

The GUI will create links to additional commands based off of the response. The GUI can color code different aspects of the response and format it. An example would be when Shipyard returns a list of tasks, that list can be used to create hyperlinks to drill down on that task (details, logs, etc.).

The GUI could start by looking similar to the CLI. Where the values in the different columns would be buttons/links to call additional commands for more information.

```
Name                 Action                                       Lifecycle      ␣
↪Execution Time            Step Succ/Fail/Oth          Footnotes
deploy_site          action/01BTP9T2WCE1PAJR2DWYXG805V        Failed           2017-09-
↪23T02:42:12           12/1/3                    (1)
update_site          action/01BZZKMW60DV2CJZ858QZ93HRS        Processing       2017-09-
↪23T04:12:21            6/0/10                   (2)
```

Security impact

None - This will continue to use HTTP/HTTPS and SSH just like today, the only difference is that it is wrapped in an application.

Performance impact

Minimal - Wrapping the commands in an Electron application might add a little latency, but only on the client side.

Future plans

Extend to other Airship components. Pegleg seems like the next step, but any componment with an exposed API/CLI.

Dependencies

None

References

3.1.2 Airship Multiple Linux Distribution Support

Various Airship services were developed originally around Ubuntu. This spec will add the ability in Airship to plug in Linux Distro's, refactor the existing Ubuntu support as the default Linux distro plugin, and add openSUSE and other Linux distro's as new plugins.

Links

The work to author and implement this spec is tracked in Storyboard 2003699 and uses Gerrit topics `airship_suse`, `airship_rhel` and similar.

Problem description

Airship was originally developed focusing on the Ubuntu environment:

- While having a well defined driver interface, the baremetal provisioner currently only supports Canonical's MAAS.
- Promenade bootstraps only on a Ubuntu deployer.
- Assumption of Debian packages in various services.
- Builds and references only Ubuntu based container images.

Airship is missing a large user base if only supports Ubuntu.

Impacted components

Most Airship components will be impacted by this spec:

1. Promenade: add the ability to bootstrap on any Linux distro and add new plugins for openSUSE, CentOS, etc.
2. Pegleg: enhanced to build image on non Debian distros and add openSUSE, CentOS and other Linux distros to CI gate.
3. Deckhand: enhanced to build image on non Debian distros and add openSUSE, CentOS and other Linux distros to CI gate.
4. Armada: enhanced to build image on non Debian distro and add openSUSE, CentOS and other Linux distros to CI gate.
5. Shipyard: enhanced to build image on non Debian distro and add openSUSE, CentOS and other Linux distros CI gate.
6. Drydock: enhanced to provision bare metal on non Ubuntu Linux distros using Ironic driver (expect to have a separate spec).
7. Airship-in-a-Bottle: add the ability to deploy Airship-in-a-Bottle on openSUSE, CentOS, etc.

Proposed change

Container Builds

- As for now, Ubuntu-based containers remain to be default to be built
- CI system for the Ubuntu-based containers must not be affected by implementation of this spec (Zuul jobs, Makefile, etc.)
- Distributive-dependant `Dockerfile` naming convention is to add a distributive flavour suffix, optionally specifying version after underscore: `Dockerfile.<distributive flavour>[_<version>]`; e.g. `Dockerfile.opensuse`
- Public non-Ubuntu container images are to be published along with Ubuntu-based images on quay.io under airshipit/ organization
- Repository naming convention remains exactly same for airship component. `airshipit/<airship component>`; e.g. `airshipit/armada`
- Updated image tagging naming convention is to add a dash separator suffix after tags based on Git commit ID of the code `:<git commit id>`, and additional `:master` (branch-based) and `:latest` (latest master) tags, following with a distributive flavour, optionally specifying distribution

version after underscore[1]: `<airship component>:<branch or commit-id>-<distributive flavour>[_<distro major version>]`; e.g. `armada:master-opensuse_15`

- As for now, Makefiles, Shell scripts, Shell script templates in Helm charts, Ansible jobs, Zuul and Jenkins jobs continue to work without changes for Ubuntu-based containers, and support non-Ubuntu containers by provision of additional variables or command line arguments

Pegleg

- Add non Ubuntu Linux distros CI gate, including openSUSE, CentOS, etc.
  - tools/gate/playbooks/docker-image-build.yaml: support Docker rpm install on non Debian Linux.
  - add gate test case for openSUSE, CentOS.

Deckhand

- Container image(s)
  - images/deckhand/Dockerfile: add rpm package support for non Debian Linux distros
- Verify Deckhand Python source code and scripts are Linux distro agnostic
- Update document for rpm package installation, e.g., getting started guide
- Add Non Debian Linux support in gate playbooks
  - tools/gate/playbooks/docker-image-build.yaml
  - tools/gate/playbooks/roles/install-postgresql/tasks/install-postgresql.yaml
  - tools/gate/playbooks/roles/run-integration-tests/tasks/integration-tests.yaml
  - tools/gate/roles/install-postgresql/tasks/install-postgresql.yaml
  - tools/gate/roles/run-integration-tests/tasks/integration-tests.yaml
  - tools/run_pifpaf.sh
  - add gate test case for openSUSE, CentOS, etc.

Shipyard

- Container image(s)
  - images/shipyard/Dockerfile: add rpm package for openSUSE, CentOS, etc.
  - images/airflow/Dockerfile: add rpm package for openSUSE, CentOS, etc.
- Verify Shipyard Python source code and scripts are Linux distro agnostic.
- Update documentation where references Ubuntu and MAAS as the sole option.
  - README.rst
  - docs/source/client-user-guide.rst
  - docs/source/deployment-guide.rst
- Add non Debian Linux support in gate playbooks

[1] Based on recommendation from quay.io technical support.

- tools/gate/playbooks/roles/build-images/tasks/airship-shipyard.yaml
- tools/gate/roles/build-images/tasks/airship-shipyard.yaml
- tools/gate/scripts/000-environment-setup.sh
- add test cases in zuul for openSUSE, CentOS, etc.

Armada

- Container image(s)
  - Dockerfile: add rpm package for non Debian Linux (Docker file location is inconsistent with other projects).
- Verify Python source code and scripts are Linux distro agnostic.
- Update documentation where references Ubuntu and MAAS as the sole option, e.g., getting-started.rst.
- Add non Debian Linux support in gate playbooks
  - Add rpm package support when ansible_os_family is SUSE or Red Hat
  - tools/gate/playbooks/docker-image-build.yaml
  - Add test cases in zuul for openSUSE, CentOS, etc.

Promenade

- Container image(s)
  - Dockerfile: add rpm package for SUSE (Docker file location is inconsistent with other projects)
- Verify Python source code and scripts are Linux distro agnostic, e.g.,
  - Genesis process assumes Debian-based OS. Changes are required to maintain this functionality for other distros as well as logic to pick the right template, e.g., promenade/templates/roles/common/etc/apt/sources.list.d.
  - tests/unit/api/test_update_labels.py: label is hard coded to “ubuntubox”. which seems to be just cosmetics
  - tests/unit/api/test_validatedesign.py: deb for Docker and socat
- Update documentation where references Ubuntu and MAAS as the sole option and add list of docker images for other Linux OS than Ubuntu
  - getting-started.rst
  - developer-onboarding.rst
  - examples: HostSystem.yaml, armada-resources.yaml
- Add non Debian Linux support in gate playbooks
  - tools/gate/config-templates/site-config.yaml: add rpm install for Docker and socat based on os family
  - tools/setup_gate.sh: add rpm install for Docker based on os family
  - tools/zuul/playbooks/docker-image-build.yaml
  - tools/cleanup.sh:
  - add test cases in zuul for openSUSE, CentOS, etc.

Treasuremap

- Update documentation to add authoring and deployment instructions for OpenSUSE, CentOS, etc. Differences are around deb vs rpm packaging, container images, repos.
  - doc/source/authoring_and_deployment.rst
  - global/profiles/kubernetes-host.yaml
  - global/schemas/drydock/Region/v1.yaml
  - global/schemas/promenade/HostSystem/v1.yaml
  - global/software/config/versions.yaml
  - tools/gate/Jenkinsfile
  - global/profiles/kubernetes-host.yaml
  - site/airship-seaworthy/networks/common-addresses.yaml (points to ubuntu ntp server)
  - site/airship-seaworthy/profiles/region.yaml (comments references "ubuntu" user)
  - site/airship-seaworthy/secrets/passphrases/ubuntu_crypt_password.yaml (name hardcoded with "ubuntu" reference)
  - site/airship-seaworthy/software/charts/ucp/divingbell/divingbell.yaml (user name is hardcoded "ubuntu")
  - tools/updater.py
- Add CI gate for openSUSE, CentOS, etc.
  - tools/gate/Jenkinsfile

Security impact

Do not expect any material change in security controls and/or policies.

SUSE plans to adopt the Airship AppArmor profile in the Treasuremap project.

Performance impact

Do not expect performance impact.

Alternatives

None. Extending Linux distro support is critical for Airship to expand its user base and for its developer community to grow.

Implementation

We propose three milestones to develop the feature in an iterative approach.

Milestone 1: Multi Linux distro support in the bring your own K8s and Ceph use case. The work in this milestone is to bring Armada, Shipyard, Deckhand and Pegleg to Linux distro agnostic, and support Ubuntu and openSUSE as the two available options, and CentOS if there are developers familiar with CentOS join the effort.

Milestone 2: Add the ability in bootstrapping to plug in the KubeAdm and Ceph release/packages built for the underlying Linux distros on the existing Physical hosts. The work is focused on Promenade component.

Milestone 3: Add the ability in Drydock to provision baremetal on Linux distros in addition to Ubuntu.

Assignee(s):

SUSE is committed to implement this spec, add the openSUSE plugins and gate tests, and welcomes the community to join the effort.

Dependencies

OpenStack Helm

1. Add the openSUSE base OS option in the OSH tool images, including cepf-config-helper, libvirt, OpenVSwitch, tempest, vbmc.
2. Add the ability to specify OS choice in loci.sh and support Ubuntu, openSUSE, CentOS etc.

LOCI

1. Add openSUSE base OS option in all OpenStack service images in LOCI.

Airship

1. Bring your own K8s and Ceph storage. Link TBD
2. Add Ironic driver in Drydock. Link TBD

References

Any external references (other than the direct links above)

3.1.3 Spyglass

Spyglass is a data extraction tool which can interface with different input data sources to generate site manifest YAML files. The data sources will provide all the configuration data needed for a site deployment. These site manifest YAML files generated by spyglass will be saved in a Git repository, from where Pegleg can access and aggregate them. This aggregated file can then be fed to Shipyard for site deployment / updates.

Problem description

During the deployment of Airship Genesis node via Pegleg, it expects that the deployment engineer provides all the information pertained to Genesis, Controller & Compute nodes such as PXE IPs, VLANs pertaining to Storage network, Kubernetes network, Storage disks, Host profiles, etc. as manifests/YAMLs that are easily understandable by Pegleg. Currently there exists multiple data sources and these inputs are processed manually by deployment engineers. Considering the fact that there are multiple sites for which we need to generate such data, the current process is cumbersome, error-prone and time-intensive.

The solution to this problem is to automate the overall process so that the resultant work-flow has standardized operations to handle multiple data sources and generate site YAMLs considering site type and version.

Impacted components

None.

Proposed change

Proposal here is to develop a standalone stateless automation utility to extract relevant information from a given site data source and process it against site specific templates to generate site manifests which can be consumed by Pegleg. The data sources could be different engineering packages or extracted from remote external sources. One example of a remote data source can be an API endpoint.

The application shall perform the automation in two stages. In the first stage it shall generate a standardized intermediary YAML object after parsing extracted information from the data source. In the second stage the intermediary YAML shall be processed by a site processor using site specific templates to generate site manifests.

