Airship Documentation

Release 0.1.0

Airship Authors

Mar 19, 2019

Contents

Note: These documents will be reworked to reflect the changes associated with becoming an OpenStack hosted project: Airship. Expect major changes to occur with time. See more at airshipit.org

Airship is a collection of components that coordinate to form a means of configuring, deploying and maintaining a Kubernetes environment using a declarative set of yaml documents. More details on using parts of Airship may be found by using the Treasuremap

CHAPTER 1

Approach

Airship revolves around the setup and use of Kubernetes and Helm and takes cues from these projects. The first use case of Airship is the deployment of OpenStack-Helm which also influences Airship's direction.

CHAPTER 2

Building this Documentation

Use of `make docs` will build a html version of this documentation that can be viewed using a browser at doc/build/index.html on the local filesystem.

CHAPTER 3

Specification Details

Proposed, approved, and implemented specifications for Airship projects are available.

CHAPTER 4

Conventions and Standards

4.1 Airship Conventions

Airship components conform to a minimal set of conventions to provide for reasonable levels of consistency.

4.1.1 Language

While these documents are not an IETF RFC, RFC 2119 provides for useful language definitions. In this spirit:

- 'must', 'shall', 'will', and 'required' language indicates inflexible rules.
- 'should' and 'recommended' language is expected to be followed but reasonable exceptions may exist.
- 'may' and 'can' language is intended to be optional, but will provide a recommended approach if used.

4.1.2 Conventions and Standards

API Conventions

A collection of conventions that components of Airship utilize for their REST APIs

Resource path naming

- Resource paths nodes follow an all lower case naming scheme, and pluralize the resource names. Nodes that refer to keys, ids or names that are externally controlled, the external naming will be honored.
- The version of the API resource path will be prefixed before the first node of the path for that resource using v#.# format.
- By default and unless otherwise noted, the API will be namespaced by /api before the version. For the purposes of documentation, this will not be specified in each of the resource paths below. In more complex APIs, Airship components may use values other than /api to be more specific to point to a particular service.

```
/api/v1.0/sampleresources/ExTeRnAlNAME-1234
      ^          ^        ^        ^
      |          |        |      defer to external naming
      |          |      plural
      |        lower case
     version here
```

Status responses

Status responses, and more specifically error responses (HTTP response body accompanying 4xx and 5xx series responses where possible) are a customized version of the Kubernetes standard for error representation. Airship utilizes the details field in a more formalized way to represent multiple messages related to a status response, as follows:

```
{
  "kind": "Status",
  "apiVersion": "v{{#.#}}",
  "metadata": {},
  "status": "{{Success | Failure}}",
  "message": "{{message phrase}}",
  "reason": "{{reason name}}",
  "details": {
    "errorCount": {{n}},
    "messageList": [
       { "message" : "{{message contents}}",
         "error": true|false,
         "kind": "SimpleMessage" }
       ...
    ]
  },
  "code": {{http status code}}
}
```

such that:

- The metadata field is optionally present, as an empty object. Clients should be ready to receive this field, but services are not required to produce it.
- The message phrase is a terse but descriptive message indicating what has happened.
- The reason name is the short name indicating the cause of the status. It should be a camel cased phrase-as-a-word, to mimic the Kubernetes status usage.
- The details field is optional.
- If used, the details follow the shown format, with an errorCount and messageList field present.
- The repeating entity inside the messageList can be decorated with as many other fields as are useful, but at least have a message field and error field.
  - A kind field is optional, but if used will indicate the presence of other fields. By default, the kind field is assumed to be "SimpleMessage", which requires only the aforementioned message and error fields.
- The errorCount field is an integer representing the count of messageList entities that have `error: true`
- When using this document as the body of a HTTP response, `code` is populated with a valid HTTP status code

Required Headers

X-Auth-Token The auth token to identify the invoking user. Required unless the resource is explictly unauthenticated.

