
Airship Project Documentation

Airship Team

Mar 18, 2021

LEARN ABOUT AIRSHIP

| | | |
|----------|---------------------------------|-----------|
| 1 | About this Documentation | 3 |
| 2 | Get Involved | 23 |

Airship is a collection of components that declaratively configure, deploy and maintain a [Kubernetes](#) environment defined by [YAML](#) documents. Airship is supported by the [OpenStack Foundation](#).

ABOUT THIS DOCUMENTATION

Airship documentation serves the entire community with resources for users and developers.

1.1 Airship Security Vulnerability Management

The Airship community is committed to expediently confirming, resolving, and disclosing all reported security vulnerabilities. We appreciate your cooperation and participation in our vulnerability management process outlined below.

1.1.1 Report a Vulnerability

If you discover a vulnerability in an Airship project, please treat the issue with a sense of confidentiality and disclose it to the [airship-security mailing list](mailto:airship-security@lists.airshipit.org):

airship-security@lists.airshipit.org

Additionally, please include any potential fixes, as doing so can expedite the disclosure and patching processes.

The Airship Working Committee is the sole subscriber of the [airship-security mailing list](mailto:airship-security@lists.airshipit.org) and monitors it for reported vulnerabilities. The committee confirms or rejects reported vulnerabilities in correspondence with the vulnerability reporter. In the event that the Airship Working Committee does not have the expertise or availability to resolve a reported vulnerability, the committee may solicit assistance from outside contributors to better facilitate the understanding and resolution of reported security vulnerabilities.

1.1.2 Receive Early Disclosures

We prefer to disclose confirmed security vulnerabilities as soon as possible. While circumstances may not always allow immediate disclosure, vulnerabilities may be disclosed over the [airship-embargo-notice mailing list](mailto:airship-embargo-notice@lists.airshipit.org) when a fix becomes available. The airship-embargo-notice mailing list notifies Airship users of confirmed vulnerabilities. If you operate Airship in a production environment, we recommend subscribing to the [airship-embargo-notice mailing list](mailto:airship-embargo-notice@lists.airshipit.org) by contacting the Airship Working Committee. The Airship Working Committee evaluates subscription requests on a case-by-case basis.

1.1.3 Receive Public Disclosures

Within ninety days of the initial vulnerability report, except in unusual circumstances, the Airship Working Committee will publicly disclose the reported vulnerability and its mitigation over the [airship-announce](mailto:airship-announce@lists.airshipit.org) and [airship-discuss](mailto:airship-discuss@lists.airshipit.org) mailing lists. If a fix merges before the aforementioned ninety day period expires, the Airship Working Committee will instead disclose the vulnerability and fix twenty-one days later. We recommend subscribing to both mailing lists in order to receive security updates.

1.2 Getting Started

Thank you for your interest in Airship. Our community is eager to help you contribute to the success of our project and welcome you as a member of our community!

We invite you to reach out to us at any time via the [Airship mailing list](#) or on our [Slack workspace](#).

Welcome aboard!

1.3 Airship 2 Development

Development is underway on Airship 2: the educated evolution of Airship 1, designed with our experience using Airship in production. Airship 2 makes the Airship control plane ephemeral, leverages entrenched upstream projects such as the [Cluster API](#), [Metal Kubed](#), [Kustomize](#), and [kubeadm](#), and embraces Kubernetes Custom Resource Definitions (CRDs). To learn more about the Airship 2.0 evolution, see the [Airship 2 evolution blog series](#).

Each Airship 2 project has its own development guidelines. Join the ongoing Airship 2 development by referencing the [airshipctl](#) or the [airshipui](#) documentation.

1.4 Airship 1 Development

Airship 1 is a collection of open source tools that automate cloud provisioning and management. Airship provides a declarative framework for defining and managing the life cycle of open infrastructure tools and the underlying hardware. These tools include OpenStack for virtual machines, Kubernetes for container orchestration, and MaaS for bare metal, with planned support for OpenStack Ironic.