Overall Architecture

```
     +-----------+           +-------------+
     |           |           |  +-------+  |
     |           |   +------>|  |Generic|  |
 +-----------+   |   |       |  |Object |  |
 |Tugboat(Xl)| I |   |       |  +-------+  |
 |Plugin     | N |   |       |     |       |
 +-----------+ T |   |       |     |       |
     |         E |   |       |  +------+   |
+------------+ R |   |       |  |Parser|   +------> Intermediary YAML
|Remote Data | F |---+       |  +------+   |
|SourcePlugin| A |           |     |       |
+------------+ C |           |     |(Intermediary YAML)
     |         E |           |     |       |
     |           |           |     |       |
     |         H |           |     v       |
     |         A |           |  +---------+|(templates)    +------------+
     |         N |           |  |Site     |+<--------------|Repository  |
     |         D |           |  |Processor||-------------->|Adapter     |
     |         L |           |  +---------+|(Generated     +------------+
     |         E |           |      ^      | Site Manifests)
     |         R |           |  +---|-----+|
     |           |           |  |  J2     ||
     |           |           |  |Templates||
     |           |           |  +---------+|
     +-----------+           +-------------+
```

_

1)Interface handler: Acts as an interface to support multiple plugins like Excel, Remote Data Source, etc. The interface would define abstract APIs which would be overridden by different plugins. A plugin would implement these APIs based on the type of data source to collect raw site data and convert them to a generic object for further processing. For example: Consider the APIs connect_data_source() and get_host_profile(). For Excel plugin the connect_data_source API would implement file-open methods and the get_host_profile would extract host profile related information from the Excel file.

In the case of a remote data source (for example an API endpoint), the API "connect_data_source" shall authenticate (if required) and establish a connection to the remote site and the "get_host_profile" API shall implement the logic to extract appropriate details over the established connection. In order to support future plugins, one needs to override these interface handler APIs and develop logic to extract site data from the corresponding data source.

2)Parser: It processes the information obtained from generic YAML object to create an intermediary YAML using the following inputs: a) Global Design Rules: Common rules for generating manifest for any kind of site. These rule are used for every plugin. for example: IPs to skip before considering allocation to host. b) Site Config Rules: These are settings specific to a particular site. For example http_proxy, bgp asn number, etc. It can be referred by all plugins. Sometimes these site specific information can also be received from plugin data sources. In such cases the information from plugin data sources would be used instead of the ones specified in site config rules.

3)Intermediary YAML: It holds the complete site information after getting it from interface handler plugin and after application of site specific rules. It maintains a common format agnostic of the corresponding data source used. So it act as a primary input to Site Processor for generating site manifests.

4)Tugboat(Excel Parser) Plugin: It uses the interface handler APIs to open and parse the Excel file to extract site details and create an in memory generic YAML object. This generic object is further processed using site specific config rules and global rules to generate an intermediary YAML. The name “Tugboat” here is used to identify “Excel Parser”. For Excel parser the plugin shall use a Site specification file which defines the various location(s) of the site information items in file. The location is specified by mentioning rows and columns of the spreadsheet cell containing the specific site data.

5)Remote Data Source Plugin: It uses the interface handler APIs to connect to the data source and extract site specific information and then construct a generic in memory YAML object. This object is then parsed to generate an intermediary YAML. There may be situations wherein the information extracted from API endpoints are incomplete. In such scenarios, the missing information can be supplied from Site Config Rules.

6)Site Processor: The site processor consumes the intermediary YAML and generates site manifests based on corresponding site templates that are written in python Jinja2. For example, for template file “baremetal.yaml.j2”, the site processor will generate “baremetal.yaml” with the information obtained from intermediary YAML and also by following the syntax present in the corresponding template file.

7)Site Templates(J2 templates): These define the manifest file formats for various entities like baremetal, network, host-profiles, etc. The site processor applies these templates to an intermediary YAML and generates the corresponding site manifests. For example: calico-ip-rules.yaml.j2 will generate calico-ip-rules.yaml when processed by the site processor.

8)Repository Adapter: This helps in importing site specific templates from a repository and also push generated site manifest YAMLs. The aim of the repository adapter shall be to abstract the specific repository operations and maintain an uniform interface irrespective of the type of repository used. It shall be possible to add newer repositories in the future without any change to this interface. The access to this repository can be regulated by credentials if required and those will be passed as parameters to the site specific config file.

9)Sample data flow: for example generating OAM network information from site manifests.

- Raw rack information from plugin:

```
vlan_network_data:
    oam:
        subnet: 12.0.0.64/26
        vlan: '1321'
```

- Rules to define gateway, ip ranges from subnet:

```
rule_ip_alloc_offset:
    name: ip_alloc_offset
        ip_alloc_offset:
            default: 10
            gateway: 1
```

The above rule specify the ip offset to considered to define ip address for gateway, reserved and static ip ranges from the subnet pool. So ip range for 12.0.0.64/26 is : 12.0.0.65 ~ 12.0.0.126 The rule “ip_alloc_offset” now helps to define additional information as follows:

- gateway: 12.0.0.65 (the first offset as defined by the field ‘gateway’)
- reserved ip ranges: 12.0.0.65 ~ 12.0.0.76 (the range is defined by adding “default” to start ip range)
- static ip ranges: 12.0.0.77 ~ 12.0.0.126 (it follows the rule that we need to skip first 10 ip addresses as defined by “default”)

• Intermediary YAML file information generated after applying the above rules to the raw rack information:

```
network:
     vlan_network_data:
        oam:
         network: 12.0.0.64/26
         gateway: 12.0.0.65 --------+
         reserved_start: 12.0.0.65  |
         reserved_end: 12.0.0.76    |
         routes:                    +--> Newly derived information
          - 0.0.0.0/0               |
         static_start: 12.0.0.77    |
         static_end: 12.0.0.126 ----+
         vlan: '1321'
```

–

• J2 templates for specifying oam network data: It represents the format in which the site manifests will be generated with values obtained from Intermediary YAML

```
---
schema: 'drydock/Network/v1'
metadata:
  schema: 'metadata/Document/v1'
  name: oam
  layeringDefinition:
    abstract: false
    layer: 'site'
    parentSelector:
      network_role: oam
      topology: cruiser
    actions:
      - method: merge
        path: .
  storagePolicy: cleartext
data:
  cidr: {{ data['network']['vlan_network_data']['oam']['network'] }}}
  routes:
    - subnet: {{ data['network']['vlan_network_data']['oam']['routes'] }}
      gateway: {{ data['network']['vlan_network_data']['oam']['gateway'] }}
      metric: 100
    ranges:
    - type: reserved
      start: {{ data['network']['vlan_network_data']['oam']['reserved_start
↪'] }}
      end: {{ data['network']['vlan_network_data']['oam']['reserved_end'] }}
```

(continues on next page)

(continued from previous page)

```
    - type: static
      start: {{ data['network']['vlan_network_data']['oam']['static_start'] }
↪}
      end: {{ data['network']['vlan_network_data']['oam']['static_end'] }}
...
```

–

- OAM Network information in site manifests after applying intermediary YAML to J2 templates.:

```
---
schema: 'drydock/Network/v1'
metadata:
  schema: 'metadata/Document/v1'
  name: oam
  layeringDefinition:
    abstract: false
    layer: 'site'
    parentSelector:
      network_role: oam
      topology: cruiser
    actions:
      - method: merge
        path: .
  storagePolicy: cleartext
data:
  cidr: 12.0.0.64/26
  routes:
    - subnet: 0.0.0.0/0
      gateway: 12.0.0.65
      metric: 100
  ranges:
    - type: reserved
      start: 12.0.0.65
      end: 12.0.0.76
    - type: static
      start: 12.0.0.77
      end: 12.0.0.126
...
```

–

Security impact

The impact would be limited to the use of credentials for accessing the data source, templates and also for uploading generated manifest files.

Performance impact

None.

Alternatives

No existing utilities available to transform site information automatically.

Implementation

The following high-level implementation tasks are identified: a) Interface Handler b) Plugins (Excel and a sample Remote data source plugin) c) Parser d) Site Processor e) Repository Adapter

Usage

The tool will support Excel and remote data source plugin from the beginning. The section below lists the required input files for each of the aforementioned plugins.

- Preparation: The preparation steps differ based on selected data source.
  1. Excel Based Data Source.
     - Gather the following input files:
       (a) Excel based site Engineering package. This file contains detail specification covering IPMI, Public IPs, Private IPs, VLAN, Site Details, etc.
       (b) Excel Specification to aid parsing of the above Excel file. It contains details about specific rows and columns in various sheet which contain the necessary information to build site manifests.
       (c) Site specific configuration file containing additional configuration like proxy, bgp information, interface names, etc.
       (d) Intermediary YAML file. In this cases Site Engineering Package and Excel specification are not required.
  2. Remote Data Source
     - Gather the following input information:
       (a) End point configuration file containing credentials to enable its access. Each end-point type shall have their access governed by their respective plugins and associated configuration file.
       (b) Site specific configuration file containing additional configuration like proxy, bgp information, interface names, etc. These will be used if information extracted from remote site is insufficient.
- Program execution
  1. CLI Options:

-g, –generate_intermediary	Dump intermediary file from passed Excel and Excel spec.
-m, –generate_manifests	Generate manifests from the generated intermediary file.
-x, –excel PATH	Path to engineering Excel file, to be passed with generate_intermediary. The -s option is mandatory with this option. Multiple engineering files can be used. For example: -x file1.xls -x file2.xls
-s, –exel_spec PATH	Path to Excel spec, to be passed with generate_intermediary. The -x option is mandatory along with this option.
-i, –intermediary PATH	Path to intermediary file,to be passed with generate_manifests. The -g and -x options are not required with this option.
-d, –site_config PATH	Path to the site specific YAML file [required]
-l, –loglevel INTEGER	Loglevel NOTSET:0 ,DEBUG:10, INFO:20, WARNING:30, ERROR:40, CRITICAL:50 [default:20]
-e, –end_point_config	File containing end-point configurations like user-name password, certificates, URL, etc.
–help	Show this message and exit.

2. Example:

1. Using Excel spec as input data source:

   Generate Intermediary: `spyglass -g -x <DesignSpec> -s <excel spec> -d <site-config>`

   Generate Manifest & Intermediary: `spyglass -mg -x <DesignSpec> -s <excel spec> -d <site-config>`

   Generate Manifest with Intermediary: `spyglass -m -i <intermediary>`

2. Using external data source as input:

   Generate Manifest and Intermediary: `spyglass -m -g -e<end_point_config> -d <site-config>`

   Generate Manifest: `spyglass -m -e<end_point_config> -d <site-config>`

Note: The end_point_config shall include attributes of the external data source that are necessary for its access. Each external data source type shall have its own plugin to configure its corresponding credentials.

- Program output:

  1. Site Manifests: As an initial release, the program shall output manifest files for "airship-seaworthy" site. For example: baremetal, deployment, networks, pki, etc. Reference: https://github.com/openstack/airship-treasuremap/tree/master/site/airship-seaworthy

  2. Intermediary YAML: Containing aggregated site information generated from data sources that is used to generate the above site manifests.

Future Work

1. Schema based manifest generation instead of Jinja2 templates. It shall be possible to cleanly transition to this schema based generation keeping a unique mapping between schema and generated manifests. Currently this is managed by considering a mapping of j2 templates with schemas and site type.
2. UI editor for intermediary YAML

Alternatives

1. Schema based manifest generation instead of Jinja2 templates.
2. Develop the data source plugins as an extension to Pegleg.