Optional Headers

X-Context-Marker A context id that will be carried on all logs for this client-provided marker. This marker may only be a 36-character canonical representation of an UUID (8-4-4-4-12)

Validation API

All Airship components that participate in validation of the design supplied to a site implement a common resource to perform document validations. Document validations are synchronous. Because of the different sources of documents that should be supported, a flexible input descriptor is used to indicate from where an Airship component will retrieve the documents to be validated.

POST /v1.0/validatedesign

Invokes an Airship component to perform validations against the documents specified by the input structure. Synchronous.

Input structure

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

Output structure

The output structure reuses the Kubernetes Status kind to represent the result of validations. The Status kind will be returned for both successful and failed validation to maintain a consistent of interface. If there are additional diagnostics that associate to a particular validation, the entries in the messageList should be of kind "ValidationMessage" (preferred), or "SimpleMessage" (assumed default base message kind).

Failure message example using a ValidationMessage kind for the messageList:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Failure",
  "message": "{{Component Name}} validations failed",
  "reason": "Validation",
  "details": {
    "errorCount": {{n}},
    "messageList": [
```

(continues on next page)

(continued from previous page)

```
        { "message" : "{{validation failure message}}",
          "error": true,
          "name": "{{identifying name of the validation}}",
          "documents": [
              { "schema": "{{schema and name of the document being validated}}",
                "name": "{{name of the document being validated}}"
              },
              ...
          ]
          "level": "Error",
          "diagnostic": "{{information about what lead to the message}}",
          "kind": "ValidationMessage" },
        ...
    ]
  },
  "code": 400
}
```

Success message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Success",
  "message": "{{Component Name}} validations succeeded",
  "reason": "Validation",
  "details": {
    "errorCount": 0,
    "messageList": []
  },
  "code": 200
}
```

ValidationMessage Message Type

The ValidationMessage message type is used to provide more information about validation results than a SimpleMessage provides. These are the fields of a ValidationMessage:

- documents (optional): If applicable to configuration documents, specifies the design documents by schema and name that were involved in the specific validation. If the documents element is not provided, or is an empty list, the assumption is that the validation is not traced to a document, and may be a validaiton of environmental or process needs.
  - schema (required): The schema of the document. E.g. drydock/NetworkLink/v1
  - name (required): The name of the document. E.g. pxe-rack1
- error (required): true if the message indcates an error, false if the message indicates a non-error.
- kind (required): ValidationMessage
- level (required): The severity of the validation result. This should align with the error field value. Valid values are "Error", "Warning", and "Info".
- message (required): The more complete message indicating the result of the validation. E.g.: MTU 8972 for pxe-rack1 is invalid for standard (non-jumbo) frames

- name (required): The name of the validation being performed. This is a short name that identifies the validation among a full set of validations. It is preferred to use non-action words to identify the validation. E.g. "MTU in bounds" is preferred instead of "Check MTU in bounds"
- diagnostic (optional): Provides further contextual information that may help with determining the source of the validation or provide further details.

Health Check API

Each Airship component shall expose an endpoint that allows other components to access and validate its health status. Clients of the health check should wait up to 30 seconds for a health check response from each component.

GET /v1.0/health

Invokes an Airship component to return its health status. This endpoint is intended to be unauthenticated, and must not return any information beyond the noted 204 or 503 status response. The component invoked is expected to return a response in less than 30 seconds.

Health Check Output

The current design will be for the component to return an empty response to show that it is alive and healthy. This means that the component that is performing the query will receive HTTP response code 204.

HTTP response code 503 with a generic response status or an empty message body will be returned if the component determines it is in a non-healthy state, or is unable to reach another component it is dependent upon.

GET /v1.0/health/extended

Airship components may provide an extended health check. This request invokes a component to return its detailed health status. Authentication is required to invoke this API call.

Extended Health Check Output

The output structure reuses the Kubernetes Status kind to represent the health check results. The Status kind will be returned for both successful and failed health checks to ensure consistencies. The message field will contain summary information related to the results of the health check. Detailed information of the health check will be provided as well.