Improvements are still made Airship 1 daily, and we welcome additional contributors. We recommend that new contributors begin by reading the high-level architecture overview included in our [treasuremap](#) documentation. The architectural overview introduces each Airship component, their core responsibilities, and their integration points.

1.4.1 Deep Dive

Each Airship component is accompanied by its own documentation that provides an extensive overview of the component. With so many components, it can be challenging to find a starting point.

We recommend the following:

1.4.2 Try an Airship environment

Airship provides two single-node environments for demo and development purpose.

[Airship-in-a-Bottle](#) is a set of reference documents and shell scripts that stand up a full Airship environment with the execution of a script.

[Airskiff](#) is a light-weight development environment bundled with a set of deployment scripts that provides a single-node Airship environment. Airskiff uses minikube to bootstrap Kubernetes, so it does not include Drydock, MaaS, or Promenade.

Additionally, we provide a reference architecture for easily deploying a smaller, demo site.

[Airsloop](#) is a fully-authored Airship site that can be quickly deployed as a baremetal, demo lab.

1.4.3 Focus on a component

When starting out, focusing on one Airship component allows you to become intricately familiar with the responsibilities of that component and understand its function in the Airship integration. Because the components are modeled after each other, you will also become familiar with the same patterns and conventions that all Airship components use.

Airship source code lives in the [OpenDev Airship namespace](#). To clone an Airship project, execute the following, replacing `<component>` with the name of the Airship component you want to clone.

Refer to the component's documentation to get started. A list of each component's documentation is listed below for reference:

- [Armada](#)
- [Deckhand](#)
- [Divingbell](#)
- [Drydock](#)
- [Pegleg](#)
- [Promenade](#)
- [Shipyard](#)

1.4.4 Testing Changes

Testing of Airship changes can be accomplished several ways:

1. Standalone, single component testing
2. Integration testing
3. Linting, unit, and functional tests/linting

Note: Testing changes to charts in Airship repositories is best accomplished using the integration method describe below.

1.4.5 Standalone Testing

Standalone testing of Airship components, i.e. using an Airship component as a Python project, provides the quickest feedback loop of the three methods and allows developers to make changes on the fly. We recommend testing initial code changes using this method to see results in real-time.

Each Airship component written in Python has pre-requisites and guides for running the project in a standalone capacity. Refer to the documentation listed below.

- [Armada](#)
- [Deckhand](#)
- [Drydock](#)
- [Pegleg](#)
- [Promenade](#)
- [Shipyard](#)

1.4.6 Integration Testing

While each Airship component supports individual usage, Airship components have several integration points that should be exercised after modifying functionality.

We maintain several environments that encompass these integration points:

1. *Airskiff*: Integration of Armada, Deckhand, Shipyard, and Pegleg
2. *Airship-in-a-Bottle Multinode*: Full Airship integration

For changes that merely impact software delivery components, exercising a full Airskiff deployment is often sufficient. Otherwise, we recommend using the Airship-in-a-Bottle Multinode environment.

Each environment's documentation covers the process required to build and test component images.

1.4.7 Final Checks

Airship projects provide Makefiles to run unit, integration, and functional tests as well as lint Python code for PEP8 compliance and Helm charts for successful template rendering. All checks are gated by Zuul before a change can be merged. For more information on executing these checks, refer to project-specific documentation.

Third party CI tools, such as Jenkins, report results on Airship-in-a-Bottle patches. These can be exposed using the "Toggle CI" button in the bottom left-hand page of any gerrit change.

1.4.8 Pushing code

Airship uses the [OpenDev gerrit](#) for code review. Refer to the [OpenStack Contributing Guide](#) for a tutorial on submitting changes to Gerrit code review.

1.4.9 Next steps

Upon pushing a change to gerrit, Zuul continuous integration will post job results on your patch. Refer to the job output by clicking on the job itself to determine if further action is required. If it's not clear why a job failed, please reach out to a team member in IRC. We are happy to assist!

Assuming all continuous integration jobs succeed, Airship community members and core developers will review your patch and provide feedback. Many patches are submitted to Airship projects each day. If your patch does not receive feedback for several days, please reach out using IRC or the Airship mailing list.