Dependencies

1. Availability of a repository to store Jinja2 templates.
2. Availability of a repository to store generated manifests.

References

None

3.1.4 Divingbell Ansible Framework

Ansible playbooks to achieve tasks for making bare metal changes for Divingbell target use cases.

Links

The work to author and implement this spec will be tracked under this Storyboard Story

Problem description

Divingbell uses DaemonSets and complex shell scripting to make bare metal changes. This raises 2 problems: - Increasing number of DaemonSets on each host with increasing Divingbell usecases - Reinventing the wheel by writing complex shell scripting logic to make bare metal changes.

Impacted components

The following Airship components will be impacted by this solution:

1. Divingbell: Introducing Ansible framework to make bare metal changes

Proposed change

This spec intends to introduce Ansible framework within Divingbell which is much simpler to make any bare metal configuration changes as compared to existing approach of writing complex shell scripting to achieve the same functionality.

Adding playbook

Ansible playbooks should be written for making any configuration changes on the host.

Existing shell script logic for making bare metal changes lives under `divingbell/templates/bin`, wherever applicable these should be replaced by newly written Ansible playbooks as described in the sections below. Ansible playbooks would be part of the Divingbell image.

A separate directory structure needs to be created for adding the playbooks. Each Divingbell config can be a separate role within the playbook structure.

```
- playbooks/
    - roles/
         - systcl/
         - limits/
    - group_vars
         - all
    - master.yml
```

Files under `group_vars` should be loaded as a Kubernetes `ConfigMap` or `Secret` inside the container. Existing entries in `values.yaml` for Divingbell should be used for populating the entries in the file under `group_vars`.

This PS Initial commit for Ansible framework should be used as a reference PS for implementing the Ansibile framework.

Ansible Host

With Divingbell DaemonSet running on each host mounted at `hostPath`, `hosts` should be defined as given below within the `master.yml`.

```
hosts: all
connection: chroot
```

Ansible chroot plugin should be used for making host level changes. *Ansible chroot plugin_*

Divingbell Image

Dockerfile should be created containing the below steps:

- Pull base image
- Install Ansible
- Define working directory
- Copy the playbooks to the working directory

Divingbell DaemonSets

All the Divingbell DaemonSets that follow declarative and idempotent models should be replaced with a single DaemonSet. This DaemonSet will be responsible for populating required entries in `group_vars` as `volumeMounts`. Ansible command to run the playbook should be invoked from within the `DaemonSet` spec.

The Ansible command to run the playbook should be invoked from within the `DaemonSet` spec.

The Divingbell DaemonSet for `exec` module should be left out from this framework and it should keep functioning as a separate DaemonSet.

Ansible Rollback

Rollback should be achieved via the `update_site` action i.e. if a playbook introduces a bad change into the environment then the recovery path would be to correct the change in the playbooks and run `update_site` with new changes.

Security impact

None - No new security impacts are introduced with this design.

Performance impact

As this design reduces the number of DaemonSets being used within Divingbell, it will be an improvement in performance.

Implementation

This implementation should start off as a separate entity and not make parallel changes by removing the existing functonality.

New Divingbell usecases can be first targetted with the Ansible framework while existing framework can co-exist with the new framework.

Dependencies

Adds new dependency - Ansible framework.

References

3.1.5 Introduce Redfish based OOB Driver for Drydock

Proposal to support new OOB type Redfish as OOB driver for Drydock. Redfish is new standard for Platform management driven by DMTF.

Links

https://storyboard.openstack.org/#!/story/2003007

Problem description

In the current implementation, Drydock supports the following OOB types

1. IPMI via pyhgmi driver to manage baremetal servers
2. Libvirt driver to manage Virtual machines
3. Manual driver

Phygmi is python implementation for IPMI functionality. Currently phygmi supports few commands related to power on/off, boot, events and Lenovo OEM functions. Introducing a new IPMI command in pyghmi is complex and requires to know the low level details of the functionality like Network Function, Command and the data bits to be sent.

DMTF's have proposed a new Standard Platform management API Redfish using a data model representation inside of hypermedia RESTful interface. Vendors like Dell, HP supports Redfish and Rest API are exposed to perform any actions. Being a REST and model based standard makes it easy for external tools like Drydock to communicate with the Redfish server.

Impacted components

The following Airship components would be impacted by this solution:

1. Drydock - new OOB driver Redfish

Proposed change

Proposal is to add new OOB driver that supports all Drydock Orchestrator actions and configure the node as per the action. The communication between the driver and node will be REST based on Redfish resources exposed by the node. There shall be no changes in the way driver creates tasks using Orchestrator, exception handling and the concurrent execution of tasks.

Redfish driver

Adding a new OOB driver requires to extend the base driver `drydock_provisioner.drivers.driver.OobDriver`.

OOB type will be named as:

```
oob_types_supported = ['redfish']
```

All the existing Orchestrator OOB actions need to be supported. New Action classes will be created for each of the OOB action and uses Redfish client to configure the node.:

```
action_class_map = {
    hd_fields.OrchestratorAction.ValidateOobServices: ValidateOobServices,
    hd_fields.OrchestratorAction.ConfigNodePxe: ConfigNodePxe,
    hd_fields.OrchestratorAction.SetNodeBoot: SetNodeBoot,
    hd_fields.OrchestratorAction.PowerOffNode: PowerOffNode,
    hd_fields.OrchestratorAction.PowerOnNode: PowerOnNode,
    hd_fields.OrchestratorAction.PowerCycleNode: PowerCycleNode,
    hd_fields.OrchestratorAction.InterrogateOob: InterrogateOob,
}
```

Implement Action classes

Action class have to extend the base action `drydock_provisioner.orchestrator.actions.orchestrator.BaseAction`. The actions are executed as threads and so each action class have to implement the start method.

Below is the table that mentions the OOB action and the corresponding Redfish commands. Details of each redfish command in terms of Redfish API is specified in the next section.

Table 1: Drydock Actions and redfish commands

Action	Redfish Commands
ValidateOobServices	Not implemented
ConfigNodePxe	Not implemented
SetNodeBoot	set_bootdev, get_bootdev
PowerOffNode	set_power, get_power
PowerOnNode	set_power, get_power
PowerCycleNode	set_power, get_power
InterrogateOob	get_power

No configuration is required for the actions ValidateOobServices, ConfigNodePxe.

Redfish client

Above mentioned commands (set_bootdev, get_bootdev, set_power, get_power) will be implemented by new class RedfishObject. This class is responsible for converting the commands to corresponding REST API and call the open-source python implementations of redfish clients. python-redfish-library provided by DMTF is chosen as Redfish client.

In addition, there will be Redfish API extensions related to OEM which will be specific to vendor. Based on the need, the RedfishObject have to handle them and provide a clean interface to OOB actions.

The redfish REST API calls for the commands:

```
Command:   get_bootdev
Request:   GET https://<OOB IP>/redfish/v1/Systems/<System_name>/
Response:  dict["Boot"]["BootSourceOverrideTarget"]

Command:   set_bootdev
Request:   PATCH https://<OOB IP>/redfish/v1/Systems/<System_name>/
           {"Boot": {
               "BootSourceOverrideEnabled": "Once",
               "BootSourceOverrideTarget": "Pxe",
           }}

Command:   get_power
Request:   GET https://<OOB IP>/redfish/v1/Systems/<System_name>/
Response:  dict["PowerState"]

Command:   set_power
Request:   POST https://<OOB IP>/redfish/v1/Systems/<System_name>/Actions/
↪ComputerSystem.Reset
           {
               "ResetType": powerstate
           }
           Allowed powerstate values are "On", "ForceOff", "PushPowerButton",
↪"GracefulRestart"
```

Configuration changes

OOB driver that will be triggered by Drydock orchestrator is determined by

- availability of driver class in configuration parameter oob_driver under [plugins] section in drydock.conf

- OOB type specified in HostProfile in Site manifests

To use the Redfish driver as OOB, the OOB type in Host profile need to be set as `redfish` and a new entry to be added for oob_driver in drydock.conf `drydock_provisioner.drivers.oob.redfish_driver.RedfishDriver`

Sample Host profile with OOB type redfish:

```
---
schema: drydock/HostProfile/v1
metadata:
  schema: metadata/Document/v1
  name: global
  storagePolicy: cleartext
  labels:
    hosttype: global_hostprofile
  layeringDefinition:
    abstract: true
    layer: global
data:
  oob:
    type: 'redfish'
    network: 'oob'
    account: 'tier4'
    credential: 'cred'
```

Security impact

None

Performance impact

None

Implementation

Work Items

- Add redfish driver to drydock configuration parameter `oob_driver`
- Add base Redfish driver derived from oob_driver.OobDriver with oob_types_supported *redfish*
- Add RedfishObject class that uses python redfish library to talk with the node.
- Add OOB action classes specified in Proposed change
- Add related tests - unit test cases

Assignee(s)

Primary assignee: Hemanth Nakkina

Other contributors: PradeepKumar KS Gurpreet Singh

Dependencies

None

References

3.1.6 Drydock: Support BIOS configuration using Redfish OOB driver

Proposal to add support for configuring BIOS settings of baremetal node via Drydock. This blueprint is intended to extend functionality of redfish OOB driver to support BIOS configuration.

Links

https://storyboard.openstack.org/#!/story/2002912

Problem description

Currently drydock does not provide a mechanism to configure BIOS settings on a baremetal node. The BIOS settings need to be configured manually prior to triggering deployment via Airship.

Impacted components

The following Airship components would be impacted by this solution:

1. Drydock - Updates to Orchestrator actions and Redfish OOB driver

Proposed change

The idea is to provide user an option to specify the BIOS configuration of baremetal nodes as part of HardwareProfile yaml in site definition documents. Drydock gets this information from manifest documents and whenever Orchestrator action PrepareNodes is triggered, drydock initiates BIOS configuration via OOB drivers. As there are no new Orchestrator actions introduced, the workflow from Shipyard –> Drydock remains the same.

This spec only supports BIOS configuration via Redfish OOB driver. Documents having BIOS configuration with oob type other than Redfish (ipmi, libvirt) should result in an error during document validation. This can be achieved by adding new Validator in Drydock.

Manifest changes

A new parameter `bios_settings` will be added to the HardwareProfile. The parameter takes a dictionary of strings as its value. Each key/value pair corresponds to a BIOS setting that need to be configured. This provides the deployment engineers the flexibility to modify the BIOS settings that need to be configured on baremetal node.

Sample HardwareProfile with bios_settings:

```
---
schema: 'drydock/HardwareProfile/v1'
metadata:
  schema: 'metadata/Document/v1'
  name: dell_r640_test
  storagePolicy: 'cleartext'
```

(continues on next page)

(continued from previous page)

```
  layeringDefinition:
    abstract: false
    layer: global
data:
  vendor: 'Dell'
  generation: '8'
  hw_version: '3'
  bios_version: '2.2.3'
  boot_mode: bios
  bootstrap_protocol: pxe
  pxe_interface: 0
  bios_settings:
    BootMode: Bios
    BootSeqRetry: Disabled
    InternalUsb: Off
    SriovGlobalEnable: Disabled
    SysProfile: PerfOptimized
    AcPwrRcvry: Last
    AcPwrRcvryDelay: Immediate
  device_aliases:
    pxe_nic01:
      # eno3
      address: '0000:01:00.0'
      dev_type: 'Gig NIC'
      bus_type: 'pci'
  cpu_sets:
    kvm: '4-43,48-87'
  hugepages:
    dpdk:
      size: '1G'
      count: 300
```

Update the HardwareProfile schema to include a new property `bios_settings` of type object. The property should be optional to support backward compatibility.

Following will be added as part of HardwareProfile schema properties:

```
bios_settings:
  type: 'object'
```

Redfish driver updates

Following OOB driver actions are introduced as part of this spec.