Failure message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Failure",
  "message": "{{Component Name}} failed to respond",
  "reason": "HealthCheck",
  "details": {
    "errorCount": {{n}},
    "messageList": [
       { "message" : "{{Detailed Health Check failure information}}",
```

(continues on next page)

(continued from previous page)

```
          "error": true,
          "kind": "SimpleMessage" },
        ...
    ]
  },
  "code": 503
}
```

Success message example:

```
{
  "kind": "Status",
  "apiVersion": "v1.0",
  "metadata": {},
  "status": "Success",
  "message": "",
  "reason": "HealthCheck",
  "details": {
    "errorCount": 0,
    "messageList": []
  },
  "code": 200
}
```

Versions API

Each Airship component shall expose an endpoint that allows other components to discover its different API versions. This endpoint is not prefixed by /api or a version.

GET /versions

Invokes an Airship component to return its list of API versions. This endpoint is intended to be unauthenticated, and must not return any information beyond the output noted below.

Versions output

Each Airship component shall return a list of its different API versions. The response body shall be keyed with the name of each API version, with accompanying information pertaining to the version's *path* and *status*. The *status* field shall be an enum which accepts the values *stable* and *beta*, where *stable* implies a stable API and *beta* implies an under-development API.

Success message example:

```
{
  "v1.0": {
    "path": "/api/v1.0",
    "status": "stable"
  },
  "v1.1": {
    "path": "/api/v1.1",
    "status": "beta"
  },
```

(continues on next page)

(continued from previous page)

```
  "code": 200
}
```

Code and Project Conventions

Conventions and standards that guide the development and arrangement of Airship component projects.

Project Structure

Charts

Each project that maintains helm charts will keep those charts in a directory `charts` located at the root of the project. The charts directory will contain subdirectories for each of the charts maintained as part of that project. These subdirectories should be named for the component represented by that chart.

e.g.: For project `foo`, which also maintains the charts for `bar` and `baz`:

- foo/charts/foo contains the chart for `foo`
- foo/charts/bar contains the chart for `bar`
- foo/charts/baz contains the chart for `baz`

Helm charts utilize the helm-toolkit supported by the Openstack-Helm team and follow the standards documented there.

Images

Each project that creates a Docker image will keep the dockerfile in a directory `images` located at the root of the project. The images directory will contain subdirectories for each of the images created as part of that project. The subdirectory will contain the dockerfile that can be used to generate the image.

e.g.: For project `foo`, which also produces a Docker image for `bar`:

- foo/images/foo contains the dockerfile for `foo`
- foo/images/bar contains the dockerfile for `bar`

Makefile

Each project must provide a makefile at the root of the project. The makefile should implement each of the following makefile targets:

- `images` will produce the docker images for the component and each other component it is responsible for building.
- `charts` will helm package all of the charts maintained as part of the project.
- `lint` will perform code linting for the code and chart linting for the charts maintained as part of the project, as well as any other reasonable linting activity.
- `dry-run` will produce a helm template for the charts maintained as part of the project.
- `all` will run the lint, charts, and images targets.

- `docs` should render any documentation that has build steps.
- `run_{component_name}` should build the image and do a rudimentary (at least) test of the image's functionality.
- `run_images` performs the inidividual run_{component_name} targets for projects that produce more than one image.
- `tests` to invoke linting tests (e.g. PEP-8) and unit tests for the components in the project

For projects that are Python based, the makefile targets typically reference tox commands, and those projects will include a tox.ini defining the tox targets. Note that tox.ini files will reside inside the source directories for modules within the project, but a top-level tox.ini may exist at the root of the repository that includes the necessary targets to build documentation.

Documentation

Also see *Documentation*

Documentation source for the component should reside in a 'docs' directory at the root of the project.

Linting and Formatting Standards

Code in the Airship components should follow the prevalent linting and formatting standards for the language being implemented. In lieu of industry accepted code formatting standards for a target language, strive for readability and maintainability.

Known Standards	
Language	Uses
Python	PEP-8

Airship components must provide for automated checking of their formatting standards, such as the lint step noted above in the makefile. Components may provide automated reformatting.

Tests Location

Tests should be in parallel structures to the related code, unless dictated by target language ecosystem.

For Python projects, the preferred location for tests is a `tests` directory under the directory for the module. E.g. Tests for module foo: {root}/src/bin/foo/foo/tests. An alternataive location is `tests` at the root of the project, although this should only be used if there are not multiple components represented in the same repository, or if the tests cross the components in the repository.

Each type of test should be in its own subdirectory of tests, to allow for easy separation. E.g. tests/unit, tests/functional, tests/integration.

Source Code Location

A standard structure for the source code places the source for each module in a module-named directory under either /src/bin or /src/lib, for executable modules and shared library modules respectively. Since each module needs its own setup.py and setup.cfg (python) that lives parallel to the top-level module (i.e. the package), the directory for the module will contain another directory named the same.

For example, Project foo, with module foo_service would have a source structure that is /src/bin/foo_service/foo_service, wherein the __init__.py for the package resides.

Sample Project Structure (Python)

Project `foo`, supporting multiple executable modules `foo_service`, `foo_cli`, and a shared module `foo_client`