1.4.10 Merging code

Like most OpenDev projects, Airship patches require two +2 code review votes from core members to merge. Once you have addressed all outstanding feedback, your change will be merged.

1.4.11 Beyond

Congratulations! After your first change merges, please keep up-to-date with the team. We hold two weekly meetings for project and design discussion:

Our weekly #airshipit IRC meeting provides an opportunity to discuss project operations.

Our weekly design call provides an opportunity for in-depth discussion of new and existing Airship features.

For more information on the times of each meeting, refer to the [Airship wiki](#).

1.5.1 Language

- ‘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.

1.5.2 Conventions and Standards

- 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.

7

(continued from previous page)

```
"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 explicitly 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)

X-End-User The user name of the initial invoker that will be carried on all logs for user tracing cross components. Shipyard doesn't support this header and when passed, it will be ignored.

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": [
      { "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 validation 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 indicates 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": {
    "errorCode": {{n}},
    "messageList": [
      { "message" : "{{Detailed Health Check failure information}}",
        "error": true,
        "kind": "SimpleMessage" },
      ...
    ]
  },
  "code": 503
}
```

Success message example:

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

(continues on next page)

(continued from previous page)

```
"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"
  },
  "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 [Docker](#) images will keep Dockerfiles 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 Dockerfiles that can be used to generate images.

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

- foo/images/foo contains Dockerfiles for `foo`
- foo/images/bar contains Dockerfiles for `bar`

Each image must include the following set of labels conforming to the [OCI image annotations standard](#) as the minimum:

```
org.opencontainers.image.authors='airship-discuss@lists.airshipit.org, irc://
↪#airshipit@freenode'
org.opencontainers.image.url='https://airshipit.org'
org.opencontainers.image.documentation='<documentation on readthedocs or in_
↪repository URL>'
org.opencontainers.image.source='<repository URL>'
org.opencontainers.image.vendor='The Airship Authors'
org.opencontainers.image.licenses='Apache-2.0'
org.opencontainers.image.revision='<Git commit ID>'
org.opencontainers.image.created='UTC date and time in RFC3339 format with seconds'
org.opencontainers.image.title='<image name, e.g. "armada">'
```

Last three annotations (revision, created and title), being dynamic, are added on a container build stage. Others are statically defined in Dockerfiles. Optional custom `org.airshipit.build=community` annotation is added to the Airship images published to the community.

Image tags must follow format:

- `:<full Git commit ID>_<distro_suffix>`
- `:<branch>_<distro_suffix>` - latest image built from specific branch
- `:latest_<distro_suffix>` - latest image built from master

The `<distro suffix>` (e.g. `_ubuntu_xenial`) could be omitted. See [Airship Multiple Linux Distribution Support](#) specification for details.

Images should follow best practices for the container images. Be slim and secure in particular.

Dockerfile

Dockerfile file names must follow format: `Dockerfile.<distro_suffix>`, where `<distro_suffix>` matches corresponding image tag suffix. The `.<distro_suffix>` could be omitted where not relevant.

Lines should be indented by a space character next to the Dockerfile instruction block they correspond to.

Dockerfile must allow base image substitution via `FROM` argument. This is to allow the use of base images stored in third-party or internal repositories.

Dockerfile should follow best practices. Use multistage container builds where possible to reduce image size and attack surface. `RUN` statements should pass linting via `shellcheck`. You may use available Dockerfile linters, if you wish to do so.

See [example Dockerfile](#) file for reference.

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 individual `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
- `format` to invoke automated code formatting specific to the project's code language (e.g. Python for armada and Go for airshipctl) as listed in the Linting and Formatting Standards.

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 | Tools Used |
| Ansible | ansible-lint |
| Bash | Shellcheck |
| Go | gofmt |
| Markdown | markdownlint |
| Python | YAPF, Flake8 |

Ansible formatting

Ansible code should be linted to be conformant to the standards checked by [ansible-lint](#) project.

Bash Formatting

Bash shell scripts code should be linted to be conformant to the standards checked by [Shellcheck](#) project.

Bash shell scripts code in Helm templates should ideally be linted as well, however gating of it is a noble goal and is only desired.

Go Formatting

Go code should be formatted using gofmt. When using gofmt be sure to use the `-s` flag to include simplification of code for example:

```
gofmt -s /path/to/file.go
```

Markdown Formatting

Markdown code (documentation) should be linted to be conformant to the standards checked by [markdownlint](#) project.

Python PEP-8 Formatting

Python should be formatted via YAPF. The knobs for YAPF can be specified in the project's root directory in `style.yapf`. The contents of this file should be:

```
[style]
based_on_style = pep8
spaces_before_comment = 2
column_limit = 79
blank_line_before_nested_class_or_def = false
blank_line_before_module_docstring = true
split_before_logical_operator = true
split_before_first_argument = true
allow_split_before_dict_value = false
split_before_arithmetic_operator = true
```

A sample Flake8 section is below, for use in `tox.ini`, and is the method of enforcing import orders via Flake8 extension `flake8-import-order`:

```
[flake8]
filename = *.py
show-source = true
# [H106] Don't put vim configuration in source files.
# [H201] No 'except:' at least use 'except Exception:'
# [H904] Delay string interpolations at logging calls.
enable-extensions = H106,H201,H904
# [W503] line break before binary operator
ignore = W503
exclude=.venv,.git,.tox,build,dist,*lib/python*,*egg,tools,*.*ini,*.*po,*.*pot
max-complexity = 24
```

Airship components must provide for automated checking of their formatting standards, such as the lint step noted above in the Makefile, and in the future via CI jobs. Components may provide automated reformatting.

YAML Schema

YAML schema defined by Airship should have key names that follow camelCase naming conventions.

Note that Airship also integrates and consumes a number of projects from other open source communities, which may have their own style conventions, and which will therefore be reflected in Airship deployment manifests. Those fall outside the scope of these Airship guidelines.

Any YAML schema that violate this convention at the time of this writing (e.g. with snake_case keys) may be either grandfathered in, or converted, at the development team's discretion.

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 alternative 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
|       |       |- 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.

CRD Conventions

Airship will use CRDs to enrich the Kubernetes API with Airship-specific document schema. Airship projects will follow the following conventions when defining Custom Resource Definitions (CRDs).

Note that Airship integrates and consumes a number of projects from other open source communities, which may have their own style conventions, and which will therefore be reflected in Airship deployment manifests. Those fall outside the scope of these Airship guidelines.

In general, Airship will follow the [Kubernetes API Conventions](#) when defining CRDs. These cover naming conventions (such as using camelCase for key names), expected document structure, HTTP request verbs and status codes, and behavioral norms.

Exceptions or restrictions from the Kubernetes conventions are specified below; this list may grow in the future.

- The `apiGroup` (and `apiVersion`) for Airship CRDs will have values following the convention `<function>.airshipit.org`.

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 - capabilities 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:

- Architecture and Design - features and structure of the software
- Inline, Code, Method - documentation specific to the functions 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](#)

Issue Tracking

Issues for the Airship Project are tracked on a per-project basis using Github Issues. All feature requests and bugs should be submitted to the respective project's Github Issues board for community evaluation and action. For additional details on a project's issue tracking workflow, see its CONTRIBUTING.md file.

Project CONTRIBUTING.md Files:

- [airshipctl CONTRIBUTING.md](#)

Submitting Issues

Issues can be submitted by navigating to a project's Github Issue page and selecting "New issue".

Depending on the project, you may be prompted to select an issue type. These selections are associated with templates to aid in the issue creation process. If there are no templates associated with the project, utilize the following templates:

- [Feature Request Template](#)
- [Bug Report Template](#)

When submitting issues, be descriptive and concise. If the issue is complex consider breaking it down into smaller issues so it can be more easily addressed by the community.

Grooming Issues

Issues are groomed by each project's core reviewers. Depending on how active the project is and their workflow, it can take up to two weeks for issues to be groomed. Issues will be marked with the "triage" label until they have been

groomed and prioritized. We encourage developers to not work on issues marked as “triage” as these issues are not guaranteed to be accepted by the project.

Each project will have a multitude of labels used to help categorize issues and improve organization. Labels will differ from project to project, but some common labels may include:

- Process labels (“ready for review”, “wip”, “blocked”, “triage”)
- Priority labels (“priority/low”, “priority/medium”, “priority/high”)
- Component labels (“component/engine”, “component/cli”, etc...)
- Type labels (“feature”, “bug”, “epic”)

As many applicable labels as possible should be applied to issues. These labels should be consistently reevaluated and updated as the issue evolves.

Issue Lifecycle

The general lifecycle for issues is as follows:

1. The issue is created by a member of the community on Github Issues. By default, the issues should have a type label and be labeled as “triage”. Neither the submitter nor any other developer should work on the issue at this time, but they may discuss it on the Github Issue board to further clarify the issue.
2. The issue is groomed by members of the core reviewer team for the project, either on a scheduled grooming call or asynchronously on the Github Issues board. During the grooming process the core reviewer team will establish whether the issue is part of the project’s scope and what the issue’s priority level is. They will also apply a multitude of labels to help improve the organization of the issue board and the visibility of the issue. At the end of the process if the issue is accepted, the “triage” label will be removed and the issue will be available to be assigned and worked on. It is possible that the core reviewer team will need additional details. Please be attentive to updates on created issues to ensure they are addressed in a timely manner.
3. The issue is assigned to a member of the community to be addressed. Only core reviewers have the ability to assign issues. If a member of the community wishes to be assigned to an issue, they should make a comment on the issue requesting to have it assigned to them. Please only start work on an issue after it is assigned to you to decrease the risk of duplicate changes. Issue assignees are encouraged to use process labels to indicate where they are in the development workflow (i.e. “wip” or “ready for review”). Consistent updates let the core reviewer team know that the issue is still active. If no measurable work has been performed on an assigned issue within two weeks, the issue may be considered “stale” and the assignee may be removed from the issue.
4. Once a developer completes work on an issue’s associated change (or research), they should indicate that work is complete and “ready for review”. Include a link to the associated change along with any observations of worth in the issue’s comments. Once the change is merged, the core reviewer team will mark the issue as completed.

Github Issues Bot

To help improve Gerrit and Github integration, Airship utilizes a python daemon to update Github Issues with Gerrit change details. The utility is hosted on Github under the name [Gerrit-to-Github-Issues](#). Currently the bot only runs against the airshipctl project, but it may be implemented later into other projects.

The bot performs the following functions:

- Comments on issues with the information of active Gerrit changes including links, review statuses, and author information.
- Updates issue process labels with the “wip” label if “DNM” or “WIP” are present in the change’s commit message and “ready for review” in all other cases.

To leverage the bot's full functionality, be sure to include a reference for the issue you are addressing from GitHub Issues inside the change commit message. There are three ways of doing this:

#. Add a statement in your commit message in the format of `Relates-To: #X`. This will add a link on issue “#X” to your change. #. Add a statement in your commit message in the format of `Closes: #X`. This will add a link on issue “#X” to your change and will close the issue when your change merges. #. Add a bracketed tag at the beginning of your commit message in the format of `[#X] <begin commit message>`. This will add a link on issue “#X” to your change. This method is considered a fallback in lieu of the other two methods.

Any issue references should be evaluated within 15 minutes of being uploaded.

For any questions, comments, or requests please reach out to Ian Pittwood either at [ian-pittwood](#) on the Airship IRC/Slack or at pittwoodian@gmail.com.

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 output 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 8-4-4-12 format.
- ExternalContextID is the UUID assigned from the external source (or generated for the same purpose), in 8-4-4-12 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.

1.6 Other Resources

- [Airship Blog](#)
- [Airship Website](#)
- [Airship Wiki](#)

GET INVOLVED

2.1 Join our mailing lists

Receive Airship announcements and interact with our community on our [mailing lists](#).

2.2 Join our weekly calls

Airship is constantly evolving. Contribute to the Airship design process and day-to-day community operations in our [weekly calls](#).

2.3 Join our Slack workspace

Get in touch with Airship developers and operators in our [Slack workspace](#).