1. hd_fields.OrchestratorAction.ConfigBIOS To configure the BIOS settings on the node based on HardwareProfile manifest document

To support the above actions, following redfish commands will be added - set_bios_settings, get_bios_settings

Redfish rest api calls to handle the above commands:

```
Command:   get_bios_settings
Request:   GET https://<OOB IP>/redfish/v1/Systems/<System_name>/Bios
Response:  dict["Attributes"]

Command:   set_bios_settings
```

(continues on next page)

(continued from previous page)

```
Request:    PATCH https://<OOB IP>/redfish/v1/Systems/<System_name>/Bios/Settings
            { "Attributes": {
                "setting1": "value1",
                "setting2": "value2"
            }}
```

The request and response objects for the above operations differ for vendors HP and Dell. Above mentioned request/response objects are for Dell. In case of HP the request/response object will be:

```
{
    "setting1": "value1",
    "setting2": "value2"
}
```

In case of failures in setting BIOS configuration, the Redfish server sends the error message along with error code. The ConfigBios action should mark the task as failure and add the error message in the task status message.

Orchestrator action updates

PrepareNodes Action currently run the following driver actions in sequence

1. hd_fields.OrchestratorAction.SetNodeBoot on OOB driver To set the boot mode to PXE
2. hd_fields.OrchestratorAction.PowerCycleNode on OOB driver To powercycle the node
3. hd_fields.OrchestratorAction.IdentifyNode on Node driver To identify the node in node driver like maas

PrepareNodes should execute the new OOB driver action as its initial step `hd_fields.OrchestratorAction.ConfigBIOS`. PrepareNodes creates subtasks to execute ConfigBios action for each node and collects the subtask status until drydock timeout `conf.timeouts.drydock_timeout`. In case of any failure of ConfigBios subtask for a node, further driver actions wont be executed for that node. This is in sync with the existing design and no changes required. ConfigBios action is not retried in case of failures.

Security impact

None

Performance impact

BIOS configuration update takes around 35 seconds when invoked from a node on same rack. This includes establishing a session, running the configuration API and logging out the session. Time for system restart is not included. Similarly retrieving BIOS configuration takes around 18 seconds.

Alternatives

This spec only implements BIOS configuration support for Redfish OOB driver.

Implementation

Work Items

- Update Hardware profile schema to support new attribute bios_setting
- Update Hardware profile objects
- Update Orchestrator action PrepareNodes to call OOB driver for BIOS configuration
- Update Redfish OOB driver to support new action ConfigBIOS
- Add unit test cases

Assignee(s):

Primary Assignee: Hemanth Nakkina

Other contributors: Gurpreet Singh

Dependencies

This spec depends on Introduce Redfish based OOB Driver for Drydock story.

References

3.1.7 Deploy Kubernetes API Server w/ Ingress and Keystone Webhook

OpenStack Keystone will the be single authentication mechanism for Airship users. As such, we need to deploy a Kubernetes API server endpoint that can utilize Keystone for authentication and authorization. The avenue to support this is using the Kubernetes webhook admission controller with a webhook supporting Keystone.

Links

None

Problem description

While Airship component APIs should care for most lifecycle tasks for the Kubernetes cluster, there will be some maintenance and recovery operations that will require direct access to the Kubernetes API. To properly secure this API, it needs to utilize the common single sign-on that operators use for accessing Airship APIs, i.e. Keystone. However, the external facing API should minimize risk to the core Kubernetes API servers used by other Kubernetes core components. This specification proposes a design to maximize the security of this external facing API endpoint and minimizes the risk to the core operations of the cluster by avoiding the need to add complexity to core apiserver configuration or sending extra traffic through the core apiservers and Keystone.

Impacted components

The following Airship components would be impacted by this solution:

1. Promenade - Maintenance of the chart for external facing Kubernetes API servers

Proposed change

Create a chart, `webhook_apiserver`, for an external facing Kubernetes API server that would create a Kubernetes Ingress entrypoint for the API server and, optionally, also spin up a webhook side-car for each API server (i.e. `sidecar` mode). The other mode of operation is `federated` mode where the webhook will be accessed over a Kubernetes service.

A new chart is needed because the *standard apiserver chart <https://github.com/openstack/airship-promenade/tree/master/charts/apiserver>* relies on the anchor pattern creating static pods. The `webhook_apiserver` chart should be based on the standard apiserver chart and use helm_toolkit standards.

The chart would provide for configuration of the Keystone webhook (also Keystone webhook addl and Keystone webhook chart) in `sidecar` mode and allow for configuring the webhook service address in `federated`` mode. The Kubernetes apiserver would be configured to only allow for authentication/authorization via webhook. No other authorization modes would be enabled. All `kube-apiserver` command line options should match the with the following exceptions:

- authorization-mode: `Webhook`
- audit-log-path: `-`
- authentication-token-webhook-config-file: path to configuration file for accessing the webhook.
- authorization-webhook-config-file: path to configuration file for accessing the webhook.
- apiserver-count: omit
- endpoint-reconciler-type: `none`

Webhook Configuration

The configuration for how the Kubernetes API server will contact the webhook service is stored in a YAML configuration file based on the kubeconfig file format. The below example would be used in `sidecar` mode.

```
# clusters refers to the remote service.
clusters:
  - name: keystone-webhook
    cluster:
      # CA for verifying the remote service.
      certificate-authority: /path/to/webhook_ca.pem
      # URL of remote service to query. Must use 'https'. May not include parameters.
      server: https://localhost:4443/

# users refers to the API Server's webhook configuration.
users:
  - name: external-facing-api
    user:
      client-certificate: /path/to/apiserver_webhook_cert.pem # cert for the webhook plugin to use
      client-key: /path/to/apiserver_webhook_key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API Server.
current-context: webhook
contexts:
- context:
    cluster: keystone-webhook
    user: external-facing-api
  name: webhook
```

Documentation impact

Documentation of the overrides to this chart for controlling webhook authorization mapping policy.

Security impact

- Additional TLS certificates for apiserver <-> webhook connections
- Keystone webhook must have an admin-level Keystone account
- Optionally, the Keystone webhook minimizes attack surface by becoming a sidecar without external facing service.

Performance impact

This should not have any performance impacts as the only traffic handled by the webhook will be from users specifically using Keystone for authentication and authorization.

Testing impact

The chart should include a Helm test that validates a valid Keystone token is usable with `kubectl` to successfully get a respond from the Kubernetes API.

Implementation

Milestone 1

Chart support for `sidecar` mode

Milestone 2

Addition of `federated` mode

Dependencies

None

References

3.1.8 Airship workflow to update Kubernetes node labels

Proposal to enhance Airship to support updating Kubernetes node labels as a triggered workflow using Shipyard as an entrypoint, Deckhand as a document repository, Drydock as the decision maker about application of node labels, and Promenade as the interactive layer to Kubernetes.

Links

None

Problem description

Over the lifecycle of a deployed site, there is a need to maintain the labels applied to Kubernetes nodes. Prior to this change the only Airship-supplied mechanism for this was during a node's deployment. Effectively, the way to change or remove labels from a deployed node is through a manual process. Airship aims to eliminate or minimize manual action on a deploy site.

Without the ability to declaratively update the labels for a Kubernetes node, the engineers responsible for a site lose finer-grained ability to adjust where deployed software runs – i.e. node affinity/anti-affinity. While the software's Helm or Armada chart could be adjusted and the site updated, the granularity of marking a single node with a label is still missed.

Impacted components

The following Airship components would be impacted by this solution:

1. Drydock - endpoint(s) to evaluate and trigger adding or removing labels on a node
2. Promenade - endpoint(s) to add/remove labels on a node.
3. Shipyard - new workflow: update_labels

Proposed change

Note: External to Airship, the process requires updating the site definition documents describing Baremetal Nodes to properly reflect the desired labels for a node. The workflow proposed below does not allow for addition or elimination of node labels without going through an update of the site definition documents.

Shipyard

To achieve the goal of fine-grained declarative Kubernetes label management, a new Shipyard action will be introduced: `update_labels`. The update_labels action will accept a list of targeted nodes as an action parameter. E.g.:

```
POST /v1.0/actions

{
  "name" : "action name",
  "parameters" : {
    "target_nodes": [ "node1", "node2"]
  }
}
```

The most recent committed configuration documents will be used to drive the labels that result on the target nodes.

- If there is no committed version of the configuration documents, the action will be rejected.
- If there are no revisions of the configuration documents marked as participating in a site action, the action will be rejected.

A new workflow will be invoked for `update_labels`, being passed the configuration documents revision and the target nodes as input, using existing parameter mechanisms.

Note: At the time of writing this blueprint, there are no other actions exposed by Shipyard that are focused on a set of target nodes instead of the whole site. This introduces a new category of `targeted` actions, as opposed to the existing `site` actions. Targeted actions should not set the site action labels (e.g. successful-site-action) on Deckhand revisions as part of the workflow.

The workflow will perform a standard validation of the configuration documents by the involved components (Deckhand, Drydock, Promenade).

Within the Shipyard codebase, a new Drydock operator will be defined to invoke and monitor the invocation of Drydock to trigger label updates. Using the task interface of Drydock, a node filter containing the target nodes will be used to limit the scope of the request to only those nodes, along with the design reference. E.g.:

```
POST /v1.0/tasks

{
  "action": "relabel_nodes",
  "design_ref": "<deckhand_uri>",
  "node_filter": {
    "filter_set_type": "union",
    "filter_set": [
      {
        "filter_type": "union",
        "node_names": ["node1", "node2"],
        "node_tags": [],
        "node_labels": {},
        "rack_names": [],
        "rack_labels": {},
      }
    ]
  }
}
```

Note: Since a node filter is part of this interface, it would technically allow for other ways to assign labels across nodes. However at this time, Shipyard will only leverage the node_names field.

After invoking Drydock (see below), the workflow step will use the top level Drydock task result, and disposition the step as failure if any nodes are unsuccessful. This may result in a partial update. No rollbacks will be performed.

Drydock

Drydock's task API will provide a new action `relabel_nodes`. This action will perform necessary analysis of the design to determine the full set of labels that should be applied to each node. Some labels are generated dynamically during node deployment; these will need to be generated and included in the set of node labels.

Since multiple nodes can be targeted, and success or failure may occur on each, Drydock will track these as subtasks and roll up success/failure per node to the top level task.

Promenade

For each node, Drydock will invoke Promenade to apply the set of labels to the Kubernetes node, using a `PUT` against the (new) `node-labels/{node_name}` endpoint. The payload of this request is a list of labels for that node. E.g.:

```
PUT /v1.0/node-labels/node1

{
  "label-a":"true",
  "label-n":"some-value"
}
```

Promenade will perform a difference of the existing node labels against the requested node labels. Promenade will then in order:

1. apply new labels and change existing labels with new values
2. remove labels that are not in the request body

API Clients and CLIs

The Drydock, Promenade, and Shipyard API Clients and CLI components will need to be updated to match the new functionality defined above.

Documentation impact

Each of the identified components have associated API (and CLI) documentation that will be updated to match the new API endpoints and associated payload formats as noted above.

Security impact

None - No new security impacts are introduced with this design. Existing mechanisms will be applied to the changes introduced.

Performance impact

None - This workflow has no specific performance implications for the components involved.