```
{root of foo}
 |- /doc
 |    |- /source
 |    |- requirements.txt
 |- /etc
 |    |- /foo
 |         |- {sample files}
 |- /charts
 |    |- /foo
 |    |- /bar
 |- /images
 |    |- /foo
 |    |    |- Dockerfile
 |    |- /bar
 |         |- Dockerfile
 |- /tools
 |    |- {scripts/utilities supporting build and test}
 |- /src
 |    |- /bin
 |    |    |- /foo_service
 |    |    |    |- /foo_service
 |    |    |    |    |- __init__.py
 |    |    |    |    |- {source directories and files}
 |    |    |    |- /tests
 |    |    |    |    |- unit
 |    |    |    |    |- functional
 |    |    |    |- setup.py
 |    |    |    |- setup.cfg
 |    |    |    |- requirements.txt (and related files)
 |    |    |    |- tox.ini
 |    |    |- /foo_cli
 |    |         |- /foo_cli
 |    |         |    |- __init__.py
 |    |         |    |- {source directories and files}
 |    |         |- /tests
 |    |         |    |- unit
 |    |         |    |- functional
 |    |         |- setup.py
 |    |         |- setup.cfg
 |    |         |- requirements.txt (and related files)
 |    |         |- tox.ini
 |    |- /lib
 |         |- /foo_client
 |              |- /foo_client
 |              |    |- __init__.py
 |              |    |- {source directories and files}
 |              |- /tests
 |              |    |- unit
```

(continues on next page)

(continued from previous page)

```
|               |    |- functional
|               |- setup.py
|               |- setup.cfg
|               |- requirements.txt (and related files)
|               |- tox.ini
|- Makefile
|- README  (suitable for github consumption)
|- tox.ini (primarily for the build of repository-level docs)
```

Note that this is a sample structure, and that target languages may preclude the location of some items (e.g. tests). For those components with language or ecosystem standards contrary to this structure, ecosystem convention should prevail.

Documentation

Each Airship component will maintain documentation addressing two audiences:

1. Consumer documentation
2. Developer documentation

Consumer Documentation

Consumer documentation is that which is intended to be referenced by users of the component. This includes information about each of the following:

- Introduction - the purpose and charter of the software
- Features - capabilies the software has
- Usage - interaction with the software - e.g. API and CLI documentation
- Setup/Installation - how an end user would set up and run the software including system requirements
- Support - where and how a user engages support or makes change requests for the software

Developer Documentation

Developer documentation is used by developers of the software, and addresses the following topics:

- Archiecture and Design - features and structure of the software
- Inline, Code, Method - documentaiton specific to the fuctions and procedures in the code
- Development Environment - explaining how a developer would need to configure a working environment for the software
- Contribution - how a developer can contribute to the software

Format

There are multiple means by which consumers and developers will read the documentation for Airship components. The two common places for Airship components are Github in the form of README and code-based documentation, and Readthedocs for more complete/formatted documentation.

Documentation that is expected to be read in Github must exist and may use either reStructuredText or Markdown. This generally would be limited to the README file at the root of the project and/or a documentation directory. The README should direct users to the published documentation location.

Documentation intended for Readthedocs will use reStructuredText, and should provide a Sphinx build of the documentation.

Finding Treasuremap

Treasuremap is a project that serves as a starting point for the larger Containerized Cloud Platform, and provides context for the Airship component projects.

Airship component projects should include the following at the top of the main/index page of their Readthedocs documentation:

Tip: {{component name}} is part of Airship, a collection of components that coordinate to form a means of configuring, deploying and maintaining a Kubernetes environment using a declarative set of yaml documents. More details on using Airship may be found by using the Treasuremap

---

Service Logging Conventions

Airship services must provide logging, should conform to a standard logging format, and may utilize shared code to do so.

Standard Logging Format

The following is the intended format to be used when logging from Airship services. When logging from those parts that are no services, a close reasonable approximation is desired.