High level process

```
    Shipyard                Workflow                 Drydock                   ␣
↪Promenade
+---------------+        +-------------+
| Submit Action +------> |             |
| update_labels |        |             |
|               |        |Drydock Task:|       +------------------+
+---------------+        | relabel_node+-----> |Evaluate baremetal|
                         |             |       |definition;       |
                         |Monitor Task +-----> |generate k8s node |
```

(continues on next page)

(continued from previous page)

```
                         |             |  Poll |labels                 |
                         |             | <-----+                       |
                         |             |       |Promenade:             |         +----------
↪----------+
                         |             |       | PUT node-labels  +--------> |Diff␣
↪existing node |
                         |             |       |   (list of labels)|  Wait   | labels.␣
↪          |
                         |             |       |                   | <--------+ Add new␣
↪labels    |
                         |             |       +-------------------+         | Remove␣
↪orphaned   |
                         |             |                                     |  labels␣
↪          |
                         |             |                                     |        ␣
↪          |
                         |             |                                     +----------
↪----------+
                         |End workflow |
                         |             |
                         +-------------+
```

Implementation

There are no specific milestones identified for this blueprint.

https://review.openstack.org/#/c/584925/ is work that has started for Promenade.

Dependencies

None

References

3.1.9 Pegleg Secret Generation and Encryption

Pegleg is responsible for shepherding deployment manifest documents from their resting places in Git repositories to a consumable format that is ready for ingestion into Airship. This spec expands its responsibility to account for secure generation and encryption of secrets that are required within an Airship-based deployment.

Links

The work to author and implement this spec will be tracked under this Storyboard Story.

Problem description

Airship supports the ability to identify secret information required for functioning deployments, such as passwords and keys; to ingest it into the site in a least-privilege-oriented fashion; and to encrypt it at rest within Deckhand. However, lifecycle management of the secrets outside the site should be made automatable and repeatable, to facilitate operational needs such as periodic password rotation, and to ensure that unencrypted secrets are only accessible by authorized individuals.

Impacted components

The following Airship components will be impacted by this solution:

1. Pegleg: enhanced to generate, rotate, encrypt, and decrypt secrets.
2. Promenade: PKICatalog will move to Pegleg.
3. Treasuremap: update site manifests to use new Catalogs.
4. Airship-in-a-Bottle: update site manifests to use new Catalogs.

Proposed change

PeglegManagedDocument

With this spec, the role of Pegleg grows from being a custodian of deployment manifests to additionally being the author of certain manifests. A new YAML schema will be created to describe these documents: `pegleg/PeglegManagedDocument/v1`. Documents of this type will have one or both of the following data elements, although more may be added in the future: `generated`, `encrypted`. PeglegManagedDocuments serve as wrappers around other documents, and the wrapping serves to capture additional metadata that is necessary, but separate from the managed document proper. The managed document data will live in the `data.managedDocument` portion of a PeglegManagedDocument.

If a PeglegManagedDocument is `generated`, then its contents have been created by Pegleg, and it must include provenance information per this example:

```
schema: pegleg/PeglegManagedDocument/v1
metadata:
  name: matches-document-name
  schema: metadata/Document/v1
  labels:
    matching: wrapped-doc
  layeringDefinition:
    abstract: false
    # Pegleg will initially support generation at site level only
    layer: site
  storagePolicy: cleartext
data:
  generated:
    at: <timestamp>
    by: <author>
    specifiedBy:
      repo: <...>
      reference: <git ref-head or similar>
      path: <PKICatalog/PassphraseCatalog details>
  managedDocument:
    schema: <as appropriate for wrapped document>
    metadata:
      storagePolicy: encrypted
      schema: <as appropriate for wrapped document>
      <metadata from parent PeglegManagedDocument>
      <any other metadata as appropriate>
    data: <generated data>
```

If a PeglegManagedDocument is `encrypted`, then its contents have been encrypted by Pegleg, and it must include provenance information per this example:

```
schema: pegleg/PeglegManagedDocument/v1
metadata:
  name: matches-document-name
  schema: metadata/Document/v1
  labels:
    matching: wrapped-doc
  layeringDefinition:
    abstract: false
    layer: matching-wrapped-doc
  storagePolicy: cleartext
data:
  encrypted:
    at: <timestamp>
    by: <author>
  managedDocument:
    schema: <as appropriate for wrapped document>
    metadata:
      storagePolicy: encrypted
      schema: <as appropriate for wrapped document>
      <metadata from parent PeglegManagedDocument>
      <any other metadata as appropriate>
    data: <encrypted string blob>
```

A PeglegManagedDocument that is both generated via a Catalog, and encrypted (as specified by the catalog) will contain both `generated` and `encrypted` stanzas.

Note that this `encrypted` key has a different purpose than the Deckhand `storagePolicy:  encrypted` metadata, which indicates an *intent* for Deckhand to store a document encrypted at rest in the cluster. The two can be used together to ensure security, however: if a document is marked as `storagePolicy:  encrypted`, then automation may validate that it is only persisted (e.g. to a Git repository) if it is in fact encrypted within a PeglegManagedDocument.

Note also that the Deckhand `storagePolicy` of the PeglegManagedDocument itself is always `cleartext`, since its data stanza is not encrypted – it only wraps a document that *is* `storagePolicy:  encrypted`. This should be implemented as a Pegleg lint rule.

Document Generation

Document generation will follow the pattern established by Promenade's PKICatalog pattern. In fact, PKICatalog management responsibility will move to Pegleg as part of this effort. The types of documents that are expected to be generated are certificates and keys, which are defined via PKICatalog documents now, and passphrases, which will be defined via a new `pegleg/PassphraseCatalog/v1` document. Longer-term, these specifications may be combined, or split further (into a CertificateCatalog and KeypairCatalog), but this is not needed in the initial implementation in Pegleg. A collection of manifests may define more than one of each of these secret catalog documents if desired.

The documents generated via PKICatalog and PassphraseCatalog will follow the PeglegManagedDocument schema above; note that this is a change to existing PKICatalog behavior. The PKICatalog schema and associated code should be copied to Pegleg (and renamed to `pegleg/PKICatalog/v1`), and during a transition period the old and new PKICatalog implementations will exist side-by-side with slightly different semantics. Promenade's PKICatalog can be removed once all deployment manifests have been updated to use the new one.

Pegleg will place generated document files in `<site>/secrets/passphrases/`, `<site>/secrets/certificates`, or `<site>/secrets/keypairs` as appropriate:

- The generated filenames for passphrases will follow the pattern `<passphrase-doc-name>.yaml`.

- The generated filenames for certificate authorities will follow the pattern `<ca-name>_ca.yaml`.
- The generated filenames for certificates will follow the pattern `<ca-name>_<certificate-doc-name>_certificate.yaml`.
- The generated filenames for certificate keys will follow the pattern `<ca-name>_<certificate-doc-name>_key.yaml`.
- The generated filenames for keypairs will follow the pattern `<keypair-doc-name>.yaml`.
- Dashes in the document names will be converted to underscores for consistency.

A PassphraseCatalog will capture the following example structure:

```
schema: pegleg/PassphraseCatalog/v1
metadata:
  schema: metadata/Document/v1
  name: cluster-passphrases
  layeringDefinition:
    abstract: false
    layer: site
  storagePolicy: cleartext
data:
  passphrases:
    - document_name: osh-nova-password
      description: Service password for Nova
      encrypted: true
    - document_name: osh-nova-oslo-db-password
      description: Database password for Nova
      encrypted: true
      length: 12
```

The nonobvious bits of the document described above are:

- `encrypted` is optional, and denotes whether the generated PeglegManagedDocument will be `encrypted`, as well as whether the wrapped document will have `storagePolicy: encrypted` or `storagePolicy: cleartext` metadata. If absent, `encrypted` defaults to `true`.
- `document_name` is required, and is used to create the filename of the generated PeglegManagedDocument manifest, and the `metadata.name` of the wrapped `deckhand/Passphrase/v1` document. In both cases, Pegleg will replace dashes in the `document_name` with underscores.
- `length` is optional, and denotes the length in characters of the generated cleartext passphrase data. If absent, `length` defaults to `24`. Note that with this length and the selected character set there will be less than 8x10^48 probability of getting a new passphrase that is identical to the previous passphrase. This is sufficiently random to ensure no duplication of rotated passphrases in practice.
- `description` is optional.

The `encrypted` key will be added to the PKICatalog schema, and adds the same semantics to PKICatalog-based generation as are described above for PassphraseCatalog.

Pegleg CLI Changes

The Pegleg CLI interface will be extended as follows. These commands will create PeglegManagedDocument manifests in the local repository. Committing and pushing the changes will be left to the operator or to script-based automation.

For the CLI commands below which encrypt or decrypt secrets, an environment variable (e.g. `PEGLEG_PASSPHRASE` will be use to capture the master passphrase to use. `pegleg site secrets` `rotate` will use a second variable (e.g. `PEGLEG_PREVIOUS_PASSPHRASE`) to hold the key/passphrase being rotated out. The contents of these keys/passphrases are not generated by Pegleg, but are created externally and set by a deployment engineer or tooling. A configurable minimum length (default 24) for master passphrases will be checked by all CLI commands which use the passphrase. All other criteria around passphrase strength are assumed to be enforced elsewhere, as it is an external secret that is consumed/used by Pegleg.

`pegleg site secrets generate passphrases`: Generate passphrases according to all PassphraseCatalog documents in the site. Note that regenerating passphrases can be accomplished simply by re-running `pegleg site secrets generate passphrases`.

`pegleg generate passphrase`: A standalone version of passphrase generation. This generates a single passphrase based on the default length, character set, and implementation described above, and outputs it to the console. The PassphraseCatalog is not involved in this operation. This command is suitable for generation of a highly-secure Pegleg master passphrase.

`pegleg site secrets generate pki`: Generate certificates and keys according to all PKICatalog documents in the site. Note that regenerating certificates can be accomplished simply by re-running `pegleg site secrets generate pki`.

`pegleg site secrets generate`: Combines the two commands above. May be expanded in the future to include other manifest generation activities.

`pegleg site bootstrap`: For now, a synonym for `pegleg site secrets generate`, and may be expanded in the future to include other bootstrapping activities.

`pegleg site secrets encrypt`: Encrypt all site documents which have `metadata.storagePolicy: encrypted`, and which are not already encrypted within a wrapping PeglegManagedDocument. Note that the `pegleg site secrets generate` commands encrypt generated secrets as specified, so `pegleg site secrets encrypt` is intended mainly for external-facing secrets which a deployment engineer brings to the site manifests. The output PeglegManagedDocument will be written back to the filename that served as its source.

`pegleg site secrets decrypt <document YAML file>`: Decrypt a specific PeglegManagedDocument manifest, unwrapping it and outputting the cleartext original document YAML to standard output. This is intended to be used when an authorized deployment engineer needs to determine a particular cleartext secret for a specific operational purpose.

`pegleg site secrets rotate passphrases`: This action re-encrypts encrypted passphrases with a new key/passphrase, and it takes the previously-used key and a new key as input. It accomplishes its task via two activities:

- For encrypted passphrases that were imported from outside of Pegleg (i.e. PeglegManagedDocuments which lack the `generated` stanza), decrypt them with the old key (in-memory), re-encrypt them with the new key, and output the results.
- Perform a fresh `pegleg site secrets generate passphrases` process using the new key. This will replace all `generated` passphrases with new secret values for added security. There is an assumption here that the only actors that need to know generated secrets are the services within the Airship-managed cluster, not external services or deployment engineers, except perhaps for point-in-time troubleshooting or operational exercises.

Similar functionality for rotating certificates (which is expected to have a different cadence than passphrase rotation, typically) will be added in the future.

Driving deployment of a site directly via Pegleg is follow-on functionality which will collect site documents, use them to create the `genesis.sh` script, and then interact directly with Shipyard to drive deployments. Its details are beyond the scope of this spec, but when implemented, it should decrypt documents wrapped by applicable PeglegManagedDocuments at the lst responsible moment, and take care not to write, log, or stdout them to disk as cleartext.

Note that existing `pegleg collect` functionality should **not** be changed to decrypt encrypted secrets; this is because it writes its output to disk. If `pegleg collect` is called, at this point in time, the PeglegManagedDocuments will be written (encrypted) to disk. To enable special case full site secret decryption, a `--force-decrypt` flag will be added to `pegleg collect` to do this under controlled circumstances, and to help bridge the gap with existing CICD pipelines until Pegleg-driven site deployment is in place. It will leverage the `PEGLEG_PASSPHRASE` variable described above.

Secret Generation

The `rstr` library should be invoked to generate secrets of the appropriate length and character set. This library uses the `os.urandom()` function, which in turn leverages `/dev/urandom` on Linux, and it is suitable for cryptographic purposes.

Characters in generated secrets will be evenly distributed across lower- and upper-case letters, digits, and punctuation in !"#$%&'()*+,-./:;<=>?@[]^_`{|}~. Note this is equivalent to the union of Python string.ascii_letters, string.digits, and string.punctuation.

Secret Encryption

The Python `cryptography` library has been chosen to implement the encryption and decryption of secrets within Pegleg. `cryptography` aims to be the standard cryptographic approach for Python, and takes pains to make it difficult to do encryption poorly (via its `recipes` layer), while still allowing access to the algorithmic details when truly needed (via its `hazmat` layer). `cryptography` is actively maintained and is the target encryption library for OpenStack as well.

The `cryptography.fernet` module will be used for symmetric encryption. It uses AES with a 128-bit key for encryption, and HMAC using SHA256 for encryption.

Fernet requires as input a URL-safe, base64-encoded 32-byte encryption key, which will be derived from the master passphrase passed into Pegleg via `PEGLEG_PASSPHRASE` as described above. The example for password-based encryption from the Fernet documentation should be followed as a guide. The `salt` to be used in key derivation will be configurable, and will be set to a fixed value within a built Pegleg container via an environment variable passed into the Pegleg Dockerfile. This will allow the salt to be different on an operator-by-operator basis.

The `cryptography.exceptions.InvalidSignature` exception is thrown by `cryptography` when an attempt is made to decrypt a message with a key that is different than the one used to encrypt a message, i.e., when the user has supplied an incorrect phassphrase. It should be handled gracefully by Pegleg, resulting in an informative message back to the user.

Security impact

These changes will result in a system that handles site secrets in a highly secure manner, in the face of multiple roles and day 2 operational needs.

Performance impact

Performance impact to existing flows will be minimal. Pegleg will need to additionally decrypt secrets as part of site deployment, but this will be an efficient operation performed once per deployment.

Alternatives

The Python `secrets` library presents a convenient interface for generating random strings. However, it was introduced in Python 3.6, and it would be limiting to introduce this constraint on Airship CICD pipelines.

The `strgen` library presents an even more convenient interface for generating pseudo-random strings; however, it leverages the Python `random` library, which is unsuitably random for cryptographic purposes.

Deckhand already supports a `storagePolicy` element which indicates whether whether Deckhand will persist document data in an encrypted state, and this flag could have been re-used by Pegleg to indicate whether a secret is (or should be) encrypted. However, "should this data be encrypted" is a fundamentally different question than "is this data encrypted now", and additional metadata-esque parameters (`generated`, `generatedLength`) were desired as well, so this proposal adds `data.encrypted` to indicate the point-in-time encryption status. `storagePolicy` is still valuable in this context to make sure everything that *should* be encrypted *is*, prior to performing actions with it (e.g. Git commits).

The `PyCrypto` library is a popular solution for encryption in Python; however, it is no longer actively maintained. Following the lead of OpenStack and others, we opted instead for the `cryptography` library.

This proposed implementation writes the output of generation/encryption events back to the same source files from which the original data came. This is a destructive operation; however, it wasn't evident that it is problematic in any anticipated workflow. In addition, it sidesteps challenges around naming of generated files, and cleanup of original files.

Implementation

Please refer to the Storyboard Story for implementation planning information.

Dependencies

This work should be based on the patchset to add Git branch and revision support to Pegleg, if it is not merged by the time implementation begins. This patchset alters the CLI interface and Git repository management code, and basing on it will avoid future refactoring.

References

3.1.10 Airship Node Teardown

Shipyard is the entrypoint for Airship actions, including the need to redeploy a server. The first part of redeploying a server is the graceful teardown of the software running on the server; specifically Kubernetes and etcd are of critical concern. It is the duty of Shipyard to orchestrate the teardown of the server, followed by steps to deploy the desired new configuration. This design covers only the first portion - node teardown

Links

None

Problem description

When redeploying a physical host (server) using the Airship Platform, it is necessary to trigger a sequence of steps to prevent undesired behaviors when the server is redeployed. This blueprint intends to document the interaction that must occur between Airship components to teardown a server.

Impacted components

Drydock Promenade Shipyard

Proposed change

Shipyard node teardown Process

1. (Existing) Shipyard receives request to redeploy_server, specifying a target server.
2. (Existing) Shipyard performs preflight, design reference lookup, and validation steps.
3. (New) Shipyard invokes Promenade to decommission a node.
4. (New) Shipyard invokes Drydock to destroy the node - setting a node filter to restrict to a single server.
5. (New) Shipyard invokes Promenade to remove the node from the Kubernetes cluster.

Assumption: node_id is the hostname of the server, and is also the identifier that both Drydock and Promenade use to identify the appropriate parts - hosts and k8s nodes. This convention is set by the join script produced by promenade.

Drydock Destroy Node

The API/interface for destroy node already exists. The implementation within Drydock needs to be developed. This interface will need to accept both the specified node_id and the design_id to retrieve from Deckhand.

Using the provided node_id (hardware node), and the design_id, Drydock will reset the hardware to a re-provisionable state.

By default, all local storage should be wiped (per datacenter policy for wiping before re-use).

An option to allow for only the OS disk to be wiped should be supported, such that other local storage is left intact, and could be remounted without data loss. e.g.: –preserve-local-storage

The target node should be shut down.

The target node should be removed from the provisioner (e.g. MaaS)

Responses

The responses from this functionality should follow the pattern set by prepare nodes, and other Drydock functionality. The Drydock status responses used for all async invocations will be utilized for this functionality.

Promenade Decommission Node

Performs steps that will result in the specified node being cleanly disassociated from Kubernetes, and ready for the server to be destroyed. Users of the decommission node API should be aware of the long timeout values that may occur while awaiting promenade to complete the appropriate steps. At this time, Promenade is a stateless service and doesn't use any database storage. As such, requests to Promenade are synchronous.

```
POST /nodes/{node_id}/decommission

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

Such that the design reference body is the design indicated when the redeploy_server action is invoked through Shipyard.

Query Parameters:

- drain-node-timeout: A whole number timeout in seconds to be used for the drain node step (default: none). In the case of no value being provided, the drain node step will use its default.
- drain-node-grace-period: A whole number in seconds indicating the grace-period that will be provided to the drain node step. (default: none). If no value is specified, the drain node step will use its default.
- clear-labels-timeout: A whole number timeout in seconds to be used for the clear labels step. (default: none). If no value is specified, clear labels will use its own default.
- remove-etcd-timeout: A whole number timeout in seconds to be used for the remove etcd from nodes step. (default: none). If no value is specified, remove-etcd will use its own default.
- etcd-ready-timeout: A whole number in seconds indicating how long the decommission node request should allow for etcd clusters to become stable (default: 600).

Process

Acting upon the node specified by the invocation and the design reference details:

1. Drain the Kubernetes node.
2. Clear the Kubernetes labels on the node.
3. Remove etcd nodes from their clusters (if impacted).
    - if the node being decommissioned contains etcd nodes, Promenade will attempt to gracefully have those nodes leave the etcd cluster.
4. Ensure that etcd cluster(s) are in a stable state.
    - Polls for status every 30 seconds up to the etcd-ready-timeout, or the cluster meets the defined minimum functionality for the site.
    - A new document: promenade/EtcdClusters/v1 that will specify details about the etcd clusters deployed in the site, including: identifiers, credentials, and thresholds for minimum functionality.
    - This process should ignore the node being torn down from any calculation of health
5. Shutdown the kubelet.
    - If this is not possible because the node is in a state of disarray such that it cannot schedule the daemonset to run, this step may fail, but should not hold up the process, as the Drydock dismantling of the node will shut the kubelet down.

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  Indicates that all steps are successful.
- Failure: Code: 404, reason: NotFound

  Indicates that the target node is not discoverable by Promenade.

- Failure: Code: 500, reason: DisassociateStepFailure

  The details section should detail the successes and failures further. Any 4xx series errors from the individual steps would manifest as a 500 here.

Promenade Drain Node

Drain the Kubernetes node for the target node. This will ensure that this node is no longer the target of any pod scheduling, and evicts or deletes the running pods. In the case of notes running DaemonSet manged pods, or pods that would prevent a drain from occurring, Promenade may be required to provide the *ignore-daemonsets* option or *force* option to attempt to drain the node as fully as possible.

By default, the drain node will utilize a grace period for pods of 1800 seconds and a total timeout of 3600 seconds (1 hour). Clients of this functionality should be prepared for a long timeout.

```
POST /nodes/{node_id}/drain
```

Query Paramters:

- timeout: a whole number in seconds (default = 3600). This value is the total timeout for the kubectl drain command.
- grace-period: a whole number in seconds (default = 1800). This value is the grace period used by kubectl drain. Grace period must be less than timeout.

Note: This POST has no message body

Example command being used for drain (reference only) *kubectl drain –force –timeout 3600s –grace-period 1800 –ignore-daemonsets –delete-local-data n1* https://git.openstack.org/cgit/openstack/airship-promenade/tree/promenade/templates/roles/common/usr/local/bin/promenade-teardown

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  Indicates that the drain node has successfully concluded, and that no pods are currently running

- Failure: Status response, code: 400, reason: BadRequest

  A request was made with parameters that cannot work - e.g. grace-period is set to a value larger than the timeout value.

- Failure: Status response, code: 404, reason: NotFound

  The specified node is not discoverable by Promenade

- Failure: Status response, code: 500, reason: DrainNodeError

  There was a processing exception raised while trying to drain a node. The details section should indicate the underlying cause if it can be determined.

Promenade Clear Labels

Removes the labels that have been added to the target kubernetes node.

```
POST /nodes/{node_id}/clear-labels
```

Query Parameters:

- timeout: A whole number in seconds allowed for the pods to settle/move following removal of labels. (Default = 1800)

Note: This POST has no message body

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  All labels have been removed from the specified Kubernetes node.

- Failure: Code: 404, reason: NotFound

  The specified node is not discoverable by Promenade

- Failure: Code: 500, reason: ClearLabelsError

  There was a failure to clear labels that prevented completion. The details section should provide more information about the cause of this failure.

Promenade Remove etcd Node

Checks if the node specified contains any etcd nodes. If so, this API will trigger that etcd node to leave the associated etcd cluster:

```
POST /nodes/{node_id}/remove-etcd

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

Query Parameters:

- timeout: A whole number in seconds allowed for the removal of etcd nodes from the targe node. (Default = 1800)

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  All etcd nodes have been removed from the specified node.

- Failure: Code: 404, reason: NotFound

  The specified node is not discoverable by Promenade

- Failure: Code: 500, reason: RemoveEtcdError

  There was a failure to remove etcd from the target node that prevented completion within the specified timeout, or that etcd prevented removal of the node because it would result in the cluster being broken. The details section should provide more information about the cause of this failure.

Promenade Check etcd

Retrieves the current interpreted state of etcd.

> GET /etcd-cluster-health-statuses?design_ref={the design ref}

Where the design_ref parameter is required for appropriate operation, and is in the same format as used for the join-scripts API.