```
Timestamp Level RequestID ExternalContextID ModuleName(Line) Function - Message
```

Where:

- Timestamp is like `2006-02-08 22:20:02,165`, or the standard ouptut from `%(asctime)s`
- Level is 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL', padded to 8 characters, left aligned.
- RequestID is the UUID assigned to the request in canonical 8-4-4-4-12 format.
- ExternalContextID is the UUID assigned from the external source (or generated for the same purpose), in 8-4-4-4-12 format.
- ModuleName is the name of the module or class from which the logging originates.
- Line is the line number of the logging statement
- Function is the name of the function or method from which the logging originates
- Message is the text of the message to be logged.

Example Python Logging Format

```
%(asctime)s %(levelname)-8s %(req_id)s %(external_ctx)s %(user)s %(module)s(
↪%(lineno)d) %(funcName)s - %(message)s'
```

See Python Logging for explanation of format.

Loggers in Code

Components should prefer loggers that are at the module or class level, allowing for finer grained logging control than a global logger.

4.2 Airship Security Guide

An undercloud environment deployed via Airship crosses many security domains. This guide explains many of the security concerns that have been reviewed and considered by the Airship developers. Because Airship is a highly configuration-driven platform, there is some onus on the end-user to make good decisions with their configuration.

4.2.1 Layout and Nomenclature

Each topic in the security guide will provide some overview for scope of that topic and then provide a list of tactical security items. For each item two statuses will be listed as well as the project scope.

- Project Scope: Which Airship projects address this security item.
- Solution: The solution is how this security concern is addressed in the platform
  - Remediated: The item is solved for automatically
  - Configurable: The item is based on configuration. Guidance will be provided.
  - Mitigated: The item currently mitigated while a permanent remediation is in progress.
  - Pending: Addressing the item is in-progress
- Audit: Auditing the item provides for ongoing monitoring to ensure there is no regression
  - Testing: The item is tested for in an automated test pipeline during development
  - Validation: The item is reported on by a validation framework after a site deployment
  - Pending: Auditing is in-progress

4.2.2 Airship Security Topics

Template for a Security Guide Topic

Updated: 1-AUG-2018

An overview of the scope of this topic.

Contents

Security Item List

Sensitive Data Security

Sensitive data should be encrypted at-rest.

- Project Scope: Deckhand
- Solution *Remediated*: The `storagePolicy` metadata determines if Deckhand will persist document data encrypted.
- Audit: *Testing*: Pipeline test checks that documents with a `storagePolicy: encrypted` are not persisted to the database with an intact `data` section.

Sensitive data should be encrypted in-transit.

- Project Scope: Shipyard, Deckhand
- Solution *Pending*: Shipyard and Deckhand API endpoints should support TLS. See *data_security*.
- Audit: *Pending*: Expect to validate post-deployment that endpoints all support TLS

Configuration Guidance

For items that require guidance on configuration that impact a security item please list an item here. Use RST anchors and links to link the security item solution status to this guidance.

Temporary Mitigation Status

Data Security In-Transit

Current work to support Deckhand enabling TLS termination, Shipyard enabling self-signing CAs and Barbican supporting TLS termination.

References

Transport Layer Security (TLS)

HAProxy Security Guide

Updated: 13-AUG-2018

This guide covers configurations for HAProxy. Specifically, in `mode tcp`.

Contents

Security Item List

TCP Mode

The instance will work in pure TCP mode. A full-duplex connection will be established between clients and servers, and no layer 7 examination will be performed. This is the default mode. It should be used for TLS.

Max Connections

Set `maxconn` in `global` to a reasonable level. HAProxy will queue requests beyond that value.