Query Parameters:

- design_ref: (Required) the design reference to be used to discover etcd instances.

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  The status of each etcd in the site will be returned in the details section. Valid values for status are: Healthy, Unhealthy

  https://github.com/openstack/airship-in-a-bottle/blob/master/doc/source/api-conventions.rst#status-responses

```
{ "...": "... standard status response ...",
  "details": {
    "errorCount": {{n}},
    "messageList": [
      { "message": "Healthy",
        "error": false,
        "kind": "HealthMessage",
        "name": "{{the name of the etcd service}}"
      },
      { "message": "Unhealthy"
        "error": false,
        "kind": "HealthMessage",
        "name": "{{the name of the etcd service}}"
      },
      { "message": "Unable to access Etcd"
        "error": true,
        "kind": "HealthMessage",
        "name": "{{the name of the etcd service}}"
      }
    ]
  }
  ...
}
```

- Failure: Code: 400, reason: MissingDesignRef

  Returned if the design_ref parameter is not specified

- Failure: Code: 404, reason: NotFound

  Returned if the specified etcd could not be located

- Failure: Code: 500, reason: EtcdNotAccessible

  Returned if the specified etcd responded with an invalid health response (Not just simply unhealthy - that's a 200).

Promenade Shutdown Kubelet

Shuts down the kubelet on the specified node. This is accomplished by Promenade setting the label *promenade-decomission: enabled* on the node, which will trigger a newly-developed daemonset to run something like: *systemctl disable kubelet && systemctl stop kubelet*. This daemonset will effectively sit dormant until nodes have the appropriate label added, and then perform the kubelet teardown.

```
POST /nodes/{node_id}/shutdown-kubelet
```

Note: This POST has no message body

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  The kubelet has been successfully shutdown

- Failure: Code: 404, reason: NotFound

  The specified node is not discoverable by Promenade

- Failure: Code: 500, reason: ShutdownKubeletError

  The specified node's kubelet fails to shutdown. The details section of the status response should contain reasonable information about the source of this failure

Promenade Delete Node from Cluster

Updates the Kubernetes cluster, removing the specified node. Promenade should check that the node is drained/cordoned and has no labels other than *promenade-decomission: enabled*. In either of these cases, the API should respond with a 409 Conflict response.

```
POST /nodes/{node_id}/remove-from-cluster
```

Note: This POST has no message body

Responses

All responses will be form of the Airship Status response.

- Success: Code: 200, reason: Success

  The specified node has been removed from the Kubernetes cluster.

- Failure: Code: 404, reason: NotFound

  The specified node is not discoverable by Promenade

- Failure: Code: 409, reason: Conflict

  The specified node cannot be deleted due to checks that the node is drained/cordoned and has no labels (other than possibly *promenade-decomission: enabled*).

- Failure: Code: 500, reason: DeleteNodeError

  The specified node cannot be removed from the cluster due to an error from Kubernetes. The details section of the status response should contain more information about the failure.

Shipyard Tag Releases

Shipyard will need to mark Deckhand revisions with tags when there are successful deploy_site or update_site actions to be able to determine the last known good design. This is related to issue 16 for Shipyard, which utilizes the same need.

Note: Repeated from https://github.com/att-comdev/shipyard/issues/16

When multiple configdocs commits have been done since the last deployment, there is no ready means to determine what's being done to the site. Shipyard should reject deploy site or update site requests that have had multiple commits since the last site true-up action. An option to override this guard should be allowed for the actions in the form of a parameter to the action.

The configdocs API should provide a way to see what's been changed since the last site true-up, not just the last commit of configdocs. This might be accommodated by new deckhand tags like the 'commit' tag, but for 'site true-up' or similar applied by the deploy and update site commands.

The design for issue 16 includes the bare-minimum marking of Deckhand revisions. This design is as follows:

Scenario

Multiple commits occur between site actions (deploy_site, update_site) - those actions that attempt to bring a site into compliance with a site design. When this occurs, the current system of being able to only see what has changed between committed and the buffer versions (configdocs diff) is insufficient to be able to investigate what has changed since the last successful (or unsuccessful) site action. To accommodate this, Shipyard needs several enhancements.

Enhancements

1. Deckhand revision tags for site actions

   Using the tagging facility provided by Deckhand, Shipyard will tag the end of site actions. Upon completing a site action successfully tag the revision being used with the tag site-action-success, and a body of dag_id:<dag_id>

   Upon completion of a site action unsuccessfully, tag the revision being used with the tag site-action-failure, and a body of dag_id:<dag_id>

The completion tags should only be applied upon failure if the site action gets past document validation successfully (i.e. gets to the point where it can start making changes via the other Airship components)

This could result in a single revision having both site-action-success and site-action-failure if a later re-invocation of a site action is successful.

2. Check for intermediate committed revisions

   Upon running a site action, before tagging the revision with the site action tag(s), the dag needs to check to see if there are committed revisions that do not have an associated site-action tag. If there are any committed revisions since the last site action other than the current revision being used (between them), then the action should not be allowed to proceed (stop before triggering validations). For the calculation of intermediate committed revisions, assume revision 0 if there are no revisions with a site-action tag (null case)

   If the action is invoked with a parameter of allow-intermediate-commits=true, then this check should log that the intermediate committed revisions check is being skipped and not take any other action.

3. Support action parameter of allow-intermediate-commits=true|false

   In the CLI for create action, the –param option supports adding parameters to actions. The parameters passed should be relayed by the CLI to the API and ultimately to the invocation of the DAG. The DAG as noted above will check for the presense of allow-intermediate-commits=true. This needs to be tested to work.

4. Shipyard needs to support retrieving configdocs and rendered documents for the last successful site action, and last site action (successful or not successful)

   –successful-site-action –last-site-action These options would be mutually exclusive of –buffer or –committed

5. Shipyard diff (shipyard get configdocs)

   Needs to support an option to do the diff of the buffer vs. the last successful site action and the last site action (succesful or not successful).

   Currently there are no options to select which versions to diff (always buffer vs. committed)

   support: –base-version=committed | successful-site-action | last-site-action (Default = committed) –diff-version=buffer | committed | successful-site-action | last-site-action (Default = buffer)

   Equivalent query parameters need to be implemented in the API.

Because the implementation of this design will result in the tagging of successful site-actions, Shipyard will be able to determine the correct revision to use while attempting to teardown a node.

If the request to teardown a node indicates a revision that doesn't exist, the command to do so (e.g. redeploy_server) should not continue, but rather fail due to a missing precondition.

The invocation of the Promenade and Drydock steps in this design will utilize the appropriate tag based on the request (default is successful-site-action) to determine the revision of the Deckhand documents used as the design-ref.

Shipyard redeploy_server Action

The redeploy_server action currently accepts a target node. Additional supported parameters are needed:

1. preserve-local-storage=true which will instruct Drydock to only wipe the OS drive, and any other local storage will not be wiped. This would allow for the drives to be remounted to the server upon re-provisioning. The default behavior is that local storage is not preserved.

2. target-revision=committed | successful-site-action | last-site-action This will indicate which revision of the design will be used as the reference for what should be re-provisioned after the teardown. The default is successful-site-action, which is the closest representation to the last-known-good state.

These should be accepted as parameters to the action API/CLI and modify the behavior of the redeploy_server DAG.

Security impact

None. This change introduces no new security concerns outside of established patterns for RBAC controls around API endpoints.

Performance impact

As this is an on-demand action, there is no expected performance impact to existing processes, although tearing down a host may result in temporary degraded service capacity in the case of needing to move workloads to different hosts, or a more simple case of reduced capacity.

Alternatives

N/A

Implementation

None at this time

Dependencies

None.

References

None

3.2 Implemented Specs

3.2.1 Deployment Grouping for Baremetal Nodes

One of the primary functionalities of the Undercloud Platform is the deployment of baremetal nodes as part of site deployment and upgrade. This blueprint aims to define how deployment strategies can be applied to the workflow during these actions.

Note: This document has been moved from the airship-in-a-bottle project, and is previously implemented. The format of this document diverges from the standard template for airship-specs.

Overview

When Shipyard is invoked for a deploy_site or update_site action, there are three primary stages:

1. Preparation and Validation
2. Baremetal and Network Deployment
3. Software Deployment

During the Baremetal and Network Deployment stage, the deploy_site or update_site workflow (and perhaps other workflows in the future) invokes Drydock to verify the site, prepare the site, prepare the nodes, and deploy the nodes. Each of these steps is described in the Drydock Orchestrator Readme

The prepare nodes and deploy nodes steps each involve intensive and potentially time consuming operations on the target nodes, orchestrated by Drydock and MAAS. These steps need to be approached and managed such that grouping, ordering, and criticality of success of nodes can be managed in support of fault tolerant site deployments and updates.

For the purposes of this document *phase of deployment* refer to the prepare nodes and deploy nodes steps of the Baremetal and Network deployment.

Some factors that advise this solution:

1. Limits to the amount of parallelization that can occur due to a centralized MAAS system.
2. Faults in the hardware, preventing operational nodes.
3. Miswiring or configuration of network hardware.
4. Incorrect site design causing a mismatch against the hardware.
5. Criticality of particular nodes to the realization of the site design.
6. Desired configurability within the framework of the Airship declarative site design.
7. Improved visibility into the current state of node deployment.
8. A desire to begin the deployment of nodes before the finish of the preparation of nodes – i.e. start deploying nodes as soon as they are ready to be deployed. Note: This design will not achieve new forms of task parallelization within Drydock; this is recognized as a desired functionality.

Solution

Updates supporting this solution will require changes to Shipyard for changed workflows and Drydock for the desired node targeting, and for retrieval of diagnostic and result information.

Deployment Strategy Document (Shipyard)

To accommodate the needed changes, this design introduces a new DeploymentStrategy document into the site design to be read and utilized by the workflows for update_site and deploy_site.

Groups

Groups are named sets of nodes that will be deployed together. The fields of a group are:

name Required. The identifying name of the group.

critical Required. Indicates if this group is required to continue to additional phases of deployment.

depends_on Required, may be empty list. Group names that must be successful before this group can be processed.

selectors Required, may be empty list. A list of identifying information to indicate the nodes that are members of this group.

success_criteria Optional. Criteria that must evaluate to be true before a group is considered successfully complete with a phase of deployment.

Criticality

- Field: critical
- Valid values: true | false

Each group is required to indicate true or false for the *critical* field. This drives the behavior after the deployment of baremetal nodes. If any groups that are marked as *critical: true* fail to meet that group's success criteria, the workflow should halt after the deployment of baremetal nodes. A group that cannot be processed due to a parent dependency failing will be considered failed, regardless of the success criteria.

Dependencies

- Field: depends_on
- Valid values: [] or a list of group names

Each group specifies a list of depends_on groups, or an empty list. All identified groups must complete successfully for the phase of deployment before the current group is allowed to be processed by the current phase.

- A failure (based on success criteria) of a group prevents any groups dependent upon the failed group from being attempted.
- Circular dependencies will be rejected as invalid during document validation.
- There is no guarantee of ordering among groups that have their dependencies met. Any group that is ready for deployment based on declared dependencies will execute. Execution of groups is serialized - two groups will not deploy at the same time.

Selectors

- Field: selectors
- Valid values: [] or a list of selectors

The list of selectors indicate the nodes that will be included in a group. Each selector has four available filtering values: node_names, node_tags, node_labels, and rack_names. Each selector is an intersection of this critera, while the list of selectors is a union of the individual selectors.