Set Headers

"set-header" does the same as "add-header" except that the header name is first removed if it existed. This is useful when passing security information to the server, where the header must not be manipulated by external users. Note that the new value is computed before the removal so it is possible to concatenate a value to an existing header.

References

HAProxy Configuration Guide

Canonical Ubuntu/MAAS Security Guide

Updated: 6-AUG-2018

This guide covers the configuration of MAAS to run securely and to deploy secure installations of Ubuntu 16.04.x. Some items are above and beyond MAAS when MAAS does not offer the functionality needed to fully secure a newly provisioned server.

Contents

Security Item List

Filesystem Permissions

Many files on the filesystem can contain sensitive data that can hasten a malignant attack on a host. Ensure the below files have appropriate ownership and permissions

Filesystem Path	Owner	Group	Permissions
`/boot/System.map-*`	root	root	`0600`
`/etc/shadow`	root	shadow	`0640`
`/etc/gshadow`	root	shadow	`0640`
`/etc/passwwd`	root	root	`0644`
`/etc/group`	root	root	`0644`
`/var/log/kern.log`	root	root	`0640`
`/var/log/auth.log`	root	root	`0640`
`/var/log/syslog`	root	root	`0640`

- Project Scope: Drydock
- Solution *Configurable*: A bootaction will be run to enforce this on first boot
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin

Filesystem Partitioning

The mounts `/tmp`, `/var`, `/var/log`, `/var/log/audit` and `/home` should be individual file systems.

- Project Scope: Drydock
- Solution *Configurable*: Drydock supports user designed partitioning, see *Filesystem Configuration*.
- Audit: *Testing*: The Airship testing pipeline will validate that nodes are partitioned as described in the site definition.

Filesystem Hardening

Disallow symlinks and hardlinks to files not owned by the user. Set `fs.protected_symlinks` and `fs.protected_hardlinks` to `1`.

- Project Scope: Diving Bell
- Solution *Configurable*: Diving Bell overrides will enforce this kernel tunable. By default MAAS deploys nodes in compliance.
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin.

Execution Environment Hardening

The kernel tunable `fs.suid_dumpable` must be set to `0` and there must be a hard limit disabling core dumps (`hard core 0`)

- Project Scope: DivingBell, Drydock
- Solution *Configurable*: Diving Bell overrides will enforce this kernel tunable, by default MAAS deploys nodes with `fs.suid_dumpable = 2`. A boot action will put in place the hard limit.
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin

Randomizing stack space can make it harder to exploit buffer overflow vulnerabilities. Enable the kernel tunable `kernel.randomize_va_space = 2`.

- Project Scope: DivingBell
- Solution *Configurable*: Diving Bell overrides will enforce this kernel tunable, by default MAAS deploys nodes in compliance.
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin

Mandatory Access Control

Put in place the approved default AppArmor profile and ensure that Docker is configured to use it.

- Project Scope: Drydock, Promenade
- Solution *Configurable*: A bootaction will put in place the default AppArmor profile. Promenade will deploy a Docker configuration to enforce the default policy.
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin probing `/proc/<pid>/attr/current`.

Put in place an approved AppArmor profile to be used by containers that will manipulate the on-host AppArmor profiles. This allows an init container in Pods to put customized AppArmor profile in place and load them.

- Project Scope: Drydock
- Solution *Configurable*: A bootaction will put in place the profile-manager AppArmor profile and load it on each boot.
- Audit: *Pending*: The availability of this profile will be verified by a Sonobuoy plugin.

Important: All other AppArmor profiles must be delivered and loaded by an init container in the Pod that requires them. The Pod must also be decorated with the appropriate annotation to specify the custom profile.

System Monitoring

Run *rsyslogd* to log events.

- Project Scope: Drydock
- Solution *Remediated*: MAAS installs rsyslog by default.
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin.

Run a monitor for logging kernel audit events such as auditd.

- Project Scope: Non-Airship
- Solution *Remediated*: The Sysdig Falco will be used and
- Audit: *Pending*: This will be verified on an ongoing basis via a Sonobuoy plugin.

Watch the watchers. Ensure that monitoring services are up and responsive.

- Project Scope: Non-Airship
- Solution *Remediated*: Nagios will monitor host services and Kubernetes resources
- Audit: *Validation*: Internal corporate systems track Nagios heartbeats to ensure Nagios is responsive

Blacklisted Services

The below services are deprecated and should not be enabled or installed on hosts.

Service	Ubuntu Package
telnet	telnetd
inet telnet	inetutils-telnetd
SSL telnet	telnetd-ssl
NIS	nis
NTP date	ntpdate

- Project Scope: Drydock
- Solution *Configurable*: A boot action will be used to enforce this on first boot.
- Audit: *Pending*: This will be verified on an ongoing basis via Sonobuoy plugin.

Required System Services

`cron` and `ntpd` **must** be installed and enabled on all hosts. Only administrative accounts should have access to cron. `ntpd -q` should show time synchronization is active.

- Project Scope: Drydock
- Solution *Remediated*: A MAAS deployed node runs cron and configured ntpd by default.
- Audit: *Pending*: This will be verified on an ongoing basis via Sonobuoy plugin.

System Service Configuration

If `sshd` is enabled, ensure it is securely configured:

- **Must** only support protocol version 2 (`Protocol 2`)
- **Must** disallow root SSH logins (`PermitRootLogin no`)
- **Must** disallow empty passwords (`PermitEmptyPasswords no`)
- **Should** set a idle timeout interval (`ClientAliveInterval 600` and `ClientAliveCountMax 0`)
- Project Scope: Drydock
- Solution *Configurable*: A boot action will install an explicit configuration file
- Audit: *Pending*: This will be verified on an ongoing basis via Sonobuoy plugin.

Network Security

Important: Calico network policies will be used to secure host-level network access. Nothing will be orchestrated outside of Calico to enforce host-level network policy.

Secure the transport of traffic between nodes and MAAS/Drydock during node deployment.

- Project Scope: Drydock, MAAS

- Solution *Pending*: The Drydock and MAAS charts will be updated to include an Ingress port utilizing TLS 1.2 and a publicly signed certificate. Also the service will enable TLS on the pod IP.
- Audit: *Testing*: The testing pipeline will validate the deployment is using TLS to access the Drydock and MAAS APIs.

Danger: Some traffic, such as iPXE, DHCP, TFTP, will utilize node ports and is not encrypted. This is not configurable. However, this traffic traverses the private PXE network.

Secure Accounts

Enforce a minimum password length of 8 characters

- Project Scope: Drydock
- Solution *Configurable*: A boot action will update `/etc/pam.d/common-password` to specify `minlen=8` for `pam_unix.so`.
- Audit: *Pending*: This will be verified on an ongoing basis via Sonobuoy plugin.

Configuration Guidance

Filesystem Configuration

The filesystem partitioning strategy should be sure to protect the ability for the host to log critical information, both for security and reliability. The log data should not risk filling up the root filesystem (`/`) and non-critical log data should not risk crowding out critical log data. If you are shipping log data to a remote store, the latter concern is less critical. Because Airship nodes are built to **ONLY** run Kubernetes, isolating filesystems such as `/home` is not as critical since there is no direct user access and applications are running in a containerized environment.

Temporary Mitigation Status

References

- OpenSCAP for Ubuntu 16.04
- Ubuntu 16.04 Server Guide
- Canonical MAAS 2.3 TLS
- Canonical MAAS 2.4 TLS

4.3 Getting Started for Airship Developers

Airship uses many foundational concepts that should be understood by developers wanting to get started. This documentation attempts to provide a survey of those topics.

4.3.1 Concepts

- Containers/Docker