- Omitting a criterion from a selector, or using empty list means that criterion is ignored.
- Having a completely empty list of selectors, or a selector that has no criteria specified indicates ALL nodes.
- A collection of selectors that results in no nodes being identified will be processed as if 100% of nodes successfully deployed (avoiding division by zero), but would fail the minimum or maximum nodes criteria (still counts as 0 nodes)
- There is no validation against the same node being in multiple groups, however the workflow will not resubmit nodes that have already completed or failed in this deployment to Drydock twice, since it keeps track of each node uniquely. The success or failure of those nodes excluded from submission to Drydock will still be used for the success criteria calculation.

E.g.:

```
selectors:
  - node_names:
      - node01
      - node02
```

(continues on next page)

(continued from previous page)

```
    rack_names:
      - rack01
    node_tags:
      - control
  - node_names:
      - node04
    node_labels:
      - ucp_control_plane: enabled
```

Will indicate (not really SQL, just for illustration):

```
SELECT nodes
WHERE node_name in ('node01', 'node02')
      AND rack_name in ('rack01')
      AND node_tags in ('control')
UNION
SELECT nodes
WHERE node_name in ('node04')
      AND node_label in ('ucp_control_plane: enabled')
```

Success Criteria

- Field: success_criteria
- Valid values: for possible values, see below

Each group optionally contains success criteria which is used to indicate if the deployment of that group is successful. The values that may be specified:

percent_successful_nodes The calculated success rate of nodes completing the deployment phase.

E.g.: 75 would mean that 3 of 4 nodes must complete the phase successfully.

This is useful for groups that have larger numbers of nodes, and do not have critical minimums or are not sensitive to an arbitrary number of nodes not working.

minimum_successful_nodes An integer indicating how many nodes must complete the phase to be considered successful.

maximum_failed_nodes An integer indicating a number of nodes that are allowed to have failed the deployment phase and still consider that group successful.

When no criteria are specified, it means that no checks are done - processing continues as if nothing is wrong.

When more than one criterion is specified, each is evaluated separately - if any fail, the group is considered failed.

Example Deployment Strategy Document

This example shows a deployment strategy with 5 groups: control-nodes, compute-nodes-1, compute-nodes-2, monitoring-nodes, and ntp-node.

```
---
schema: shipyard/DeploymentStrategy/v1
metadata:
  schema: metadata/Document/v1
  name: deployment-strategy
```

(continues on next page)

(continued from previous page)

```
  layeringDefinition:
      abstract: false
      layer: global
  storagePolicy: cleartext
data:
  groups:
    - name: control-nodes
      critical: true
      depends_on:
        - ntp-node
      selectors:
        - node_names: []
          node_labels: []
          node_tags:
            - control
          rack_names:
            - rack03
      success_criteria:
        percent_successful_nodes: 90
        minimum_successful_nodes: 3
        maximum_failed_nodes: 1
    - name: compute-nodes-1
      critical: false
      depends_on:
        - control-nodes
      selectors:
        - node_names: []
          node_labels: []
          rack_names:
            - rack01
          node_tags:
            - compute
      success_criteria:
        percent_successful_nodes: 50
    - name: compute-nodes-2
      critical: false
      depends_on:
        - control-nodes
      selectors:
        - node_names: []
          node_labels: []
          rack_names:
            - rack02
          node_tags:
            - compute
      success_criteria:
        percent_successful_nodes: 50
    - name: monitoring-nodes
      critical: false
      depends_on: []
      selectors:
        - node_names: []
          node_labels: []
          node_tags:
            - monitoring
          rack_names:
            - rack03
```

(continues on next page)

(continued from previous page)

```
          - rack02
          - rack01
  - name: ntp-node
    critical: true
    depends_on: []
    selectors:
      - node_names:
          - ntp01
        node_labels: []
        node_tags: []
        rack_names: []
    success_criteria:
      minimum_successful_nodes: 1
```

The ordering of groups, as defined by the dependencies (`depends-on` fields):

```
 __________     __________________
| ntp-node |   | monitoring-nodes |
 ----------     ------------------
     |
 ____V__________
| control-nodes |
 ---------------
     |_________________________
         |                     |
   ______V__________     ______V__________
  | compute-nodes-1 |   | compute-nodes-2 |
   -----------------     -----------------
```

Given this, the order of execution could be:

- ntp-node > monitoring-nodes > control-nodes > compute-nodes-1 > compute-nodes-2
- ntp-node > control-nodes > compute-nodes-2 > compute-nodes-1 > monitoring-nodes
- monitoring-nodes > ntp-node > control-nodes > compute-nodes-1 > compute-nodes-2
- and many more ... the only guarantee is that ntp-node will run some time before control-nodes, which will run sometime before both of the compute-nodes. Monitoring-nodes can run at any time.

Also of note are the various combinations of selectors and the varied use of success criteria.

Deployment Configuration Document (Shipyard)

The existing deployment-configuration document that is used by the workflows will also be modified to use the existing deployment_strategy field to provide the name of the deployment-straegy document that will be used.

The default value for the name of the DeploymentStrategy document will be `deployment-strategy`.

Drydock Changes

API and CLI

- A new API needs to be provided that accepts a node filter (i.e. selector, above) and returns a list of node names that result from analysis of the design. Input to this API will also need to include a design reference.

- Drydock needs to provide a "tree" output of tasks rooted at the requested parent task. This will provide the needed success/failure status for nodes that have been prepared/deployed.

Documentation

Drydock documentation will be updated to match the introduction of new APIs

Shipyard Changes

API and CLI

- The commit configdocs api will need to be enhanced to look up the DeploymentStrategy by using the DeploymentConfiguration.
- The DeploymentStrategy document will need to be validated to ensure there are no circular dependencies in the groups' declared dependencies (perhaps NetworkX).
- A new API endpoint (and matching CLI) is desired to retrieve the status of nodes as known to Drydock/MAAS and their MAAS status. The existing node list API in Drydock provides a JSON output that can be utilized for this purpose.

Workflow

The deploy_site and update_site workflows will be modified to utilize the DeploymentStrategy.

- The deployment configuration step will be enhanced to also read the deployment strategy and pass the information on a new xcom for use by the baremetal nodes step (see below)
- The prepare nodes and deploy nodes steps will be combined to perform both as part of the resolution of an overall `baremetal nodes` step. The baremetal nodes step will introduce functionality that reads in the deployment strategy (from the prior xcom), and can orchestrate the calls to Drydock to enact the grouping, ordering and success evaluation. Note that Drydock will serialize tasks; there is no parallelization of prepare/deploy at this time.

Needed Functionality

- function to formulate the ordered groups based on dependencies (perhaps NetworkX)
- function to evaluate success/failure against the success criteria for a group based on the result list of succeeded or failed nodes.
- function to mark groups as success or failure (including failed due to dependency failure), as well as keep track of the (if any) successful and failed nodes.
- function to get a group that is ready to execute, or 'Done' when all groups are either complete or failed.
- function to formulate the node filter for Drydock based on a group's selectors
- function to orchestrate processing groups, moving to the next group (or being done) when a prior group completes or fails.
- function to summarize the success/failed nodes for a group (primarily for reporting to the logs at this time).

Process

The baremetal nodes step (preparation and deployment of nodes) will proceed as follows:

1. Each group's selector will be sent to Drydock to determine the list of nodes that are a part of that group.
    - An overall status will be kept for each unique node (not started | prepared | success | failure).
    - When sending a task to Drydock for processing, the nodes associated with that group will be sent as a simple *node_name* node filter. This will allow for this list to exclude nodes that have a status that is not congruent for the task being performed.
        - prepare nodes valid status: not started
        - deploy nodes valid status: prepared
2. In a processing loop, groups that are ready to be processed based on their dependencies (and the success criteria of groups they are dependent upon) will be selected for processing until there are no more groups that can be processed. The processing will consist of preparing and then deploying the group.
    - The selected group will be prepared and then deployed before selecting another group for processing.
    - Any nodes that failed as part of that group will be excluded from subsequent deployment or preparation of that node for this deployment.
        - Excluding nodes that are already processed addresses groups that have overlapping lists of nodes due to the group's selectors, and prevents sending them to Drydock for re-processing.
        - Evaluation of the success criteria will use the full set of nodes identified by the selector. This means that if a node was previously successfully deployed, that same node will count as "successful" when evaluating the success criteria.
    - The success criteria will be evaluated after the group's prepare step and the deploy step. A failure to meet the success criteria in a prepare step will cause the deploy step for that group to be skipped (and marked as failed).
    - Any nodes that fail during the prepare step, will not be used in the corresponding deploy step.
    - Upon completion (success, partial success, or failure) of a prepare step, the nodes that were sent for preparation will be marked in the unique list of nodes (above) with their appropriate status: prepared or failure
    - Upon completion of a group's deployment step, the nodes status will be updated to their current status: success or failure.

4. Before the end of the baremetal nodes step, following all eligible group processing, a report will be logged to indicate the success/failure of groups and the status of the individual nodes. Note that it is possible for individual nodes to be left in *not started* state if they were only part of groups that were never allowed to process due to dependencies and success criteria.
5. At the end of the baremetal nodes step, if any nodes that have failed due to timeout, dependency failure, or success criteria failure and are marked as critical will trigger an Airflow Exception, resulting in a failed deployment.

Notes:

- The timeout values specified for the prepare nodes and deploy nodes steps will be used to put bounds on the individual calls to Drydock. A failure based on these values will be treated as a failure for the group; we need to be vigilant on if this will lead to indeterminate states for nodes that mess with further processing. (e.g. Timed out, but the requested work still continued to completion)

Example Processing

Using the defined deployment strategy in the above example, the following is an example of how it may process:

```
Start
|
| prepare ntp-node           <SUCCESS>
| deploy ntp-node            <SUCCESS>
V
| prepare control-nodes      <SUCCESS>
| deploy control-nodes       <SUCCESS>
V
| prepare monitoring-nodes   <SUCCESS>
| deploy monitoring-nodes    <SUCCESS>
V
| prepare compute-nodes-2    <SUCCESS>
| deploy compute-nodes-2     <SUCCESS>
V
| prepare compute-nodes-1    <SUCCESS>
| deploy compute-nodes-1     <SUCCESS>
|
Finish (success)
```

If there were a failure in preparing the ntp-node, the following would be the result:

```
Start
|
| prepare ntp-node           <FAILED>
| deploy ntp-node            <FAILED, due to prepare failure>
V
| prepare control-nodes      <FAILED, due to dependency>
| deploy control-nodes       <FAILED, due to dependency>
V
| prepare monitoring-nodes   <SUCCESS>
| deploy monitoring-nodes    <SUCCESS>
V
| prepare compute-nodes-2    <FAILED, due to dependency>
| deploy compute-nodes-2     <FAILED, due to dependency>
V
| prepare compute-nodes-1    <FAILED, due to dependency>
| deploy compute-nodes-1     <FAILED, due to dependency>
|
Finish (failed due to critical group failed)
```

If a failure occurred during the deploy of compute-nodes-2, the following would result:

```
Start
|
| prepare ntp-node           <SUCCESS>
| deploy ntp-node            <SUCCESS>
V
| prepare control-nodes      <SUCCESS>
| deploy control-nodes       <SUCCESS>
V
| prepare monitoring-nodes   <SUCCESS>
| deploy monitoring-nodes    <SUCCESS>
V
```

(continues on next page)

(continued from previous page)

```
| prepare compute-nodes-2    <SUCCESS>
| deploy compute-nodes-2     <FAILED>
V
| prepare compute-nodes-1    <SUCCESS>
| deploy compute-nodes-1     <SUCCESS>
|
Finish (success with some nodes/groups failed)
```

Schemas

A new schema will need to be provided by Shipyard to validate the DeploymentStrategy document.

Documentation

The Shipyard action documentation will need to include details defining the DeploymentStrategy document (mostly as defined here), as well as the update to the DeploymentConfiguration document to contain the name of the DeploymentStrategy document.

Index