- RESTful APIs
- YAML
- Security

Containers/Docker

Airship is, at its core, intended to be used in a containerized fashion. Dockerfile resources exist in each of the project repositories that are used by the build process to generate Docker images. Images are hosted on quay.io under `airshipit`.

Each main component is responsible for generating one or more images (E.g.: Shipyard produces a Shipyard image and an Airflow image).

When running, nearly every aspect of Airship runs as a container, and Airship (primarily Promenade + Armada) sets up many of the other foundational components as containers, including many Kubernetes components, etcd, Calico, and Ceph.

RESTful APIs

Each Airship component that runs as a service provides a RESTful API. Some *API Conventions* exist explaining the basic format of requests and responses and required endpoints that are exposed, such as health check and and design validation.

YAML

The YAML document format is used along with JSON Schema to define the declarative site and software design inputs to the Airship components.

Security

Security is a consideration from the ground-up for Airship components. Some technologies in this space are TLS and Keystone auth. Airship APIs are protected by RBAC policies implemented with oslo.policy (with some exceptions for basic health checking and listing of API versions). Keystone middleware serves as a layer in the pipeline of service layers for each component, providing lookup of authenticated users, resolving their roles, which are then checked. Access enforcement is within the Airship components, using a decorator for each API that requires limited access.

4.3.2 Environment

- Helm
- Kubernetes
- Linux

Helm

Airship components are deployed into Kubernetes using Armada, which in turn uses the Tiller component of Helm. Helm charts are used to generate the Kubernetes artifacts (deployments, jobs, configmaps, etc...).

Kubernetes

Airship is thoroughly intertwined with Kubernetes:

- Airship depends on Kubernetes as the orchestrator of the containers that make up the platform.
- Airship sets up a single node Kubernetes instance during the Promenade genesis process, with the necessary configuration to become the seed of a resilient Kubernetes cluster during later stages of Airship.
- Airship's components run as containers inside the Kubernetes cluster.

Linux

Airship is targeted to a Linux platform. There are significant elements of Airship that use shell scripts to drive processes.

4.3.3 Coding

Further information is available in *Code and Project Conventions*.

Airship is primarily a combination of Python 3 and shell scripting. There are several Python libraries that are used in common across many components:

- Falcon: A service framework providing the API endpoints.
- uWSGI: The service container.
- oslo_config: Provides per-deployment, configuration file configurability.
- oslo_policy: Provides RBAC support for API endpoints (and more).
- Requests: A framework for making HTTP requests and receiving responses.
- Click: A CLI framework used to provide component-level Command Line Interfaces.

Each component also brings in their own dependencies as needed.

4.3.4 Database(s)

Several of the Airship components require some data persistence. Some data persistence is achieved by utilizing Kubernetes provided mechanisms, and the Keystone software uses a MariaDB instance, but most is accomplished using a containerized PostgreSQL database.

Interaction with PostgreSQL uses the following:

- SQLAlchemy: A python library providing most of the needed database functionality.
- Alembic: Version management for database schemas and data.
- oslo_db: An OpenStack layer providing additional functionality over SQLAlchemy.

4.3.5 Testing

- Unit
- Functional
- Integration

Unit and functional tests are used in the gating of changes before merging code. Unit tests utilize combinations of pytest and stestr. Functional tests utilize Gabbi. These tools are not exclusive of others, but are the primary tools being used for unit and functional tests.

Integration testing is orchestrated in the merge gates, and uses various means of testing.